A Data Locality Aware Online Scheduling Approach for I/O-Intensive Jobs with File Sharing *

Gaurav Khanna[1], Umit Catalyurek[2],
Tahsin Kurc[2], P. Sadayappan[1], Joel Saltz[2]

[1] Dept. of Computer Science and Engineering
{khannag, saday}@cse.ohio-state.edu
[2] Dept. of Biomedical Informatics
{umit,kurc}@bmi.osu.edu, Joel.Saltz@osumc.edu
The Ohio State University

Abstract. Many scientific investigations have to deal with large amounts of data from simulations and experiments. Data analysis in such investigations typically involves extraction of subsets of data, followed by computations performed on extracted data. Scheduling in this context requires efficient utilization of the computational, storage and network resources to optimize response time. The data-intensive nature of such applications necessitates data-locality aware job scheduling algorithms. This paper proposes a hypergraph based dynamic scheduling heuristic for a stream of independent I/O intensive jobs with file sharing behavior. The proposed heuristic is based on an event-driven, run-time hypergraph modeling of the file sharing characteristics among jobs. Our experiments on a coupled compute/storage cluster show it performs better compared to previously proposed strategies, under a varying set of parameters for workloads from the application domain of biomedical image analysis.

1 Introduction

Data-driven approaches that make use of large datasets to solve complex problems in science and engineering have become increasingly important. Data analysis is a key component in data-driven science and engineering, enabling a better understanding of the problem under study and more efficient refinement of the search space of solutions. Data analysis applications often involve access and processing of many subsets of a dataset. Most scientific datasets are stored in files. A request for the region of interest specifies a subset of data files and/or segments in data files – either directly as input parameters or after an index lookup that finds the files and file segments of interest. The data of interest is retrieved from the storage system and transformed into a data product that is more suitable for examination by the scientist.

When several data-intensive jobs are submitted to a high-performance system, they have to be scheduled to compute nodes for execution. Unlike traditional compute intensive jobs, data analysis jobs may require access to a large

* This research was supported in part by the National Science Foundation under Grants #CCF-0342615 and #CNS-0403342.

number of files and high data volume. When mapping such data-intensive jobs to compute nodes, scheduling mechanisms need to take into account not only the computation time of the jobs, but also the overheads of retrieving files requested by those jobs. Moreover, the staging of files should be carefully coordinated to minimize I/O overheads. Traditional job schedulers for compute-intensive jobs running at supercomputer centers are not designed for data intensive jobs, since they take into account CPU related metrics (e.g. user estimated job run times) and system state (e.g. queue wait times) for making scheduling decisions, but they do not take into account data related metrics.

This paper addresses the efficient execution of a stream of dynamically arriving data-intensive jobs exhibiting file-shared I/O behavior [14]. In our model, the files required by the jobs are initially resident on a storage cluster. When a job is scheduled to a compute node, the files accessed by the job are staged from remote storage nodes to the compute node before the job is executed. Since disk space on compute nodes is limited, effective management of data on the local disk of compute nodes is also important. Obviously, by running jobs on the storage nodes the cost of data staging can be avoided; however, in real setups storage nodes are designed to maximize storage space and I/O bandwidth, and have only limited computation power[3]. Thus, we assume that jobs cannot be executed directly on storage nodes.

We propose a new algorithm to schedule a stream of dynamically arriving jobs that share input files. The algorithm is based on a hypergraph formulation of the workload and a K-way partitioning of the hypergraph to yield a locality aware and load-balanced allocation of jobs on the compute cluster. The proposed approach formulates the sharing of files among jobs as a hypergraph. The hypergraph representation also models the load on the compute nodes due to currently executing jobs. It also takes into account the fact that some files might already have been staged or are currently being staged to the compute nodes due to previously executed or currently running jobs. The experimental results show that when there is high degree of file sharing among jobs, our formulation results in much better schedules compared to the *JobDataPresent + DataLeastLoaded* algorithm [13] and the *Minimum Execution Time*, *Minimum Completion Time*, *Switching Algorithm* heuristics [10, 2, 3], modified to handle data intensive jobs. We have also observed that as the average job inter-arrival times decrease, the proposed approach outperforms the other heuristics.

2 Problem Definition and Use-case Applications

We target streams of dynamically arriving jobs which consist of independent sequential programs. Each job requests a subset of data files from a dataset

[3] Even though per node storage nodes might have comparable power to compute nodes, generally the number of storage nodes are much less than the number of compute nodes. For example, at Ohio Supercomputer Center 0.5 Petabyte Mass Storage System is derived from 24 storage nodes, whereas they have thousands of compute nodes.

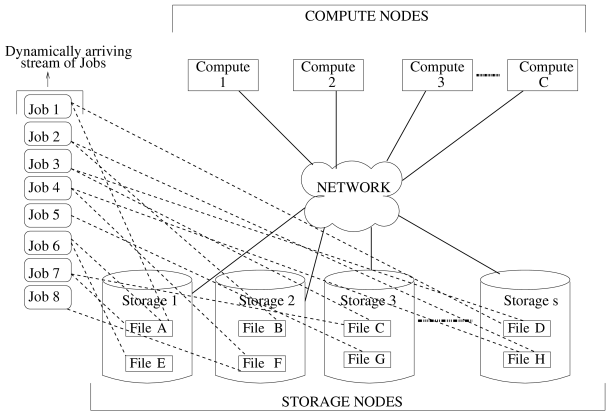


Fig. 1. Scheduling problem.

and can be executed on any of the nodes in the compute cluster. The data files required by a job should be staged to the compute node where the job is allocated for the job to execute correctly; a data file is the unit of I/O transfer from the storage cluster to the compute cluster. The jobs may share a number of files with previously scheduled jobs or with jobs arriving in future.

Our objective is, given a stream of dynamically arriving jobs and a set of files required by these jobs, 1) to schedule the jobs in an efficient manner, 2) to decide which files need to be remotely transferred and their respective destination nodes, so as to minimize the average job response time. Figure 1 depicts an illustration of this problem. Each job in the job stream is represented by a computation weight, a list of input files, and their file sizes.

