Gang Scheduling Extensions for I/O Intensive Workloads

Yanyong Zhang[†], Antony Yang[†], Anand Sivasubramaniam[‡], Jose Moreira[§]

[†] Department of Electrical & Computer Engg.
Rutgers, The State University of New Jersey
Piscataway NJ 08854
{yyzhang,pheroth}@ece.rutgers.edu

[‡] Department of Computer Science & Engg.
The Pennsylvania State University
University Park PA 16802
{ anand}@cse.psu.edu

[§] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{jmoreira}@us.ibm.com

Abstract

Scientific applications are becoming more complex and more I/O demanding than ever. For such applications, the system with dedicated I/O nodes does not provide enough scalability. Rather, a serverless approach is a viable alternative. However, with the serverless approach, a job's execution time is decided by whether it is co-located with the file blocks it needs. Gang scheduling (GS), which is widely used in supercomputing centers to schedule parallel jobs, is completely not aware of the application's spatial preferences.

In this paper, we show that gang scheduling does not do a good job scheduling I/O intensive applications. We extend gang scheduling by adding different levels of I/O awareness, and propose three schemes. We show that all these three new schemes are better than gang scheduling for I/O intensive jobs. One of them, with the help of migration, outperforms the others significantly for all the workloads we look at.

1 Introduction

Scientific applications, involving the processing, analyzing, and visualizing of scientific and engineering datasets, are of great importance to the advancement of society. Large-scale parallel machines are used in supercomputing centers to meet their needs. Scheduling of jobs onto the nodes of these machines thus becomes an important and challenging area of research. A large number of jobs are to run on the system. These jobs are usually long running, requiring significant amount of CPU and storage resources. A scheduling scheme that is not carefully designed can easily result in unbearably high response time and low throughput. The research is challenging because of the numerous tradeoffs from the application characteristics as well as the underlying system configurations involved in designing a scheduler. Some of the influencing factors are job arrival rates, job sizes, job behaviors, system configuration parameters, and system capacity. Now, this research is facing new challenges because the applications are increasing dramatically in their sizes, lengths, and resource requirements, at a pace which has never been seen before.

Scheduling strategies can have a significant impact on the performance characteristics of a large parallel system [4, 5, 8, 10, 13, 14, 19, 20, 23]. Early strategies used a space-sharing approach, wherein jobs can run side by side on different nodes of the machine at the same time, but each node is exclusively assigned to a job. The wait and response times for jobs with an exclusively space-sharing strategy can be relatively high. Gang scheduling (or coscheduling) is a technique to improve the performance by adding a time-sharing dimension to space sharing [18]. This technique virtualizes the physical machine by slicing the time axis into multiple virtual machines. Tasks of a parallel job are coscheduled to run in the same time-slices. The number of virtual machines created (equal to the number of time slices), is called the multiprogramming level (MPL) of the system. This approach opens more opportunities for the execution of parallel jobs, and is thus quite effective in reducing the wait time, at the expense of increasing the apparent job execution time. Gang scheduling can increase the system utilization as well. A considerable body of previous work has been done in investigating different variations of gang scheduling [25, 24, 26, 22]. Gang-scheduling is now widely used in supercomputing centers. For example, it has been used in the prototype GangLL job scheduling system developed by IBM Research for the ASCI Blue-Pacific machine at LLNL (a large scale parallel system spanning thousands of nodes [16]).

On the application's end, over the past decade, the input data sizes that these applications are facing have increased dramatically, at a much faster pace than the performance improvement of the I/O system [15]. Supercomputing centers usually deploy dedicated I/O nodes to store the data and manage the data access. Data is striped across high-performance disk arrays. The I/O bandwidth provided in such systems is limited by the number of I/O channels and the number of disks, which are normally smaller than the number of available computing units. This is not enough to serve the emerging scientific applications that are more complex than ever, and are more I/O demanding than ever. Fortunately, several research groups have proposed a serverless approach to address this problem [1, 2]. In a serverless file system, each node in the system serves as both compute node and I/O node. All the data files are distributed across the disks of the nodes in the system. Suppose a task (from a parallel job) wants to access a data block. If the block is hosted on the disk of the node on which this task is running, then the data can be fetched from the local disk, incurring a local disk I/O cost. If the block resides on the disk of another node, then an extra network cost is needed, in addition to the cost of accessing the disk, to fetch the data. With this approach, we have as many I/O nodes as the compute nodes. It is much easier to scale than the dedicated solution. The only assumption this approach has is reasonably fast interconnect between the nodes, which should not be a concern given the rapid progress in technology.

While the serverless approach shows a lot of promise, realizing the promise is not a trivial task. It raises new challenges to the design of a job scheduler. Files now are partitioned across a set of nodes in the system, and, on the same set of nodes, jobs are running as well. Suppose task t needs to access block b of file f. The associated access cost can vary considerably depending on the location of b. If b is hosted on the same node where t is running, then t only needs to go to the local disk to fetch the data. Otherwise, t needs to pay extra network cost to fetch the data from a remote disk. The disk accesses now become asymmetric. In order to avoid this extra network cost, a scheduler must co-locate tasks of parallel jobs with the file blocks they need. This will be even more urgent for I/O intensive jobs.

Although gang scheduling, which is widely used in many supercomputing centers, is effective for traditional workloads, most of its implementations are completely unaware of applications' spatial preferences. Instead, they focus on maximizing system usage when they allocate the jobs into the system so that a new job can be accommodated into the system quickly, and a running job can receive more CPU time as well. However, it is not yet clear which of the two factors - I/O awareness, or system utilization - is more important to the performance. Or, both are important for some scenario. If so, what is better, at what scenario, by how much? And, can we do even better by adaptively balancing between the two? This paper sets out to answer all the above questions by conducting numerous simulation based experiments on a wide range of workloads. This paper proposes several new scheduling heuristics, which try to schedule jobs to their preferred nodes, without compromising the system utilization. This paper also extensively evaluates their performances.

In this paper, we have made the following contributions:

- We propose a new variation of gang scheduling, I/O aware gang scheduling (IOGS), which is aware of the jobs' spatial preferences, and always schedules jobs to their desired nodes.

- We quantitatively compare IOGS and pure gang scheduling under both I/O intensive workloads and medium I/O workloads. We find that pure gang scheduling is better than IOGS for medium workloads, while IOGS is much better than gang scheduling for I/O intensive workloads, which are the motivating workloads for this study.

- We propose a hybrid scheme, adaptive I/O aware gang scheduling (Adaptive-IOGS), which tries to combine the benefits of both schemes. We show that this adaptive scheme performs the best in most of the situations except for I/O intensive workloads at very high loads.

- Further, we propose another scheme, migration adaptive I/O aware gang scheduling (Migrate-IOGS), which combines migration technique with the existing scheme. Migration technique can move more jobs to their desired nodes, thus leading to better performance.

- Finally, we have shown that Migrate-IOGS delivers the best performance among the four across a wide range of workloads.

The rest of the paper is organized as follows: Section 2 describes the system and workload models used in this study. Section 3 presents all the proposed scheduling heuristics, and report their performance results. Section 4 presents our conclusions and possible direction for future work.

2 System and Workload Model

Large scale parallel applications are demanding large data inputs during their executions. Optimizing their I/O performance is becoming a critical issue. In this study, we set out to investigate the impact of application I/O intensity on the design of a job scheduler.

2.1 I/O Model

One of the latest trends in file system design is serverless file system [1, 2], where there is no centralized server, and the data is distributed on the disks of the client nodes that participate in the file system. Each node in the system serves as both compute node and I/O node. In a system that has dedicated I/O nodes (such as storage area network), I/O bandwidth is limited by the number of I/O channels and number of disks, which is not suitable for the more complex and more I/O demanding applications. However, in a serverless approach, we have as many I/O nodes as the nodes in the system, which provides much better scalability. Earlier work has shown that the serverless approach is a viable alternative, provided the interconnect is reasonably fast. (The details about serverless file systems are beyond the scope of this paper, and interested readers can read the references.) In this study, we adopt this serverless file system approach.

