A Study on Data-Aware Scheduling for Post-Peta Scale Systems

March 2017

Xieming Li

A Study on Data-Aware Scheduling for Post-Peta Scale Systems

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2017

Xieming Li

Abstract

Handling large databases with uniform storage access file systems requires high-throughput networking systems, which incurs high cost in a large environment. While non-uniform storage access (NUSA) file systems federate local storages of compute nodes, which are able to scale out with lower cost. However, in this architecture, locality is quite important for effectively accessing files.

In this study, two scheduling algorithms–Data Aware Dispatch (DAD) and Improved Data Aware Dispatch (IDAD) are proposed to effectively dispatching tasks for NUSA file systems by taking advantage of the locality.

These two approaches are implemented on the top of stock Torque scheduler and evaluated with three benchmarks: thput-gfpio, Readgf, and BLAST benchmark. In the evaluation with thput-gfpio, the two data-aware approaches improved average read throughput from 448MB/s to about 7000MB/s. The reason for this huge difference is unveiled with Readgf benchmark. Finally, In an evaluation of BLAST benchmark, scheduler integrated with DAD and IDAD reduced the makespan for 29.88% and 33.52% in comparison with stock Torque scheduler, respectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Simulation technologies have become one of the most important branches for scientific computations in energy physics, genomics, astronomy and other fields. As the granularity of simulations increases, the demand for handling larger datasets is growing accordingly.

The European X-ray project XFEL [1] generates 10 to 50 petabytes of data per second, which requires its underlying file system to handle data at petabytes scale. As the performance of sensors improves and granularity of sampling data increases, this number grows exponentially.

Obviously, data of this scale is not able to fit in the main memory of computers. Access to external storage devices such as HDD, SSD and ioDrive is a necessity. However, the access performance of these devices is significantly slower than that of main memory. Therefore, the I/O access to these devices is usually the bottleneck of the entire computing system. Such challenge is met by distributed file systems that bundle multiple storage devices and achieves a high I/O performance through simultaneous access.

There are numbers of well-known distributed file systems such as GPFS [2], Lustre [3], pNFS [4], Gfarm [5], and Google File System [6]. These file systems can be classified into two main categories in term of access pattern: uniform storage access file system and non-uniform storage access file systems. In a former file system, all accesses to the storage devices have to go through the network, thus the access cost is nearly identical. while in a latter storage access file system, clients can access the data both on local and remote storage, where the accesses to local storage are considered more effective. This characteristic is referred as data locality or data affinity.

In addition, in a large-scale computing environment where a distributed file system is often deployed, it is rare for a single computing job to occupy all of the computing resources. More often, multiple jobs are executed in parallel, where a task scheduler is often employed to control the usage of hardware resources such as the CPU cycles, memory, and disk space. There are some widely deployed batch schedulers such as Sun Grid Engine [7], Torque [8], PBS Pro [9], LSF [10], and Slurm [11]. These schedulers are well suited for uniform storage access file systems. However, they do not consider data placement of underlying file systems, and thus are unable to exploit the data locality, resulting in poor performance on non-uniform storage access file systems.

Jobs can be classified into interactive jobs and batch (or non-interactive) jobs. In an interactive job, a user enters individual commands to be processed immediately, whereas in a batch job, a sequence of programs listed in a file is executed unattended. In this study, the main subjects are batch jobs as they are the mainstream in scientific computations.

The objective of this study is to propose a scheduling approach for non-uniform storage access file systems, which exploit the data locality to improve the performance. Here, two approaches named Data-Aware Dispatch and Improved Data-Aware Dispatch are proposed and implemented on an open source scheduler called Torque.

The contributions of this study are listed as follows:

- Existing algorithms for batch queueing systems either are unaware of data placement or try to improve the locality under the restriction of HDFS prerequisites. While in this work, I proposed two unique approaches for task dispatching in batch queueing systems that improve data locality without the prerequisites of HDFS.
- Two designs of data-driven scheduling algorithms, Data-Aware Dispatch (DAD) and Improved Data Aware Dispatch (IDAD), are proposed. DAD employs a global parameter to strike a balance between CPU load and locality, and IDAD enables a per-task parameter setting instead global one, and thus is more adaptive to different types of tasks.

- DAD and IDAD implementations on the Torque scheduler with Gfarm are presented along with an adaptation of Delay Scheduling [12].
- The *Score* of IDAD, a parameter to select the best execution node, is evaluated with Readgf and BLAST benchmark to show its effectiveness.
- Evaluations of schedulers integrated with DAD and IDAD using three benchmarks in comparison with stock Torque scheduler are also described and shows noticeable improvement.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of file system classification in relation to scheduling and a brief summary of schedulers. Then, two designs of data-aware approaches are described in Chapter 3, followed by their implementation in Chapter 4 and evaluation in Chapter 5. Chapter 6 introduces previous research related to the present topic, and Chapter 7 provides some concluding remarks.

Chapter 2

Background and Motivation

In this chapter, the background and the motivation for this study are described. Section 2.1 introduces a classification of distributed file systems. In addition, since this is a study for schedulers. The taxonomy of scheduler is described in Section 2.2. Moreover, Gfarm file system and Torque scheduler are used as the base of this study, therefore, they are described in detail in Section 2.3 as the technical background. Finally, the study purpose is described in Section 2.4.