Formally, let $S =< j_1, j_2, \ldots, j_n >$ be a stream of n jobs arriving dynamically. Let $Arrival(j_i)$ be the arrival time of the job j_i and $Exec(j_i)$ be the total time the job j_i spends in execution. Some of the jobs will not be able start execution as soon as they have been submitted. Let $Start(j_i)$ be the time instant when the job j_i starts execution. In our case, this corresponds to the case when the first data transfer for the job j_i starts. If the job finds all its files locally, then it is the time when the job starts its computation. The wait time of a job $Wait(j_i)$ is the time it spends in the queue before it starts execution.

$$Wait(j_i) = Start(j_i) - Arrival(j_i) \tag{1}$$

The response time $Response(j_i)$ of the job is the turnaround time which refers to the total time spent by job in the queue and in execution.

$$Response(j_i) = Wait(j_i) + Exec(j_i) \tag{2}$$

$Completion(j_i)$ refers to the instant when the job finishes execution.

$$Completion(j_i) = Arrival(j_i) + Response(j_i) \tag{3}$$

And the $AverageResponseTime$ is defined as the overall average of response times of the jobs in the stream.

$$AverageResponseTime = \frac{\sum_{i=1}^{i=n} Response(j_i)}{n} \quad (4)$$

We have evaluated our approach using application scenarios from **Biomedical Image Analysis** application class. Biomedical imaging is a powerful method for disease diagnosis and for monitoring therapy. State-of-the-art studies make use of large datasets, which consist of time dependent sequences of 2D and 3D images from multiple imaging sessions. Systematic development and assessment of image analysis techniques requires an ability to efficiently invoke candidate image quantification methods on large collections of image data. A researcher may apply several different image analysis methods on image datasets containing thousands of 2D and 3D images to assess ability to predict outcome or effectiveness of a treatment across patient groups.

3 Related Work

Relatively little scheduling research so far has given importance to the issues of data locality and I/O contention. Ranganathan et. al. [13] proposed a decoupled approach to scheduling of computations and data for data-intensive applications in a grid environment, and evaluated its effectiveness via simulation studies. The algorithm combines a scheduling scheme, called *Job Data Present* with a replication heuristic, referred to as *Data Least Loaded* in a decoupled fashion. The algorithm incorporates a notion of eligible nodes for each job, which are the set of nodes that store the file required by the job. It works by picking a job from a FIFO queue and assigning it to the node that already has the required data. If more than one compute nodes are eligible candidates, then it chooses the least loaded node. The replication mechanism *Data Least Loaded* is decoupled from the scheduling policy. The replication mechanism keeps track of the popularity of files, and when the popularity of a file exceeds a threshold, then the file is replicated to the least loaded node in the compute cluster. As the replication threshold decreases, the number of dynamic data replications increases. This in turn increases the possibility of increased end-point contention on the storage cluster. Therefore, there is a tradeoff between the benefit of a low replication threshold and the increased contention. In our case, we allow multiple files per job which means that there may exist compute nodes which store subsets of the files required by a job. This essentially amounts to allocating a job to a node such that the expected data transfer time to stage in the set of files required by a job is minimized.

Casanova et al. [3] modified the MinMin, MaxMin, and Sufferage job scheduling heuristics to take into account the cost of inter-site file access, in the context of scheduling parameter sweep applications in a Grid environment. Jain et.al. [5] model scheduling of I/O operations (with certain assumptions) as a bipartite graph coloring problem with two separate sets of nodes namely, disks and processors. Our difference is that we consider grouping and mapping of jobs to

compute nodes in tandem with ordering of jobs and scheduling of remote I/O operations for file transfers. Mohamed et al. [12] presented a Close-To-Files (CF) job placement algorithm which tries to place jobs on clusters with enough idle processors that are close to the storage sites where the files reside.

Multi-query workloads also arise in the context of database applications. The work of Mehta et al. [11] is one of the first to address the problem of scheduling queries in a parallel database by considering batches of queries. In [1], Andrade et.al. propose a dynamic scheduling model for multi-query workloads in data analysis applications. The goal is to maximize data and computation reuse and concurrent execution on SMP nodes through semantic caching and ordering of queries based on priority metric. These strategies mainly target efficient reuse of results from previously executed queries.

Kotz et al. [8] propose a technique called disk-directed I/O to organize multiple overlapping I/O operations with a view to optimize disk performance which is the bottleneck. The work of Kavas et al. [6] focusses on loading of executables on the compute nodes and not just data. They propose reliable multicast mechanisms to load a file to multiple nodes at once thereby reducing the storage node overheads.

In an earlier work [7], we looked at the problem of scheduling a batch of data-intensive jobs with batch-shared I/O behavior. We modeled the sharing of files among jobs as a hypergraph and employed hypergraph partitioning to obtain a partitioning of jobs onto compute nodes that computationally balanced the workload and reduced remote I/O operations for file transfers. In this paper, we are targeting an online scenario where a set of file-shared data-intensive jobs arrive over time. To accomplish this, we have extended our previous work [7] in such a way so as to dynamically model the state of the system at each scheduling instant which includes the content of disk caches at the compute nodes, the remaining execution time of the running jobs, and the pending jobs that are present in the system. Our approach for the batch mode case involves a one time hypergraph modeling and partitioning which looks at the entire set of jobs that have arrived together as a batch and the initial system state which is cold, to yield a load-balanced connectivity minimizing allocation of jobs. Whereas for this work, we propose repeated partitioning and remapping of jobs at each scheduling instant by taking into account the current state of the system at each scheduling instant.

4 Dynamic Job Scheduling

We propose an Online Hypergraph partitioning based scheduling (Online-HPS) heuristic, a two stage dynamic scheduling framework. In the first stage, jobs are mapped to compute nodes, and in the second stage, the order of the jobs in each compute node are determined. These two stages are then applied in a repeated fashion at certain scheduling events which may correspond to job arrivals or job completions.

For mapping jobs to compute nodes we employ a hypergraph-based formulation, hence we start with a brief description of hypergraphs and hypergraph partitioning followed by our proposed mapping technique. We will continue with a description of job ordering stage.

4.1 Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyper-edges) $\mathcal{N}$ among those vertices. Every net $n_j \in \mathcal{N}$ is a subset of vertices, i.e., $n_j \subseteq \mathcal{V}$. The size of a net n_j is equal to the number of vertices it has, i.e., $s_j = |n_j|$. Weights (w_i) and costs (c_j) can be assigned to the vertices $(v_i \in \mathcal{V})$ and edges $(n_j \in \mathcal{N})$ of the hypergraph, respectively. $\mathcal{P} = \{V_1, V_2, \ldots, V_P\}$ is a *P-way partition* of $\mathcal{H}$ if 1) each part is a nonempty subset of $\mathcal{V}$, 2) parts are pairwise disjoint and 3) union of P parts is equal to $\mathcal{V}$.