The data files in the system are partitioned according to some heuristics, and distributed on the disks of a subset of the nodes (clients that participate the file system). The file partitioning heuristic is a very important issue, and will be described in detail in next section.

We use t_{IO}^{local} and t_{IO}^{remote} to denote the local and remote I/O costs respectively. In this paper, when we use the term *I/O costs*, we mean the costs associated with those I/O requests that are not satisfied in the file cache, and have to fetch data from disk.

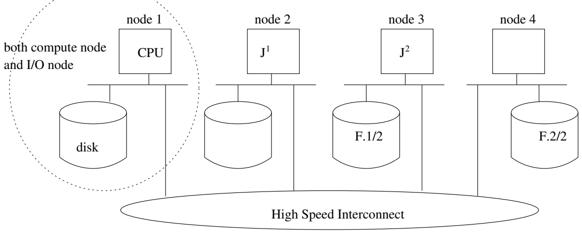


Figure 1: the I/O model example

Figure 1 illustrates such a system with 4 nodes, which are connected by high-speed interconnect. File F has two partitions, $F.1/2$ and $F.2/2$, hosted by nodes 3 and 4 respectively. Parallel job J (with tasks J^1 and J^2) is running on nodes 2 and 3, and needs data from F during its execution. In this example, if the data needed by J^2 belongs to partition $F.1/2$, then J^2 spends much less time in I/O because all its requests can be satisfied by local disks, while J^1 has to fetch the data from remote disks.

2.2 File Partition Model

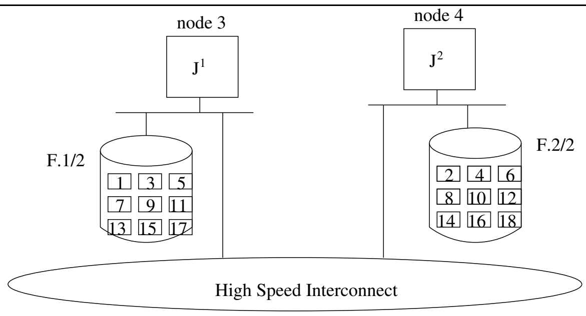


Figure 2: File partition Example

File partition plays an important role in deciding a task's I/O portion. Even though a task is running on a node that hosts one partition of its file, it does not mean that the task can enjoy lower local I/O costs because the data it needs may not belong to this partition. In the example shown in Figure 2, F has 18 blocks in total, and all the odd-numbered blocks belong to $F.1/2$, while the even-numbered blocks belong to $F.2/2$. Task J^1 is co-located with all the odd-numbered blocks of F. Unfortunately, J^1 needs the first 10 blocks of file F, which are evenly distributed between F.1/2 and F.2/2. Thus, half of its I/O requests have to go to remote disks. This example shows that even if a job scheduler manages to assign tasks to the nodes where their files are hosted, their I/O requests may not be fully served by local disks. This observation suggests that we need to coordinate the file partitioning and applications' data access pattern to better appreciate the results of good job schedulers.

As suggested by Corbett and Feitelson in [3], it is possible for the applications to pass their access patterns to the underlying file system, so that the system can partition the files accordingly, leading to a one-to-one mapping between the tasks and partitions. The authors even suggested that file partitioning can be dynamically changed if the access pattern changes. Thus, as long as the task is assigned to the appropriate node (hosting the corresponding partition), all its I/O accesses are local. In this study, we used the above approach. Although the hardware configurations in [3] and in this study are different, we believe that we can implement the same idea on our platform. Again, please note that our schemes can work with other partitioning heuristics as well. In that case, we need to compose the job access pattern model and quantify the ratio of the local I/Os out of the total number of I/Os made by each task.

In this study, we do not dynamically re-partition files in order to avoid the associated overheads. In fact, as statistics in [17] show, file sharing across applications is rare, which implies that dynamic re-partitioning is not necessary.

2.3 Workload Model

We conduct a simulation based study of our scheduling strategies using synthetic workloads. The basic synthetic workloads are generated from stochastic models that fit actual workloads at the ASCI Blue-Pacific system in Lawrence Livermore National Laboratory (a 320-node RS/6000 SP). We first obtain one set of parameters that characterizes a specific workload. We then vary this set of parameters to generate a set of nine different workloads, which impose an increasing load on the system. This approach, described in more detail in [12], allows us to do a sensitivity analysis of the impact of the scheduling algorithms over a wide range of operating points.

From the basic synthetic workload, we obtain the following set of parameters for job i: (1) t_i^a: arrival time, (2) t_i^C: CPU time (if i is run in a dedicated setting, the total time that is spent on CPU), (3) n_i: number of nodes.

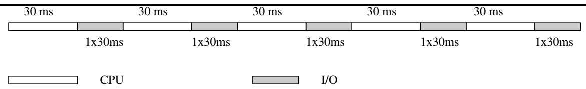


Figure 3: The execution model for task i with $d_i^I = 0.03$second, n_i^B=1, and $t_{IO}^{local} = 0.03$second.

Next, we add one more dimension to the jobs: I/O intensity. In this study, we employ a simple model assuming I/O accesses are evenly distributed throughout the execution. We use two more parameters to support this simple model for job i: (4) d_i^I: the I/O requests interarrival time, and (5) n_i^B: number of disk blocks per request. In the example shown in Figure 3, I/O operations are evenly distributed throughout the execution. There is an I/O operation every 0.03 second (30 milliseconds). It accesses one block during each I/O operation. Since we assume every I/O access is local, and the local cost is 30 milliseconds, then every I/O operation takes 1×30 milliseconds. Please note that we do not care whether these I/O requests are for one file, or multiple files. As discussed in Section 2.2, all the files a job accesses are partitioned according to its access patterns. By varying the values of d_i^I and n_i^B, we can vary the I/O intensity of job i, with smaller d_i^I and greater n_i^B implying higher I/O intensity. In addition to these application parameters, system parameters also play a role in a job's I/O intensity, such as local and remote I/O costs (t_{IO}^{local} and t_{IO}^{remote}), with higher I/O costs leading to higher I/O intensity. The I/O intensity of job i is thus calculated as $\frac{n_i^B \times t_{IO}^{local}}{n_i^B \times t_{IO}^{local} + d_i^I}$, where we assume that all the I/O requests are local. For example, if job i issues I/O requests every 0.03 second ($d_i^I = 0.03 second$), accessing one block per request ($n_i^B = 1$), and a local I/O request takes 0.03 second to finish, then i's I/O intensity is 50%.

Based on the definition of the model, we can further derive (6) $n_i^I = \lceil \frac{t_i^C}{d_i^I} \rceil \times n_i^B$: number of disk blocks accessed, and (7) $t_i^e = t_i^C + \sum_{j=0}^{j<n_i^I}(l_{i,j} * t_{IO}^{loca} + (1 - l_{i,j}) * t_{IO}^{remote})$: i's execution time on a dedicated setting, where $l_{i,j}$ is 1 when the jth I/O of job i is local, and 0 otherwise. Next we need to elaborate on deciding the value of $l_{i,j}$, i.e., how to decide if an I/O is a local or remote. In our job model, we do not differentiate tasks of the same job. We assume they are identical. Thus, even though only one task of a job is not running on the file node , and the other tasks are all on the file nodes, the job, as a whole, is considered to have remote I/O accesses because the tasks from the same parallel job need to synchronize with each other and progress at the same pace. From the definition, we find that a job's execution time is dependent on its location, which is decided by the scheduling strategy. In order to have a common " execution time " across different scheduling strategies, we calculate another parameter (8)$t_i^{e\prime}$: local execution time (the execution time if all the I/O requests are local). It is calculated as $t_i^{e\prime} = t_i^C + \sum_{j=0}^{j<n_i^I} t_{IO}^{local}$. $t_i^{e\prime}$ is used later when we calculate job slow down.

In addition to the parameters from the workload, during simulation, we also monitor (9) t_i^s: start time and (10) t_i^f: finish time for job i.

From the above parameters, we can calculate the following: (11) $t_i^r = t_i^f - t_i^s$: response time, (12) $t_i^w = t_i^s - t_i^a$: wait time, and (13) $s_i = \frac{max(t_i^r, T)}{max(t_i^{e\prime}, T)}$: the slowdown for job i, where T is the time-slice for gang-scheduling. To reduce the statistical impact of very short jobs, it is common practice [6, 7] to adopt a minimum execution time. We adopt a minimum of one time slice. That is the reason for the $max(., T)$ terms in the definition of slowdown. Another thing to notice is that we use $t_i^{e\prime}$ instead of t_i^e in the definition because the former is independent of scheduling schemes.

We have looked at the parameters for a job. Next, let us look at the parameters for a workload which contains 10,000 jobs. A workload has parameters of two dimensions: workload type and workload load. Workload type, in this paper, means if a workload is I/O intensive or not. We are only interested in workloads with medium to high I/O intensities. A workload's I/O intensity is the average I/O intensity of the jobs included in this workload. A workload's load is described by the job arrival rate and the average job execution time.

To compare different scheduling strategies, we look at average job slowdown, average job response time and average job wait time. Job slowdown measures how much slower than a dedicated machine the system appears to the users, which is relevant to both interactive and batch jobs. Job response time measures how long a job takes to finish execution, which is also relevant to both interactive and batch jobs. Job wait time measures how long a job takes to start execution and therefore it is an important measure for interactive jobs.

3 Scheduling Strategies

3.1 Gang-Scheduling

Gang scheduling [4, 5, 9, 11, 21] is a timed-shared parallel job scheduling technique. This technique virtualizes the physical machine by slicing the time axis into multiple virtual machines. Tasks of a parallel job are coscheduled to run in the same time-slices (same The number of virtual machines created (equal to the number of time slices), is called the multiprogramming level (MPL) of the system. This multiprogramming level in general depends on how many jobs can be executed concurrently, but is typically limited by system resources. This approach opens more opportunities for the execution of parallel jobs, and is thus quite effective in reducing the wait time. Gang-scheduling has been used in the prototype GangLL job scheduling system developed by IBM Research for the ASCI Blue-Pacific machine at LLNL (a large scale parallel system spanning thousands of nodes [16]).

As an example, gang-scheduling an 8-processor system with a multiprogramming level of four is shown in 4. The figure shows the Ousterhout matrix that defines the tasks executing on each processor and each time-slice. J_i^j represents the j-th task of job J_i. The matrix is cyclic in that time-slice 3 is followed by time-slice 0. One cycle through all the rows of the matrix defines a *scheduling cycle.* Each row of the matrix defines an 8-processor virtual machine, which runs at $1/4$th of the speed of the physical machine. We use these four virtual machines to run two 8-way parallel jobs (J_1 and J_2) and several smaller jobs (J_3, J_4, J_5, J_6). All tasks of a parallel job are always coscheduled to run concurrently. This approach gives each job the impression that it is still running on a dedicated, albeit slower, machine. This type of scheduling is commonly called *gang-scheduling* [4]. Note that some jobs can appear in multiple rows (such as jobs J_4 and J_5).

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
time-slice 0	J_1^0	J_1^1	J_1^2	J_1^3	J_1^4	J_1^5	J_1^6	J_1^7
time-slice 1	J_2^0	J_2^1	J_2^2	J_2^3	J_2^4	J_2^5	J_2^6	J_2^7
time-slice 2	J_3^0	J_3^1	J_3^2	J_3^3	J_4^0	J_4^1	J_5^0	J_5^1
time-slice 3	J_6^0	J_6^1	J_6^2	J_6^3	J_4^0	J_4^1	J_5^0	J_5^1

Figure 4: The scheduling matrix defines spatial and time allocation.

Every job arrival or departure constitutes a *scheduling event* in the system. For each scheduling event, a new scheduling matrix is computed for the system. Even though we analyze various scheduling strategies in this paper, they all follow an overall organization for computing that matrix, which can be divided into the following steps:

1. **CleanMatrix:** The first phase of a scheduler removes every instance of a job in the Ousterhout matrix that is not at its assigned home row. Removing duplicates across rows effectively opens the opportunity of selecting other waiting jobs for execution.

2. **CompactMatrix:** This phase moves jobs from less populated rows to more populated rows. It further increases the availability of free slots within a single row to maximize the chances of scheduling a large job.

3. **Schedule:** This phase attempts to schedule new jobs. We traverse the queue of waiting jobs as dictated by the given priority policy until no further jobs can be fitted into the scheduling matrix.

4. **FillMatrix:** This phase tries to fill existing holes in the matrix by replicating jobs from their home rows into a set of replicated rows. This operation is essentially the opposite of **CleanMatrix**.

In the rest of this paper, we describe each scheduling strategy based on their implementations of these scheduling steps.

3.2 Plain Gang Scheduling (GS)

The simulation model is based on the implementation of the GangLL scheduler [16] on the Blue Pacific machine at Lawrence Livermore National Labs. For GS, we implement the four scheduling steps of Section 3.1 as follows:

1. **CleanMatrix:** The implementation of CleanMatrix is best illustrated with the following algorithm:

   ```
   for i = first row to last row
       for all jobs in row i
           if row i is not home of job, remove it
   ```

   It eliminates all occurrences of a job in the scheduling matrix other than the one in its home row. A job is assigned a home row when it is first scheduled into the system, and its home row may change when the matrix is re-calculated.

2. **CompactMatrix:** We implement the CompactMatrix step in GS according to the following algorithm:

   ```
   do{
      for i = least populated row to most populated row
          for j = most populated row to least populated row
              for all jobs in row i
                  if they can be moved to row j, then move and break
   }while matrix changes
   ```

   We traverse the scheduling matrix from the least populated row to the most populated row. We attempt to find new homes for the jobs in each row. The goal is to pack as many jobs in as few rows as possible.

3. **Schedule:** The Schedule phase for GS traverses the waiting queue in FCFS order. For each job, it looks for the row with the least number of free slots in the scheduling matrix that has enough free columns to hold the job. This corresponds to a best fit algorithm. This algorithm opens more opportunities for the larger jobs to be accommodated into the matrix. The row to which the job is assigned becomes its home row. Within that row, we look for the columns that are free across multiple rows, increasing the likelihood of this job being replicated to other rows, which helps with the matrix utilization. We stop when the next job in the queue cannot be scheduled right away.

4. **FillMatrix:** We use the following algorithm in executing the FillMatrix phase.

   ```
   do {
      for each job in starting time order
          for all rows in matrix,
              if job can be replicated in same columns do it and break
   } while matrix changes
   ```

3.3 IO-Aware Gang Scheduling (IOGS)

GS tries to reduce job wait times by letting them accommodated as soon as possible, and to keep the matrix efficiently utilized. This goal is attempted by its allocation heuristics, including searching for the target row with BestFit algorithm, and then searching for the target columns for the possibility of future replication. These optimizations would translate into performance improvement if the jobs do not have any co-location preferences. Nonetheless, this is not true for the motivating applications in this study. The job execution times are significantly affected by their locations. Suppose job i needs file f during its execution, and partitions of f reside on nodes $n_{i_1}, n_{i_2}, ..., n_{i_{n_i}}$. These nodes are called *file nodes* of i. i's execution time will be shortened if it is allocated to its file nodes. If an allocation scheme is not aware of this, and thus causes longer execution times, it may hurt the overall performance. To address this issue, we propose a new scheme, called I/O aware gang scheduling (IOGS). IOGS schedules every job to its file nodes, with the cost of longer wait times in the queue.

At every scheduling event, IOGS also re-calculates the matrix using the following four steps:

1. **CleanMatrix:** Same as in GS.

2. **CompactMatrix:** Same as in GS.

3. **Schedule:** The Schedule phase for IOGS traverses the waiting queue in FCFS order. For each job, it looks for the row with the least number of free slots where the job's file nodes are available. Similar to the Schedule phase for GS, this also corresponds to a best fit algorithm. However, here, the criteria of "fit" is having all file nodes available (to the interests of applications), instead of having enough free columns as in GS (to the interests of matrix usage). Within that row, we choose the file nodes to schedule the job. Please note that the number of file nodes equals the corresponding job size (Section 2.2). We stop when the next job in the queue cannot find a row that has its files nodes available.