In a partition $\mathcal{P}$ of $\mathcal{H}$, *connectivity* λ_j of a net n_j denotes the number of parts connected by n_j. A net n_j is said to be *cut* if it connects more than one part, i.e. $\lambda_j > 1$. The cost of a partition Π is computed as $\chi(\Pi) = \sum_{n_j \in \mathcal{N}_E} c_j(\lambda_j - 1)$, where $\mathcal{N}_E$ is the set of cut nets and each cut net n_j contributes $c_j(\lambda_j - 1)$ to the cutsize. This cost metric is also known as *connectivity-1* metric. The hypergraph partitioning problem can be defined as the job of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion among the part weights is maintained. Algorithms based on the *multi-level* paradigm, such as PaToH [4], have been shown to compute good partitions quickly for this NP-hard problem.

4.2 Runtime Hypergraph-based Mapping of the System State

We develop a hypergraph formulation to model the sharing of files among the jobs present in the system. At each scheduling event, a new hypergraph is constructed which models 1) the current state of the system that includes the pending jobs and the files requested by them, 2) the currently executing jobs, and 3) the files already cached on the compute nodes due to previously executed jobs. This is followed by K-way partitioning of the hypergraph to obtain a load-balanced cut minimizing mapping of the pending jobs onto the compute nodes. The currently executing jobs are incorporated in the partitioner to take into account the current value of load on each of the compute nodes and thereby facilitate load balance as a result of the new partitioning.

Our hypergraph model consists of two sets of vertices, one set of vertices represents the pending jobs which are present in the system and the other set represents jobs currently in execution on the compute nodes. A particular job j_i is represented by a vertex v_i in the hypergraph. Each hyper-edge n_j represents a file f_j and connects to two different set of vertices, one set is the set of vertices corresponding to pending jobs that require this file as input, and the other is the vertices corresponding to running jobs which are running on a node already having a cached a copy of file f_j. This hypergraph is partitioned into P groups,

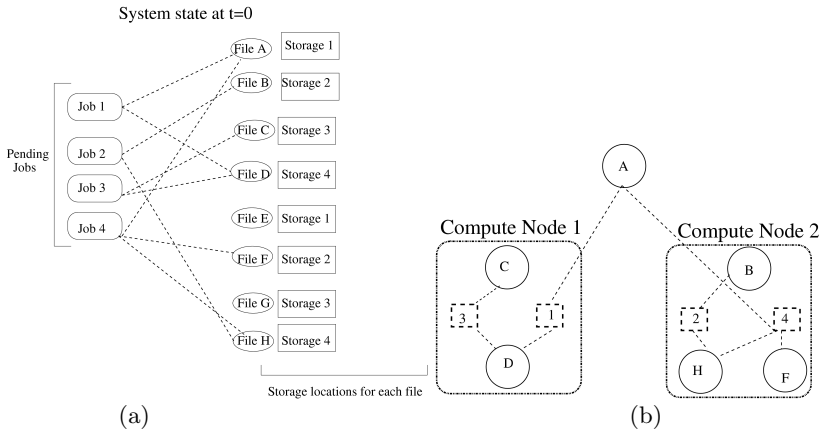


Fig. 2. a) A snapshot of the system at t=0. Jobs 1,2,3 and 4 have arrived into the system. Letters represent files and numbers represent the jobs. Lines connecting the jobs to files represent the associated file requests for each job. b) Hypergraph partitioning across two compute nodes at t=0.

where P is the number of compute nodes, and each group is mapped to a compute node. The partitioning is done so that the compute and I/O weight of the clusters are balanced and the cost of transferring shared files across clusters is minimized. The partitioning should ensure that the vertices corresponding to running jobs are allocated to the same compute node on which they are already running. This is made sure by pinning the vertices corresponding to running jobs onto the nodes in which they are running.

Figure 2(a) illustrates the state of the system at time t=0. It shows the arrival of 4 jobs into the system and their associated file requests. The boxes next to each file represent the storage locations for each file at t=0. Figure 2(b) illustrates a partitioning of the hypergraph representation of the system state shown in Figure 2(a). The figure shows that the hypergraph partitioning tries to cluster jobs sharing files together. Figure 3(a) illustrates the state of the system at time t=10. The figure shows two sets of vertices corresponding to pending jobs and running jobs respectively. Job 1 and Job 2 have run to completion and hence the corresponding vertices are not present. Replicas of files (i.e., multiple copies of files on the compute nodes) have been created as files had been staged onto the compute cluster for previous jobs. The solid lines show the file requests by running jobs which can be served locally whereas the dotted lines represent the file requests which may or may not be served locally based on the result of the subsequent partitioning.

Figure 3(b) illustrates a partitioning of the hypergraph representation of the system state shown in Figure 3(a). The solid boxes represent the running jobs which have been mapped to the same nodes as in Figure 2(b). This is accomplished by pinning down the running jobs onto the nodes on which they are already running. The dotted boxes represent the pending jobs which have been been mapped to one of the compute nodes. The partitioning in Figure 3(b)

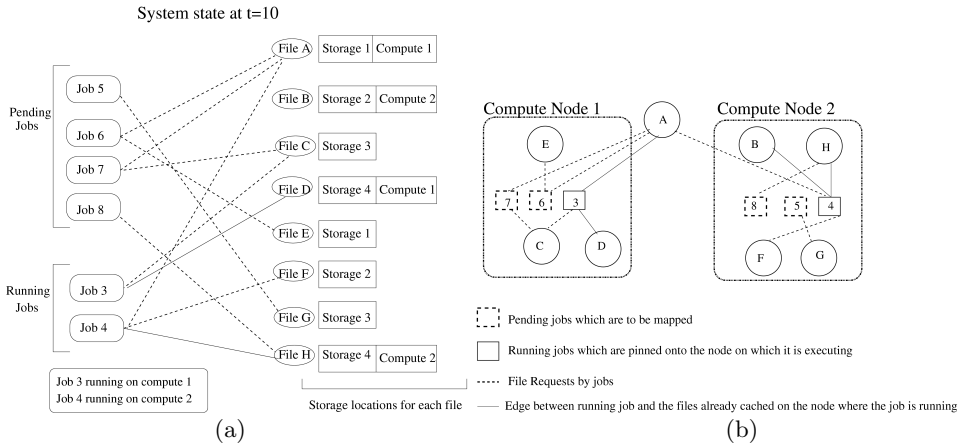


Fig. 3. a) A snapshot of the system at t=10. Jobs 5,6,7 and 8 have arrived into the system. Jobs 1 and 2 have finished execution. Jobs 3 and 4 are currently in execution on nodes 1 and 2 respectively. b) Hypergraph partitioning across two compute nodes at t=10.

shows that the jobs have been mapped to nodes with which they have strong affinity in terms of the files already cached on those nodes while maintaining load balance. The figure shows two sets of lines. The dotted lines represent the file requests associated with the jobs. The solid lines connect each running job to the files that are already cached on the node on which the job is running. These associations between a net representing a file already cached on a node with the vertex representing the job running on that node are done to exploit the file affinities of certain pending jobs to nodes which have copies of one or more files requested by these jobs. Any pending job which requests a lot of files already cached on a node will therefore have greater inter-job affinity with the running job on that node. Therefore, in essence, we have modeled both the inter-job file sharing affinities and the job-node affinity due to caching of files.

The weight of a vertex representing a pending job is equal to the estimated execution time of the corresponding job. The estimated execution time of a job is calculated as the sum of I/O overhead (the transfer time of files from storage nodes plus the I/O time to read files from local disk) and the computation cost of the job. The hypergraph based strategy globally partitions all the existing jobs into groups before any order for job execution is determined for a group. Hence it has to use a static vertex weights. The expected execution time of a job can possibly vary depending upon the node allocated to the job. This is because different nodes may have staged in different sets of files and therefore the job will have different locality of reference with each node. In other words, the execution times of jobs are not fixed but vary based on the allocation of the nodes and in time. In order to alleviate this issue and provide a better estimate of the execution time of a job, we compute the weight of a vertex as follows.

Let the set of files a job j_i needs be F_i. The cost of transferring one byte of file f_j, Tr_j, for job j_i is equal to

$$Tr(j_i) = \frac{Prob_{FNE}}{RBW} + (1 - Prob_{FNE}) \times \frac{(1 - Prob_{FE})}{RBW}. \quad (5)$$

Here, RBW is the I/O bandwidth between a storage node and a compute node, $Prob_{FNE}$ is the probability that job j_i will be the first job to execute in its group that requires f_j, and $Prob_{FE}$ is the probability that j_i executes on a node, to which file f_j has already been transferred. In our current implementation, we assume uniform probability distribution. Hence, we have used $Prob_{FNE} = \frac{1}{s_j}$ and $Prob_{FE} = \frac{1}{P}$. Recall that s_j is the size of the hyper-edge n_j that represents file f_j. Hence it also denotes the number of jobs that shares the file f_j.

We assume that the computation time of a job is linear with the size of the input files it requires. This is a reasonable assumption since we assume that multiple instances of only a single application are being run on the system and there is no interference effect due to multiple different applications. With this assumption, the estimated execution time of job j_i is computed as

$$EstimatedExec(j_i) = \sum_{f_j \in F_i} filesize(f_j) \times (Tr_j + \frac{1}{LBW} + C) \quad (6)$$

where LBW is the I/O bandwidth from local disk on a compute node and C is the compute cost of one byte. By assigning the files sizes as hyper-edge costs, the proposed method reduces the job mapping problem to the P-way hypergraph partitioning problem according to the *connectivity-1* cutsize definition [4]. Each and every file needed by the jobs in the job trace will be transfered to the compute system at least once. More specifically, if the jobs that share the file f_j is assigned to λ_j compute nodes, file f_j needs to transfered $\lambda_j - 1$ more times after its first transfer.

The weight of a vertex representing an already running job is equal to the remaining estimated execution time of the corresponding job. This is computed in a similar fashion as explained above except that it models the fact that some of the files required by a running job may already have been staged and therefore would not contribute to its remaining execution time.

By using expected execution times as vertex weights, the algorithm aims to balance computational load across the compute nodes. The expected execution time as calculated in equation 6 is based on a probabilistic model for estimating the cost of file transfer which assumes a uniform distribution. In scenarios where the data-staging costs are high and much more significant as compared to the computational costs, the impact of making such an assumption could affect load balance but the overall system performance would depend more on the connectivity metric. Therefore, the impact of the inaccuracy of this assumption would be lesser in such scenarios.

4.3 Job Ordering in a Compute node and Scheduling of Remote file transfers

Once the jobs have been mapped to a node, the local scheduling algorithm within each compute node decides the order in which to schedule the queued jobs and their associated file transfers. When a node becomes idle, the local scheduling algorithm running at the node decides the next job to execute on that node and also decides the schedule for its remote file transfers. Two jobs that are in different compute nodes may have their input files stored on the same set of nodes. Thus, ordering of jobs in each compute node and transfer of files should be done in a way to minimize end-point contention on the storage cluster.

We employ a strategy in which jobs within a group are scheduled based on their earliest completion time. Therefore, when a node becomes idle, the algorithm computes the completion time of each of the queued jobs present on that node and schedules the job with the earliest completion time. The earliest completion time of a job is computed iteratively and dynamically based on the availability of resources.

The algorithm maintains a *Gantt chart* for storage nodes. When a job in a group is scheduled for execution, time slots are reserved on storage nodes for file transfers required for this job. These time slots for a job are marked on the Gantt chart. In calculating the duration of time slots and marking them on the Gantt chart, we assume that multiple requests to the same storage node are serialized and that a compute node can receive a file after it has finished storing the previously received file on local disk.