4. **FillMatrix:** Same as in GS.

3.4 Comparing IOGS and GS

Looking at the heuristics of the two schemes, we know that GS tries to maximize the matrix usage, while IOGS provides local I/O accesses. Before we report our results, let us first take a closer look at what happens to a job so that we can understand the pros and cons for both schemes better for the motivating workloads.

Once a job starts executing, how soon it could finish is determined by its execution time (job duration on a dedicated setting) as well as how much CPU time it receives. A job's execution consists of two components: computation and I/O (Section 2.3), and the latter is affected by whether the data can be accessed locally or not. IOGS yields short execution times by scheduling jobs to their files nodes. On the other hand, GS allocation scheme opens more opportunities for the jobs to be replicated to multiple rows, thus receiving more CPU time. Moreover, in IOGS, a job is scheduled into the matrix only if all its file nodes are free during at least one time quantum, leading to longer job wait time in the queue.

Intuitively, the relative importance of these two factors should be related to the I/O intensity of the workload. Next we conduct experiments to confirm this hypothesis. Table 1 summarizes the relevant parameters, and their

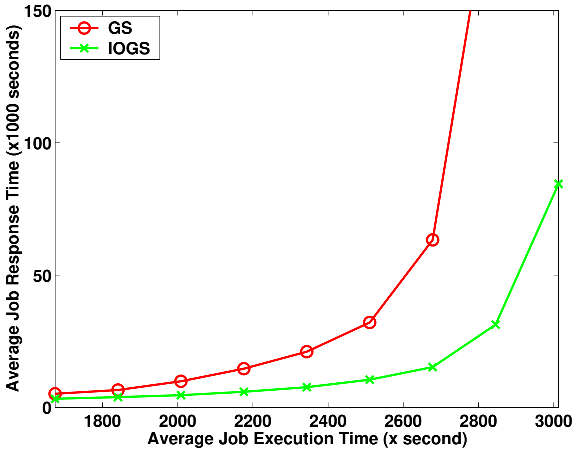
parameter	value(s)
n_i^B for any job i	1
d_i^I for any job i	0.03, 0.06 second
t_{IO}^{local}	0.03 second/block
t_{IO}^{remote}	0.09 second/block
high I/O intensity	50%
medium I/O intensity	33%
T	200 seconds

Table 1: The relevant paramenters and their values in the results shown in this paper.

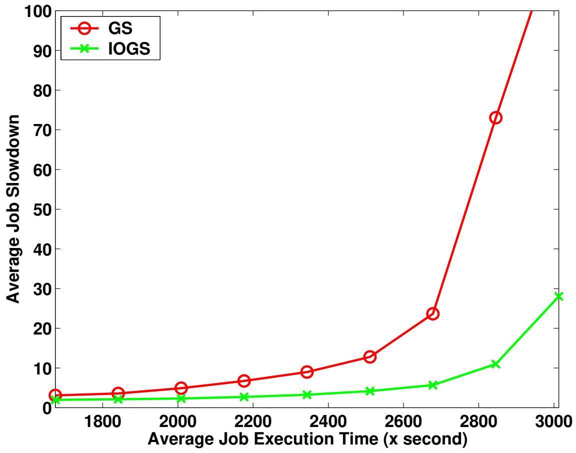
values used in the results shown here. As discussed in Section 2.3, a workloads I/O intensity is decided by the average I/O requests interarrival times. Since our interests in this study are in workloads with medium to high I/O intensities, we adopt 0.06 and 0.03 for the average I/O requests interarrival times to model these two types of workloads (corresponding to I/O intensity of 33% and 50% respectively). Now, we have two classes of workloads:

high I/O intensity and medium I/O intensity. Each class has nine workloads, each with the same job arrival rate but increasing average job execution time. The average job arrival time is 534 seconds. Within each set, we have nine workloads corresponding to increasing loads (increasing average CPU times) to the system.

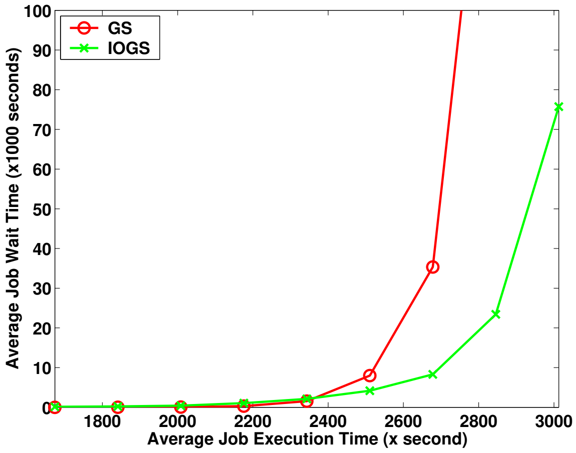
The results of the two scheduling strategies for these two classes of workloads are shown in Figures 5 and 6 respectively.



(a)



(b)



(c)

Figure 5: Comparison of GS and IOGS for I/O intensive workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time.

For high I/O intensity workload (Figure 5), IOGS is clearly better than GS for all nine workloads with different loads. for all three performance metrics (response times, slow down and wait times). With respect to response time, IOGS is always 100% better. Under most of the loads (i.e.,$t^{e\prime} \geq 2000$ seconds), the improvement is over 200%. With respect to job slow down, we see similar trends as response time. With respect to wait time, the improvement of IOGS over GS is not as obvious under low loads because the wait time is low for both schemes, and we observe significant improvement under very high loads (i.e.,$t^{e\prime} \geq 2600$ seconds). When the workload is I/O intensive (more than 50% of I/O ratio), the drawbacks of longer execution times due to remote I/O accesses far outweigh the benefits of more CPU time.

For medium workload (Figure 6), we observe the opposite trend. GS is always better. The reason is, for medium to low I/O intensity workloads, because of the less importance of I/O, the benefits of shorter execution times are overshadowed by the drawbacks of less CPU resources received.

From this set of experiments, we learn that neither IOGS nor GS is the best for all the workloads. For I/O