The earliest completion time of a job j_i is estimated as the sum of time to stage its input files F_i and its execution time. The staging time is the time spent to make the input files ready in the compute node. If all of the input files are already in the compute node, the staging time will be zero. Otherwise, it will be the amount of time spent to transfer the last input file from the storage node. The transfer completion time for each file $f_j \in F_i$ (TCT_j) is estimated as the sum of the earliest time a transfer can start (first available slot in the Gantt chart after the time that the compute node becomes available) and the actual transfer time (size of f_j divided by the storage bandwidth; computed as the minimum of remote disk bandwidth and network bandwidth). The file f_j with the minimum TCT_j is picked and tentatively scheduled for transfer. TCTs of the rest of the input files are recomputed and the next file with the minimum TCT is picked and tentatively scheduled. This process is repeated until all of the input files are scheduled. TCT of the last file scheduled actually gives the staging time. Then the earliest estimated completion time for j_i is computed as the sum of 1) the completion time of file transfers from storage nodes, 2) I/O time to read the files on local disk, and 3) CPU time to process the files. The scheduling algorithm determines the job with the least completion time in each group, and the job j_i with the lowest *earliest completion time* out of these is scheduled first. Once j_i is scheduled, out of the other job groups (excluding the one containing j_i), the job with the minimum earliest completion time (taking into account the current

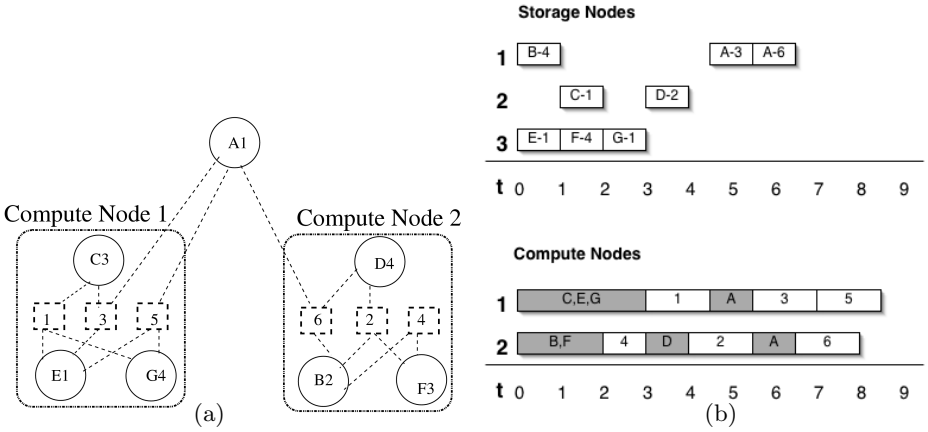


Fig. 4. a) Hypergraph representation of a queue of jobs at a certain point in execution. The numbers next to the alphabets representing the files are the storage node ids on which the corresponding files are resident. b) An illustration of the execution of the ordering algorithm on the set of queued jobs.

reservations) is now picked and scheduled. When a running job completes, the job with earliest completion time from that group is scheduled.

Let us consider a hypergraph partitioning of a stream of six jobs which were submitted to a system of two compute nodes and 4 storage nodes. Figure 4(a) illustrates the corresponding hypergraph partitioning. Figure 4(b) illustrates the execution of the ordering algorithm on the set of mapped jobs shown in Figure 4(a) . In this figure transfer of each file takes 1 unit of time, and I/O and processing of a file takes 0.3 and 0.2 units of time, respectively. Since job 4 depends on two files, its earliest completion time is 3. Hence it has been scheduled first and 1 unit of time on storage node 1 and 1 unit of time on storage node 3 have been reserved. Since a job has been scheduled from group 2, next the job with the earliest completion time from group 1 is scheduled. Since all of the job in the group depends on 3 files, and they can be scheduled to transfer all of the files in 3 units, we pick one of them, say job 1. The algorithm continues by reserving the transfer of files for job 1, and another job from group 2 is picked.

4.4 File Eviction Policy

If the transfer of file for a particular job violates the disk space constraint on the compute cluster, a disk file eviction mechanism is invoked which deletes files in the increasing order of their value metric. The value of a file $Value_\ell$, is calculated as follows.

$$Value_\ell = \frac{AccessFreq(f_\ell) \times filesize(f_\ell)}{Numcopies(f_\ell)} \tag{7}$$

$AccessFreq(f_\ell)$ represents the number of accesses to the file so far and is representative of its frequency of access. $filesize(f_\ell)$ represents the size of the

file f_ℓ. $Numcopies(f_\ell)$ represents the number of copies of file f_ℓ in the compute cluster. If two files have the same frequency of access up to the current time in execution, and the same size, the file with fewer copies gets a higher popularity value as evicting that file is more likely to result in a remote file transfer when the file is again needed. The intuition behind including the file size in popularity computation is that greater the size of the file is, greater the cost of getting the file back to a node will be. The algorithm evicts smaller files, since the cost of staging such files again in future will be less.

We have integrated this file eviction mechanism into our proposed approach as well as MCT, MET, and SA approaches for the purpose of performance comparison. For the algorithm *Job Data Present* with *Data Least Loaded*, we employ an LRU based eviction mechanism as described in [13].

5 Existing Job Mapping Techniques

In this paper, we examine the *JobDataPresent + DataLeastLoaded* algorithm proposed in [13] in the context of data grids and the *Minimum Execution Time* (MET), *Minimum Completion Time* (MCT), *Switching Algorithm* (SA) heuristics, which were originally proposed for scheduling independent computational jobs to compute resources [10]. As in [2, 3, 7], we modify MET, MCT and SA to take into account 1) the time it takes to transfer input and output files to and from compute nodes in the environment, 2) files that have already been staged to a compute node in estimating the minimum completion time of a job and 3) in case of MCT and SA, also the files that are being staged to a compute node due to currently running job on that node. We also integrate the Gantt chart based explicit scheduling of remote file transfers as explained in Section 4.3 into the MET, MCT and SA algorithms.

JobDataPresent + DataLeastLoaded : The algorithm combines a scheduling scheme, called *Job Data Present* with a file replication heuristic, referred to as *Data Least Loaded* in a decoupled fashion. The details of the algorithm have been explained in Section 3.

Minimum Execution Time (MET): The MET heuristic assigns each job to a node that results in the least execution time ($Exec_i$) for that job. As a job arrives, all the compute nodes in the cluster are examined to determine the node that gives the best execution time for the job. When computing the expected execution time of a job on a node, MET takes into account the files already available on the node. If none of the files required by a job are found in any compute node, then the first available node is chosen to run the job. In other words, if the minimum execution time of a job an each node of the cluster is the same, then the first available node is chosen to execute the job. Therefore, MET heuristic inherently favors data locality since nodes which cache files required by a particular job are the ones which will get its best execution time.