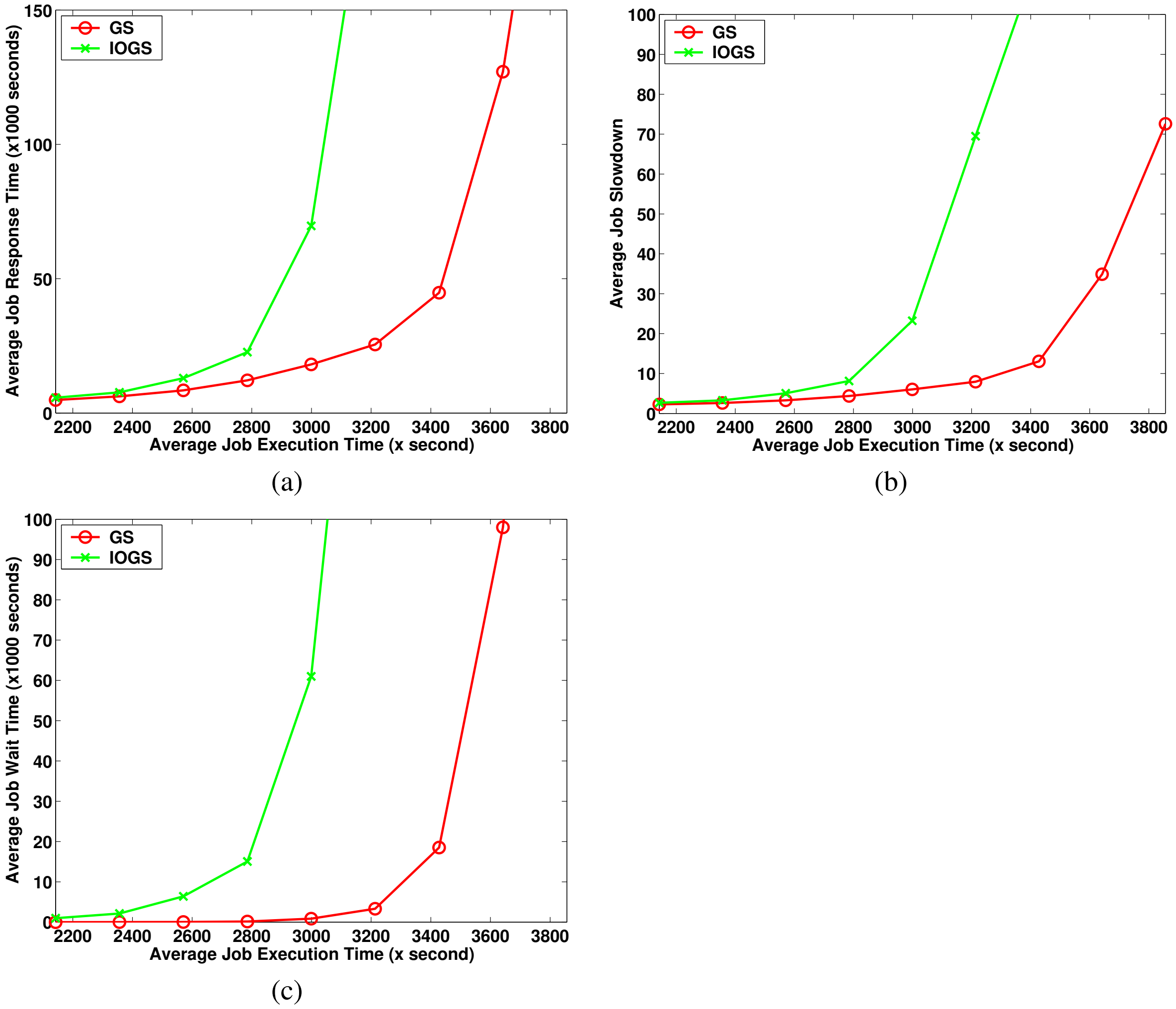
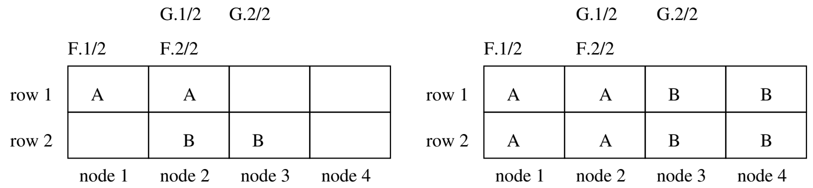


Figure 6: Comparison of GS and IOGS for medium workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time.

intensive workloads, IOGS is better, while GS is a much better choice for low to medium I/O workloads.

3.5 Adaptive I/O Aware Gang Scheduling (Adaptive-IOGS)

From the previous section, we learn that IOGS delivers better performance for I/O intensive workloads that motivated this work, as it yields much shorter execution times by scheduling jobs to their file nodes. However, the downside of doing so is that it causes fragmentation in the scheduling matrix. Figure 7 illustrates this problem. In this example,



(a) IOGS (b) GS

Figure 7: An example: job A needs file F, and job B needs file G. F is distributed on nodes 1 and 2, and G on 2 and 3.

node 2 hosts partitions of both files (F and G), which means both jobs will be assigned to node 2. As a result, A and B will be allocated to different rows (Figure 7(a)), and neither can be replicated to the second row because they share a common node. The matrix utilization in this case is only 50%. Moreover, this allocation heuristic leaves two small holes in both rows. If job C, with 3 tasks, arrives, it has to be kept waiting. On the other hand, GS can utilize the matrix 100%, as shown in Figure 7(b). Further, IOGS makes a job wait if it cannot find a row where all its file nodes are free, even though there are rows that have enough free slots to hold the job, which makes it more difficult to accommodate a job, thus leading to another type of matrix fragmentation: a job cannot start execution even though there are free slots because of strict allocation heuristics. To make matters even worse, because this job is the first in the waiting queue, all the jobs that arrive later than it have to wait as well, until its file nodes are freed after the jobs that occupy them finish.

Next, we propose a new strategy that alleviates the fragmentation caused by IOGS, while keeping its benefit for I/O intensive workloads. We call it Adaptive-IOGS. Adaptive-IOGS tries to first allocate a job as IOGS does. If no row has all the files nodes free, then we allocate this job as GS does, looking for rows/columns so that the matrix usage is maximized, instead of making it wait. If IOGS and GS are the two extremes, then this approach stands in the middle, using an adaptive allocation heuristic.

A formal description of its four scheduling steps is as follows:

1. **CleanMatrix:** Same as in IOGS and GS.
2. **CompactMatrix:** Same as in IOGS and GS.
3. **Schedule:** The Schedule phase for Adaptive-IOGS traverses the waiting queue in FCFS order. For each job, it first looks for the row with the least number of free slots where the job's file nodes are available. If such a row is found, then we schedule the job to its file nodes within that row. Otherwise, we go back to GS: we look for the row with the least number of free slots that has enough free slots to hold the job. Within that row, we look for the columns/nodes that are empty across the most number of rows to open opportunities of replicating the job to other rows.
4. **FillMatrix:** Adaptive-IOGS puts every running job into one of the two queues: local job (jobs that are assigned to their file nodes) or remote job (jobs that are not assigned to their file nodes). The FillMatrix phase for Adaptive-IOGS examines local jobs first to see whether they can be replicated to other rows. After no local job can be replicated, it examines all the remote jobs. Within each queue, the jobs are examined in the FCFS order.

We conduct experiments to compare Adaptive-IOGS with both IOGS and GS, using the same configurations described in Section 3.4. We compare these three strategies under both I/O intensive workloads and medium workloads. The results for these two workloads are shown in Figures 8 and 9 respectively.

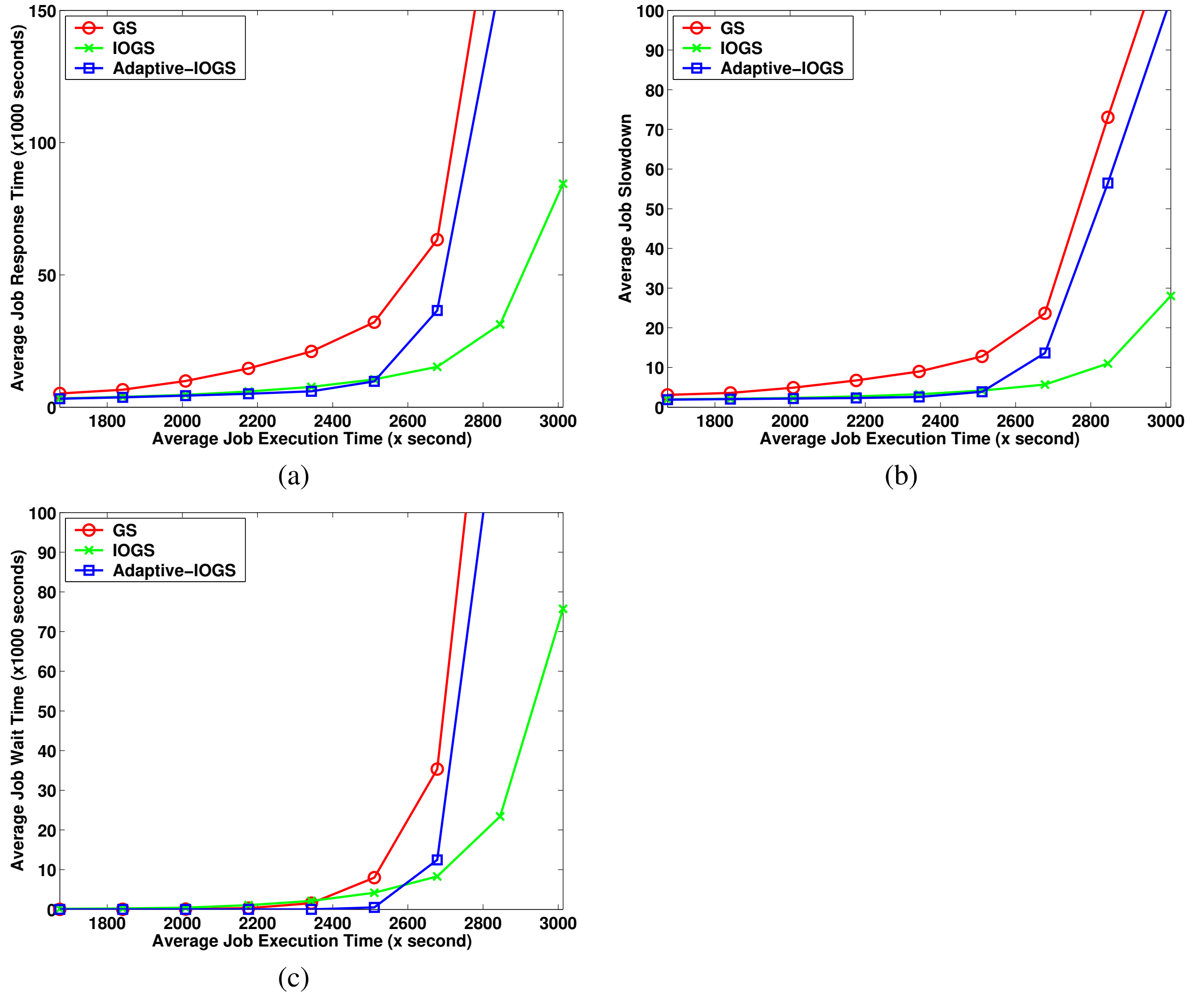


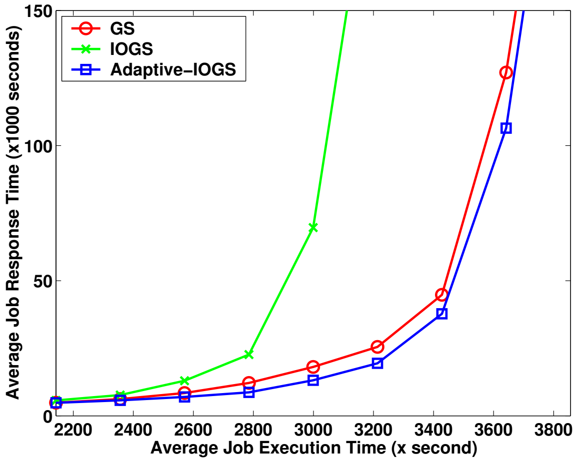
Figure 8: Comparison of GS, IOGS and Adaptive-IOGS for I/O intensive workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time.

We observe that Adaptive-IOGS is the best among the all for (i) medium I/O workloads (for all nine different loads), and (ii) I/O intensive workloads for low to medium loads (i.e., $t^{e\prime} \leq 2500$ seconds).

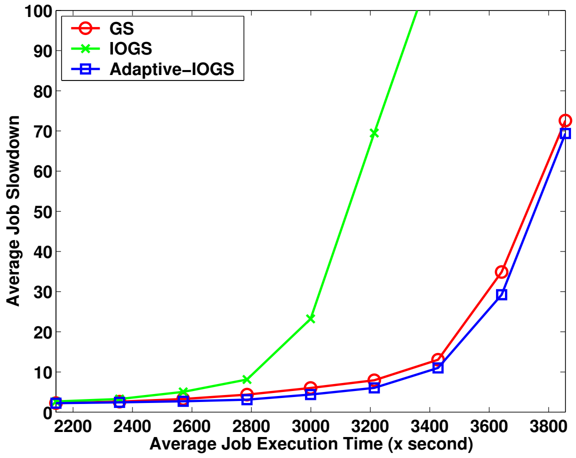
For I/O intensive workloads, under low to medium loads, Adaptive-IOGS enjoys both benefits of local I/O accesses and high matrix utilization. For instance, under load $t^{e\prime} = 1800$ seconds, Adaptive-IOGS manages to assign 9101 local jobs out of 10000, while GS only has 20 local jobs. (IOGS has 10000 local jobs out of 10000.) Please note that local I/O accesses are the key for I/O intensive workloads, and we find that Adaptive-IOGS can do almost as well as IOGS, with the additional benefit of less fragmentation. On the other hand, when the loads become high, very few jobs can be assigned to their files nodes when they are first scheduled. Adaptive-IOGS thus approaches GS, and its performance starts to degrade dramatically. Quantitatively, for the highest load $t^{e\prime} = 3000$ seconds, Adaptive-IOGS only has 831 local jobs, which is comparable to GS with 5 local jobs.

For medium I/O workloads, the key is efficient matrix usage, which is achieved by Adaptive-IOGS by scheduling the jobs of which the files nodes are not free to the rows/columns that minimize matrix fragmentation. In addition, Adaptive-IOGS can reduce the execution times of the jobs that are assigned to the local nodes. As a result, it has the best performance throughout the entire load range for medium I/O workloads.

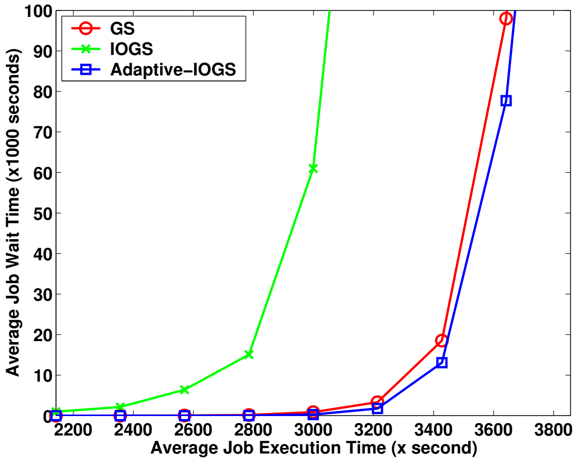
Next, we show how we further improve Adaptive-IOGS under the high loads for the I/O intensive workloads.



(a)



(b)



(c)

Figure 9: Comparison of GS, IOGS and Adaptive-IOGS for medium workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time.

3.6 Migration Adaptive I/O Aware Gang Schedule (Migrate-IOGS)

From the earlier results, we conclude that the challenge here is *how to achieve local I/O requests and low matrix fragmentation at the same time.* These two goals seem contradicting with each other. IOGS and GS works on one of the two goals respectively. Adaptive-IOGS tries to balance the two factors by adaptively change the goal based on the context. Although Adaptive-IOGS outperforms both GS and IOGS in most of the scenarios, it is not as good as IOGS under high loads for I/O intensive workloads. However, high loads of I/O intensive workloads are the most interesting scenarios to this study. We need to further investigate on how to improve its performance.

Adaptive-IOGS is approaching GS under high loads. In these scenarios, the matrix is already full, so few jobs can find their file nodes available when they are first scheduled. Thus, most of the jobs are scheduled into the system using GS Schedule heuristic. These jobs stay on the remote nodes once they are allocated there, paying higher remote I/O costs throughout the execution, even though their file nodes might have been freed immediately after they are scheduled.

	F.1/2	F.2/2		
row 1	A	A	C	C
row 2	D	B	D	
	node 1	node 2	node 3	node 4

Figure 10: An example illustrating Adaptive-IOGS: job D needs file F, which is distributed on nodes 1 and 2.

In the example shown by Figure 10, When D was first scheduled, its file nodes (nodes 1 and 2) were occupied by jobs A in row 1 and B in row 2, so it was allocated to row 2, on nodes 1 and 3. After a short period of time, job A finishes execution and leaves the system, D's file nodes are now free in row 1, but D still stays on nodes 1 and 3, and pays remote I/O costs, which can also be considered as one type of fragmentation. In order to address this type of fragmentation, we propose to use migration technique. In the above example, if we can migrate job D from row 2 to row 1, to its file nodes, D's remaining execution time will be significantly shortened.

The next scheduling strategy we propose uses migration technique to enhance Adaptive-IOGS, and we call the new scheme Migration Adaptive I/O Aware Gang Scheduling (Migrate-IOGS). Every time when a job leaves the system, we will examine each remote job to see if its file nodes are freed by then. If so, we move the job to its file nodes. We put this migration phase after we clean the matrix, and before we compact the scheduling matrix.