Minimum Completion Time (MCT): The MCT heuristic assigns each job to a node that results in that job's earliest completion time ($Completion_i$). As a job arrives, all the compute nodes in the cluster are examined to determine the node that gives the earliest completion time for the job. When computing the expected completion time of a job on a node, MCT takes into account the files already available on the node and files which be available on the compute node in future due to staging of data caused by the currently executing job on the node, as well as the completion time of the currently assigned jobs to that node. Hence, MCT may discard data locality and assign a new job to node which does not have any of its files cached because the wait times on the nodes with which the job have very good file locality may be high.

Switching algorithm (SA): The MET heuristic has a potential drawback in that it can lead to load imbalance across nodes by assigning many more jobs to some node than to others since it blindly looks at data locality without considering possible load imbalance. The MCT heuristic assigns jobs to nodes to achieve earliest completion time thereby ensuring load balance but does not necessarily exploit data locality since it may not allocate a job to a node which already has its files cached due to excess waiting times on that node. SA heuristic is motivated by the fact that it is possible to use MET at the expense of load imbalance until a given threshold and then use MCT to smooth the load across the cluster. Similar to [10], let ib be the *load balance index* defined as $ib = load_{\min}/load_{\max}$ where $load_{\min}$ and $load_{\max}$ are the loads (completion time of the last job on that node) of minimum and maximum loaded nodes. We define two thresholds l and h. SA starts mapping jobs with MCT heuristic until the load balance index reaches to h, after that point it switches to MET and continues until load balance index decreases below l at that point it switches to MCT again and this cycle continues. In our experiments we have used $l = 0.3$ and $h = 0.7$. The goal of SA is to have a heuristic with the desirable properties of load balance as well as data locality optimization.

6 Experimental Results

We now present an experimental evaluation of the proposed strategies along with the MET, MCT, SA and JobDataPresent-DataLeastLoaded (JDPDLL) strategies. For evaluation, we used an application class: biomedical image analysis. We compared the performance of the various scheduling schemes under a varying set of scenarios covering multiple job-file sharing patterns and different distributions of job inter-arrival times.

6.1 Application Workloads

For the image analysis application, we implemented a program to emulate studies that involve analysis on images obtained from MRI and CT scans (captured on multiple days as follow-up studies). An image dataset consists of a series of 2D

images obtained for a patient and is associated with meta-data describing patient and study related information (in our case, we used patient id and study id as the meta-data). Each image in a dataset is associated with an imaging modality and the date of image acquisition and stored in a separate file. An image analysis program can select a subset of images based on a set of patient ids and study ids, image modality, and a date range.

We evaluated the scheduling schemes using job traces where several aspects were varied: 1) job inter-arrival rate (to vary system load), 2) extent of file sharing among jobs, 3) temporal clustering characteristics of file-sharing behavior between jobs, and 4) burstiness of job arrivals.

We generated workloads with different degrees of file sharing among jobs: *high sharing*, *medium sharing*, and *low sharing*. The different degrees of sharing is achieved by varying the values of patient and time attributes across requests by different jobs. We generated workloads with 85%, 40%, and 10% overlap, on average, in terms of files requested by different jobs in the job trace for high, medium, and low overlap cases.

The dataset generated by the emulator corresponded to a dataset of 2000 patients and images acquired over several days from MRI and CT scans. Each job on an average accessed 6 files. The number of files accessed by a job varied from 4 to 10. The sizes of images were 4 MB and 64 MB for MRI and CT scans, respectively. The overall size of the dataset was around 2 Terabytes. Images for each patient were distributed among all the storage nodes in a round robin fashion.

The image analysis application typically involve computations equivalent of two floating point operations per word. We, therefore, emulated it with 2 FP operations per word and measured that this translates to a processing time of approximately 0.001s/MB of data in our test-bed[4].

6.2 Modeling the Load

In traditional compute-intensive job scheduling, the offered load on the system is calculated as:

$$OfferedLoad = \frac{\sum_{\forall i} Exec(j_i) \times n(j_i)}{P \times \max_{\forall i}(Arrival(j_i))} \tag{8}$$

where $n(j_i)$ represents the number of nodes allocated to a job j_i, P is the number of nodes in the system. In compute-intensive job scheduling, the $OfferedLoad$ metric is entirely dependent on the job trace under consideration and is independent of the scheduling policy being employed. However, in the data-intensive scheduling scenario we are focusing on, the metric defined in

[4] It can be expected that when computation time dominates the overall execution time, the traditional job scheduling strategies would work well. The CPU power and memory bandwidth are increasing very rapidly and faster than the bandwidth of I/O devices. With such a trend, the I/O cost will become more pronounced thus entailing the need to develop scheduling algorithms which target data intensive applications.

Equation 8 is no longer dependent only on the job trace but is also a function of the scheduling policy. This is because in the data-intensive scenario, the job execution times are not fixed. Instead, they vary with time due to staging of files by previously run jobs and also vary based on the node allocated to the job because of varying degrees of locality. Therefore, the job execution times depend upon the scheduling policy. To address this issue, we propose the following new characterization of load which is dependent only on the characteristics of the job trace and is independent of the scheduling policy.

Let ArrivalRate be the job arrival rate in Jobs/sec. Let ServiceRate be the expected Job service rate in Jobs/sec. The expected load is defined as follows.

$$Load = \frac{ArrivalRate}{ServiceRate} \tag{9}$$

Let us consider a trace of N jobs, where each job has an associated set of file transfers. Let the set of files needed by job j_i be F_i.