The formal description of the scheme is as follows:

1. **CleanMatrix:** Same as in Adaptive-IOGS, IOGS and GS.

2. **MigrateMatrix:** MigrateMatrix is a new phase added to Migrate-IOGS, where remote jobs can be migrated to their file nodes if those nodes are freed due to this new scheduling event.

   As explained in [24, 26], there is a cost associated with migration. In systems like IBM RS/6000 SP , migration involves a checkpoint/restart operation. Let C denote the total migration cost, including checkpointing and restarting a process. After a task is migrated to another node, it can resume its execution only after C. Further, even though only one task of a job is migrated, we make the other tasks also wait for C to model the synchronization between the tasks. Although all tasks can perform a checkpoint in parallel, resulting in a C that is independent of job size, there is a limit to the capacity and bandwidth that the file system can accept. Therefore we introduce a parameter Q that controls the maximum number of tasks that can be migrated in any time-slice.

   In this phase, for each remote job, Migrate-IOGS looks for the row that has all its file nodes available. If such a row is found, then Migrate-IOGS moves the job there so that its remaining execution will be significantly shortened. The job needs to pay the migration overhead only once (at the beginning of the next time quantum), and the value of C is negligible compared to the savings on the job's execution time, so we do not expect to observe any negative impact of migration. On the other hand, if there are more than one such rows (in which the file nodes are free), we have two options: (1) randomly choose one row, or (2) we can choose the row with the least number of free slots so that we can accommodate larger waiting jobs. In our study, we simply chose the first option due to the time limit. We will conduct further investigations for the final version.

We implement the MigrateMatrix step in Migrate-IOGS according to the following algorithm:

```
do{
   for each running job that is remote
     for all rows in matrix,
       if the job can become local in that row, move it and break
}while matrix changes
```

3. **CompactMatrix:** Same as in Adaptive-IOGS, IOGS and GS.

4. **Schedule:** Same as in Adaptive-IOGS.

5. **FillMatrix:** Same as in Adaptive-IOGS.

3.7 Comparing Migrate-IOGS, Adaptive-IOGS, IOGS, and GS

Finally, we can compare all four schemes under the same configurations. As in Sections 3.4 and 3.5, we use two workloads: I/O intensive workload (I/O requests interarrival time 0.03 second), and medium I/O workload (I/O requests interarrival time 0.06 second). In the experiments, we choose migration cost as 10% of the time quantum ($10\% \times 200 = 20$ seconds), and we assume any number of tasks can be migrated during a quantum.

The results for I/O intensive workloads are shown in Figure 11, and medium I/O workloads in Figure 12. For both workloads, Migrate-IOGS outperforms the other three scheduling strategies. With respect to I/O intensive workloads, under high loads (i.e., $t^{e\prime} \geq 2500$ seconds), Migrate-IOGS improves all three performance metrics by 100% compared to the second best scheme, IOGS. With the help of migration, Migrate-IOGS has 6524 local jobs at load $t^{e\prime} = 3000$ seconds, which is a significant improvement compared to its counter part without migration (Adaptive-IOGS) which only has 831 local jobs. With respect to medium I/O workloads, it improves the performance by at least 60% for all three metrics compared to the second best scheme - Adaptive-IOGS.

Also, we would like to point out that although we have no limit on the number of tasks that can be migrated during each time quantum, we did not observe any single occasion where the number exceeds 64, which is a reasonably small number compared to the simulated system with 320 nodes.

With the introduction of Migrate-IOGS, we find the scheduling scheme that works the best for all the workloads we look at.

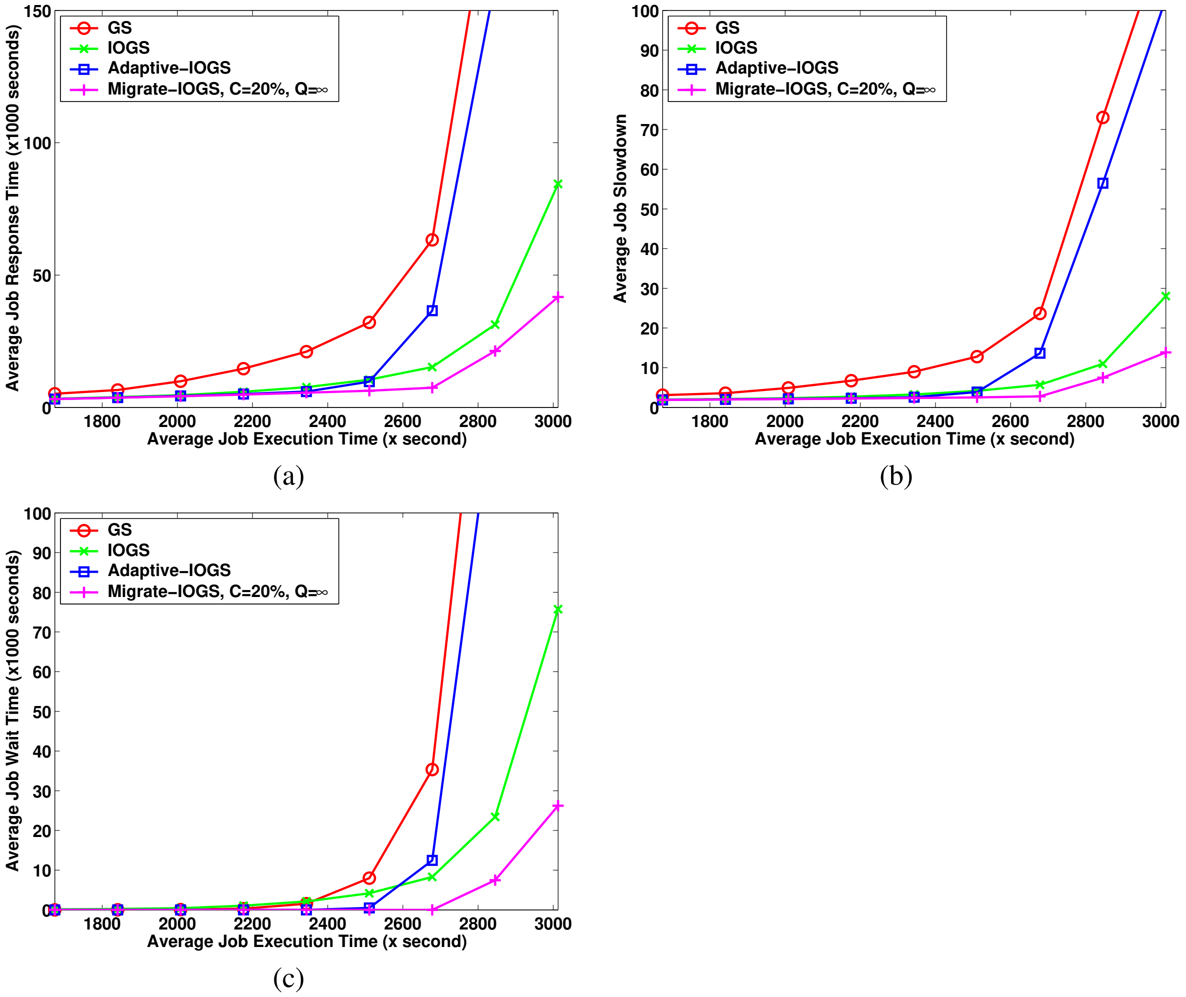


Figure 11: Comparison of Migrate-IOGS, Adaptive-IOGS, IOGS and GS for I/O intensive workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time. For Migrate-IOGS, we assume the migration cost 10% of the time quantum, and no limit on how many tasks can be migrated in the same quantum.

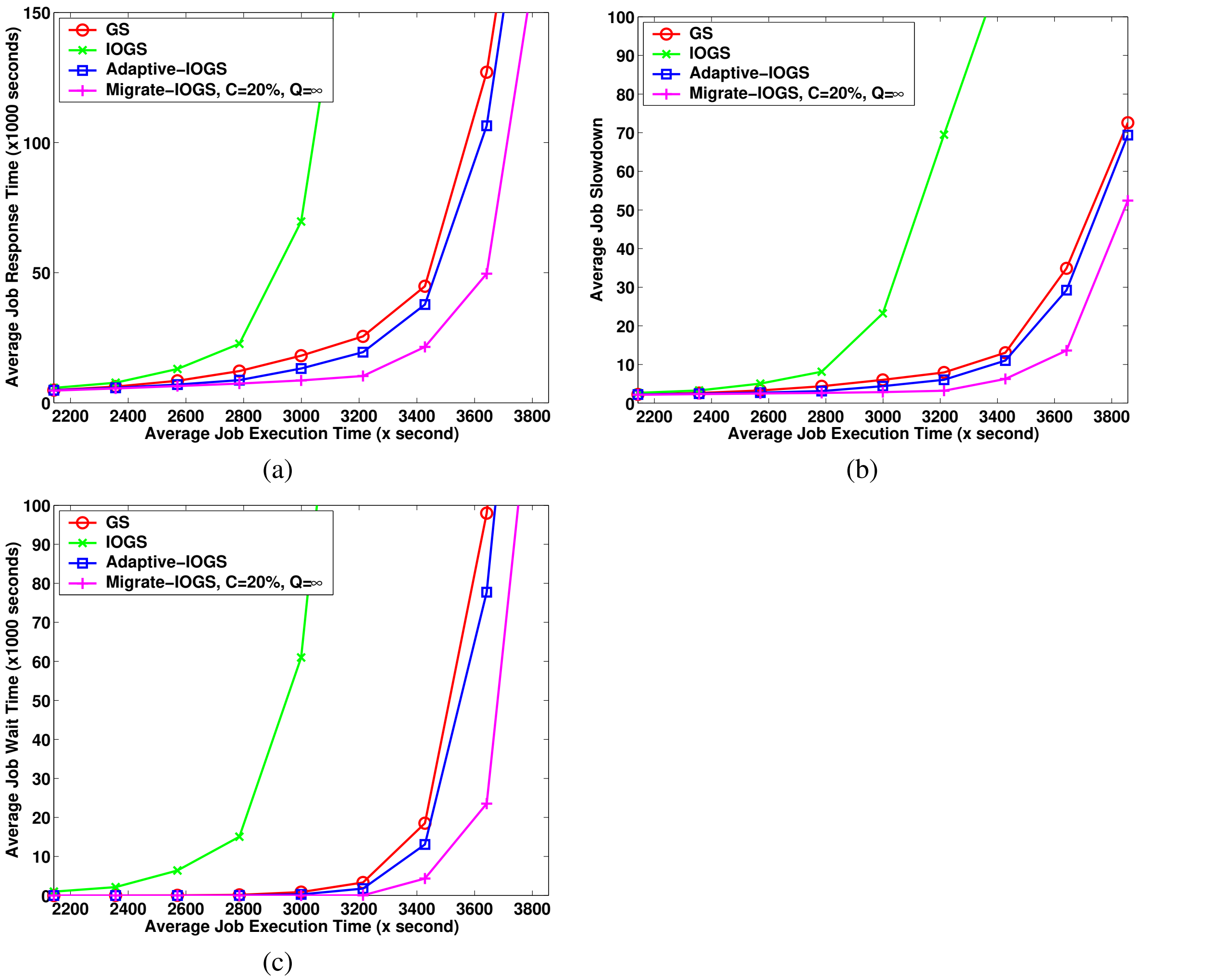


Figure 12: Comparison of GS, IOGS and Adaptive-IOGS for medium workloads: (a) average job response time, (b) average job slowdown, and (c) average job wait time.

4 Concluding Remarks

This paper studies the impact of job I/O intensity on the design of a job scheduler. Nowadays, scientific applications are becoming more complex and more I/O demanding than ever. For such applications, the system with dedicated I/O nodes does not provide enough scalability. Rather, a serverless approach [1, 2] is a viable alternative. However, with the serverless approach, a job's execution time is decided by whether it is co-located with the file blocks it needs.

Gang scheduling (GS), which is widely used in supercomputing centers to schedule parallel jobs, is completely not aware of the application's spatial preferences. We find that gang scheduling does not do a good job if the applications are I/O intensive.

We propose a new scheme, called IOGS (I/O aware gang scheduling), which assigns jobs to their file nodes, leading to a much shortened execution. Our results show that this scheme can improve the performance of I/O intensive workloads significantly, while it does not fare well for workloads with lower I/O intensity.

Further, we propose Adaptive-IOGS (Adaptive I/O aware gang scheduling), which takes the middle ground between gang scheduling and IOGS. Based on the matrix utilization, it adaptively assigns jobs either to their files nodes or just random nodes. We show that Adaptive-IOGS is the best for most of the workloads, except for those with high I/O intensity, and imposing very high load on the system.

Finally, we integrate migration technique to Adaptive-IOGS, by migrating jobs to their file nodes during their execution if those nodes are freed by others, leading to Migrate-IOGS (Migration adaptive I/O aware gang scheduling). Our results clearly show that Migrate-IOGS outperforms the other three significantly for all the workloads.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Roselli D. Patterson, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer System*, 14(1):41–79, 1996.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs . In *Proceedings of the ACM SIGMETRICS 2000 Conference on Measurement and Modeling of Computer Systems*, pages 34–43, 2000.

[3] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer System*, 14(3):225–264, 1996.

[4] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.

[5] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, April 1997.

[6] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, April 1997.

[7] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium*, pages 542–546, April 1998.

[8] H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proceedings of SC99, Portland, OR*, November 1999. IBM Research Report RC21559.

[9] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling for Highly Efficient Multiprocessors. In *Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Maryland*, 1996.

[10] B. Gorda and R. Wolski. Time Sharing Massively Parallel Machines. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.

[11] N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. Extensible Resource Management for Cluster Computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 561–568, 1997.

[12] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. Modeling of Workload in MPPs. In *Proceedings of the 3rd Annual Workshop on Job Scheduling Strategies for Parallel Processing*, pages 95–116, April 1997. In Conjuction with IPPS'97, Geneva, Switzerland.

[13] H. D. Karatza. A Simulation-Based Performance Analysis of Gang Scheduling in a Distributed System. In *Proceedings 32nd Annual Simulation Symposium*, pages 26–33, San Diego, CA, April 11-15 1999.

[14] D. Lifka. The ANL/IBM SP scheduling system. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer-Verlag, April 1995.

[15] X. Ma, J. Jiao, M. Campbell, and M. Winslett. Flexible and efficient parallel i/o for large-scale multi-component simulations. In *Proceedings of The 4th Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications, in conjunction with the 2003 International Parallel and Distributed Processing Symposium*, 2003.

[16] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. Yoo. A Gang-Scheduling System for ASCI Blue-Pacific. In *High-Performance Computing and Networking, 7th International Conference*, volume 1593 of *Lecture Notes in Computer Science*, pages 831–840. Springer-Verlag, April 1999.

[17] N. Nieuwejaar and D. Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[18] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[19] U. Schwiegelshohn and R. Yahyapour. Improving First-Come-First-Serve Job Scheduling by Gang Scheduling. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[20] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY-LoadLeveler API project. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, April 1996.

[21] K. Suzaki and D. Walsh. Implementation of the Combination of Time Sharing and Space Sharing on AP/Linux. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[22] Y. Wiseman and D. G. Feitelson. Paired Gang Scheduling. *IEEE Transactions on Parallel and Distributed Systems*.

[23] K. K. Yue and D. J. Lilja. Comparing Processor Allocation Strategies in Multiprogrammed Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 49(2):245–258, March 1998.

[24] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. The Impact of Migration on Parallel Job Scheduling for Distributed Systems . In *Proceedings of 6th International Euro-Par Conference Lecture Notes in Computer Science 1900*, pages 245–251, Aug/Sep 2000.

[25] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 133–142, May 2000.

[26] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling and Migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.