Let $AvgExectime$ denote the average of the execution times over all the jobs.

$$AvgExecTime = \frac{1}{N} \times \sum_{\forall i} EstimatedExec(j_i) \tag{10}$$

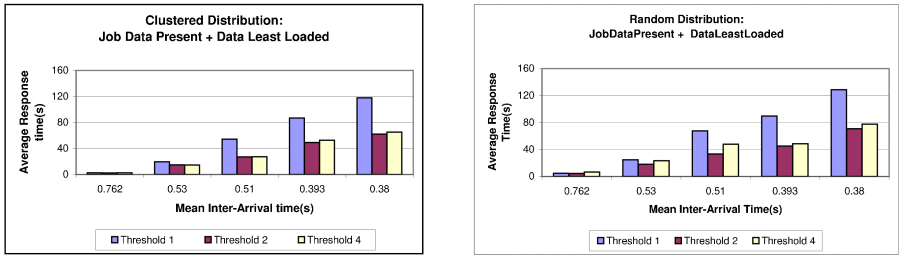
The $EstimatedExec$ time is same as calculated based on the probabilistic model explained in Section 4.2. To achieve an overall load of 1, The time of arrival of the last arriving job $TLarrival$ in the system is calculated as follows.

$$TLarrival = AvgExectime \times \frac{N}{P} \tag{11}$$

To summarize, we first determine the arrival time of the last job by using the information about the files accessed by each job so as to achieve a load value of 1. We then generate job traces with different values of load by varying the number of jobs which arrive over a fixed period of time. The modeling of load is based on estimated execution times which are based on a probabilistic model as shown in equation 6. In reality, some jobs will require a lower actual execution time than their expected execution time if some needed files are locally available since they were staged by previously executed jobs. On the other hand, the execution time may be higher in reality, due to contention at the storage server node for file transfer.

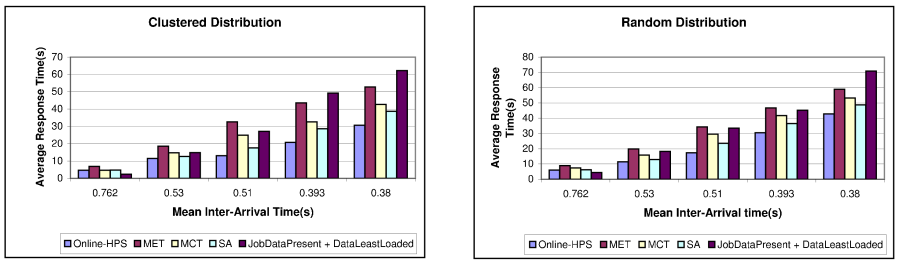
6.3 Modeling the Arrival Process

We model the arrival process as a Poisson random process and evaluate it with two distributions corresponding to different job orderings - clustered distribution and random distribution. Clustered distribution refers to the case where jobs sharing files among themselves occur closer together in time. Random distribution refers to the case where jobs come in any random order. Here, the arrival times of file-sharing jobs may be widely separated from each other over



(a) (b)

Fig. 5. Performance of *Job Data Present* coupled with *Data Least Loaded* under various replication thresholds



(a) (b)

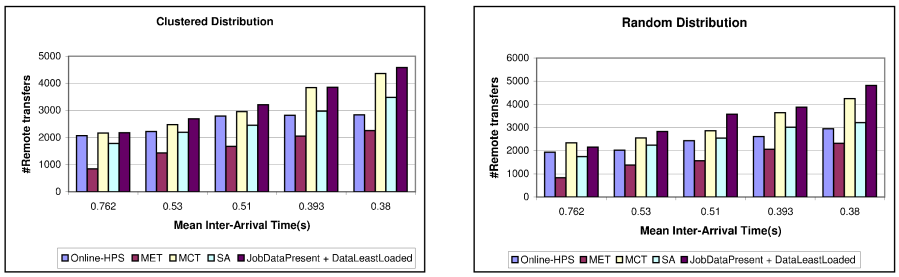
Fig. 6. Average Response time achieved by different algorithms for the (a) Clustered Distribution and (b) Random Distribution

time. We also model the arrival times using the model proposed by Lublin [9]. The Lublin model is based on analysis of different production logs and uses statistical methods in order to achieve a good match of synthetic traces and actual trace data. The job arrival model takes into account both the stationary arrival process during peak hours and also the daily cycle. Since the model is based on long-running jobs from production supercomputer installations, we scaled down the arrival times to reduce the overall time to run our experiments.

6.4 Performance Evaluation on a Cluster

We conducted our experiments using a memory/storage cluster at the Department of Biomedical Informatics at the Ohio State University. The cluster consists of 64 nodes with an aggregate 0.5 TBytes of physical memory and 48TB of disk storage. These nodes are connected to each other through Infiniband.

One of the comparison schemes - JDPDLL - uses a critical "threshold" parameter to decide when a file should be replicated at another node. We first ran JDPDLL with different values of the replication threshold parameter. Figure 5 shows the variation in performance. The replication threshold represents the minimum number of references to a file by a compute node needed to trigger a replication of that file to a least-loaded node. Three different threshold values were used: 1, 2 and 4. Figure 5 show that the choice of the threshold has



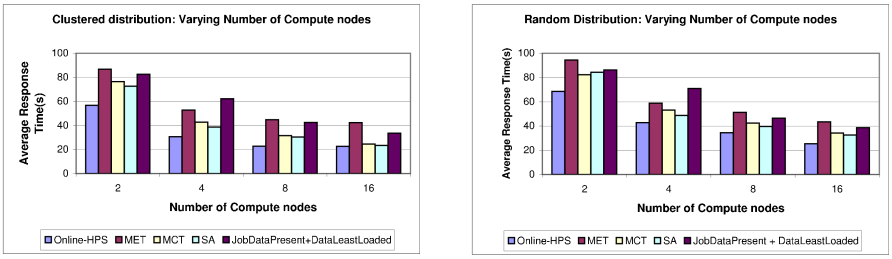
(a) (b)

Fig. 7. Number of remote file transfers in different algorithms for the (a) Clustered Distribution and (b) Random Distribution

a significant effect on the performance of this algorithm - there is a trade off between benefits of increased replication and the storage node end-point contention caused by an increasing number of dynamic data replications. In our experiments, we noted that a threshold value of 2 gave the best results and therefore this threshold is used for comparing the performance of this scheme against others.

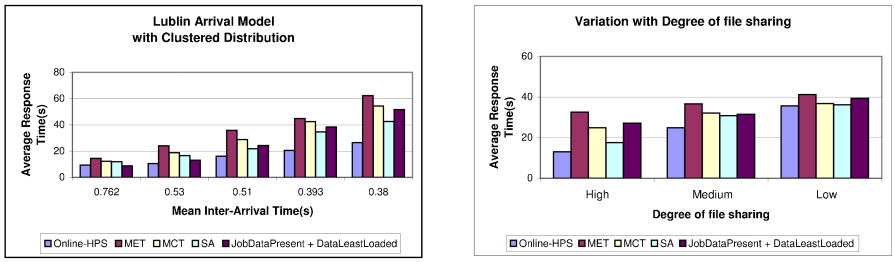
Figure 6 shows the relative performance of the various scheduling schemes in terms of the average response time. These experiments were conducted using 4 compute nodes and 4 storage nodes. The number of jobs in the traces used for this experiment varied from 800 to 1600 and the time of arrival of the last job in each trace was around 600 secs. The value of load based on our characterization as explained in Section 6.2 varied from being around 1 for the 800 job trace to around 2 for the 1600 job trace. Each compute node used for this experiment had an available space of 15GB. The figures show that hypergraph-partitioning scheme (Online-HPS) performs better than the other schemes in most of the cases. This is because it models the inter-job affinity due to file-sharing and clusters jobs that share files transfers transfer of the same file multiple times. The benefit of the proposed scheme is higher as the inter-arrival times decrease since the partitioning scheme has information about more jobs at its disposal and it exploits this information to make more informed global decisions. The base schemes MCT, MET, SA, and JDPDLL consider one job at a time when making local greedy job mapping decisions and therefore do not take into account the implicit inter-job affinities due to file sharing.

At very low loads, JDPDLL performs the best since the average inter-arrival times are high and there are significant idle periods during which file replication occurs without interfering with other file transfers. of storage node end-point of both the job play a job-inter arrival time decreases, the performance of JDPDLL deteriorates compared to Online-HPS because the file replication activity causes contention with I/O from jobs reading input files from the storage nodes. The effect of end-point contention becomes more and more significant as the system load increases.



(a) (b)

Fig. 8. (a) Average Response time achieved by the various algorithms with varying number of compute nodes for the (a) Clustered Distribution and (b) Random Distribution



(a) (b)

Fig. 9. (a) Performance of the various algorithms under the Lublin arrival model and (b) Performance of the different algorithms with variation in the degree of file sharing across jobs

Figure 7 shows the number of remote file transfers for all the algorithms for the same set of experiments as shown in Figure 6. As might be expected, Online-HPS causes fewer remote transfers compared to MCT, SA and JDPDLL. This is because it attempts to cluster together jobs that share files, thereby reducing the need for multiple transfers of the same file. The MET heuristic results in the least number of remote file transfers over all the schemes. This is because it maps each job to a node with which the job has maximum affinity in terms of the files already cached on it and required by the job. However, while doing so, it does not model the queue wait times at each node, thereby causing severe load imbalance across the nodes. Therefore, it gives the worst average response time in spite of being the best in terms of minimizing the remote file transfers.

To analyze the scalability of the proposed scheme with respect to the number of compute nodes, we ran experiments with the high overlap workload consisting of 1600 jobs. The number of compute nodes were varied from 2 to 16. These experiments were run using 4 storage nodes. Figure 8 shows the results with varying number of compute nodes. As is seen from the figure, Online-HPS achieves the best performance in terms of average response time in all the cases.

Figure 9(a) shows the relative performance of the various scheduling schemes in terms of the average response time by employing the Lublin arrival model to

generate the job inter-arrival times. The results show that Online-HPS consistently performs well compared to the other schemes. The relative performance improvement under the Lublin model is higher compared to the traces modeling a Poisson arrival process. With the bursty nature of job arrival with the Lublin arrival process, the partitioning heuristic makes better job allocation decisions during bursts where a large number of queued jobs are available and inter-job file affinities can be exploited.

Figure 9(b) shows the relative performance of the various scheduling schemes on job traces with different degrees of shared I/O among jobs. These experiments were conducted using 4 compute nodes and 4 storage nodes. The high overlap job had 1200 jobs with an average inter-arrival time of 0.51. The medium and low overlap workloads had 800 and 400 jobs, respectively. These workloads were generated to have a uniform value of expected load. However, in reality, the medium and low overlap workloads took a longer time to execute since endpoint contention became more significant as the degree of file sharing decreased (due to increase in the number of remote file transfers). The results in Figure 9(b) show that the benefit of the Online-HPS scheme is greatest for the high overlap workload and reduces as the degree of overlap decreases.

7 Conclusions

This paper proposes a novel hypergraph based dynamic scheduling heuristic for a stream of dynamically arriving independent I/O intensive jobs. The approach is based on a run-time hypergraph based modeling of the system state, followed by locality-aware and load balanced mapping and scheduling of jobs onto the compute nodes. The performance results obtained on a coupled compute/storage cluster show that it achieves significant performance improvement over previously proposed heuristics - *MET*, *MCT*, *SA* and *JobDataPresent* with *Data Least Loaded* - when there is a high degree of file sharing among jobs. The previous schemes do not explicitly consider inter-job dependences arising out of file-sharing and thus make local decisions based on greedy heuristics. The choice of the best scheduling algorithm for a particular scenario depends upon parameters such as inter-arrival times and inter-job file sharing. Under very lightly loaded conditions, when the average job inter-arrival time is high, data replication proves to be more beneficial if a good choice of replication threshold is made. As inter-arrival times decrease, the proposed approach, which takes an integrated view of scheduling of computation and data placement, outperforms the other heuristics.

References

1. H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, FL, April 2002.

2. H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE SC00 Conference*, pages 75–76, 2000.
3. H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, 2000.
4. U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.
5. R. Jain, K. Somalwar, J. Werth, and J. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, Mar 1997.
6. A. Kavas, D. Er-El, and D. G. Feitelson. Using multicast to pre-load jobs on the parpar cluster. *Parallel Computing*, 27(3):315–327, 2001.
7. G. Khanna, N. Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, and P. Sadayappan. A hypergraph partitioning based approach for scheduling of tasks with batch-shared I/O. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, May 2005.
8. D. Kotz. Disk-directed i/o for mimd multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.
9. U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.
10. M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, Apr. 1999.
11. M. Mehta, V. Soloviev, and D. J. DeWitt. Batch scheduling in parallel database systems. In *Proceedings of the 9th International Conference on Data Engineering (ICDE 1993)*, Vienna, Austria, 1993.
12. H. Mohamed and D. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *2004 IEEE International Conference on Cluster Computing*, pages 287–298. IEEE Society Press, 2004.
13. K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing (HPDC)*, Edinburgh, Scotland, July 2002.
14. D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proceedings of High-Performance Distributed Computing (HPDC-12)*, pages 152–161, Seattle, Washington, June 2003.