2.1 The Uniformity of File Systems

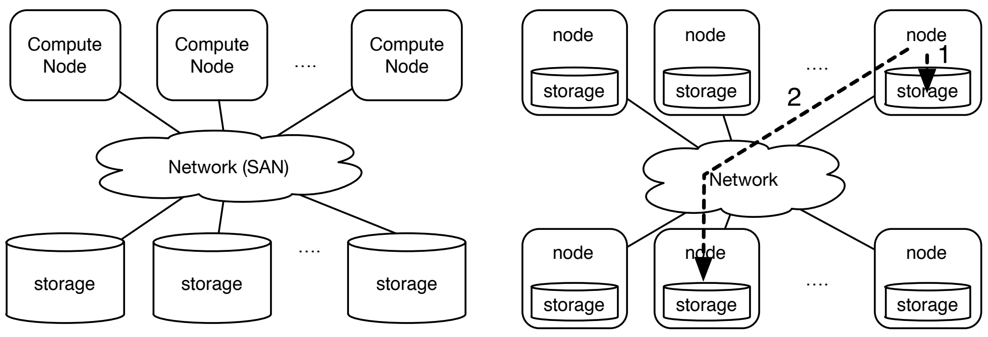


Figure 2.1: The architecture of uniform storage access file system and non-uniform storage access file system

Distributed file systems can be classified into two categories in terms of the accessing pattern: uniform storage access file system and non-uniform storage access file systems. GPFS, Lustre, pNFS and PVFS [13] require dedicated storage nodes connected to compute nodes using a storage area network (SAN). The performance in accessing files on such file systems can be considered rather “uniform” because each access has to travel through the network (Figure 2.1, left). However, this approach requires a high bandwidth network between the compute and storage nodes, which could incur a relatively high cost, especially for a large-scale cluster.

In contrast, non-uniform storage access (NUSA) file systems such as Gfarm and Google File System, which federate local file systems on compute nodes, have been proposed. In this type of file system, the access performance can be considered “non-uniform” because the compute node can now access the files on its local drive (route 1 in Figure 2.1, left) as well as files on the remote node through the network (route 2 in Figure 2.1, right).

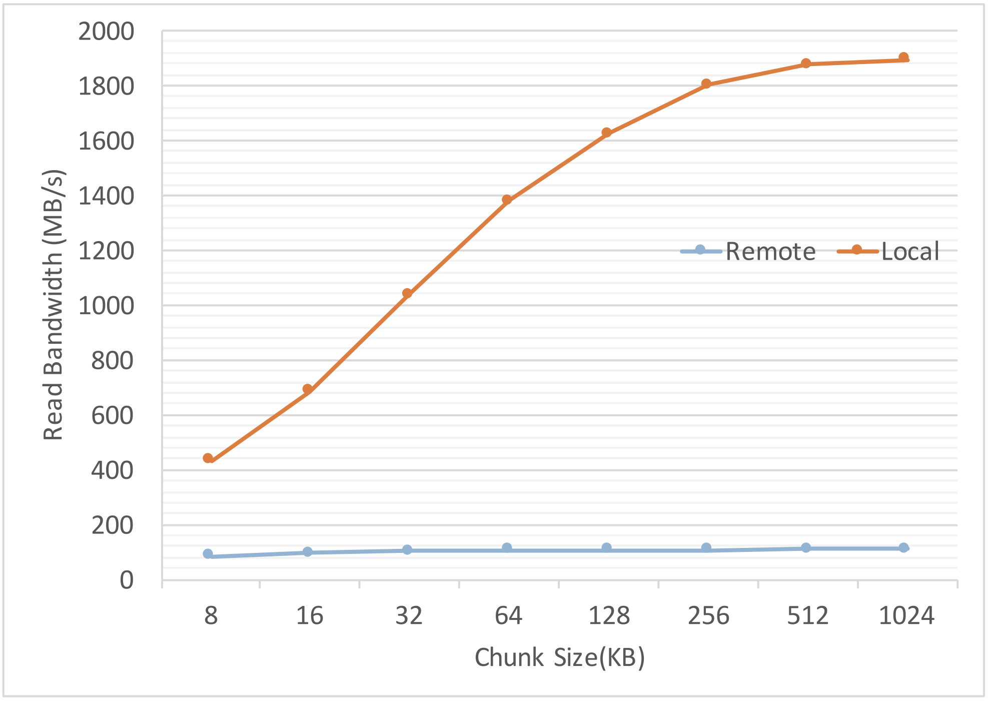


Figure 2.2: Performance variation of Gfarm

The difference in throughput between local and remote access may be significant in NUSA file systems. I have conducted an evaluation of Gfarm file system with the thput-gfpio benchmark, which is a component included in Gfarm and aims to evaluate I/O bandwidth. The result (Figure 2.2) shows that local access dominates remote access in all chunk sizes.

2.2 Task Schedulers for Cluster

In a large-scale computing environment, it is rare for a single task to occupy all computing resources. More often, multiple jobs are executed in parallel, where a batch scheduler is often utilized to control the usage of hardware resources. In this section, a brief introduction to scheduler is given, and the Torque scheduler is described in detail as it is used as the base of this study. In terms of architecture, schedulers can be divided into three categories: monolithic, offer-based (or two-level), and shared-state schedulers. These architectures are described below.

2.2.1 Monolithic Schedulers

Monolithic schedulers are the most common type of schedulers in high-performance computing (HPC) environment. Some well-known monolithic schedulers include Torque, PBS Pro, LSF, and Slurm, Platform Load Sharing, Open Grid Scheduler [14] and Stock Scheduler for MapReduce in Hadoop 1.x [15]. A monolithic scheduler has a single process for controlling and making scheduling decisions (e.g sge_qmaster in Open Grid Scheduler, pbs_server in Torque, slurmctld in Slurm), as well as multiple daemons for monitoring task execution (sge_execd in Open Grid Scheduler, pbs_mom in Torque, slurmd in Slurm). In a monolithic scheduler, a centralized scheduling algorithm is responsible for all jobs.

Utilizing one single algorithm for all of the jobs may not be ideal, especially in some commercial environment with heterogenous workloads. For example, in Graviton, a cluster consists of 4000+ nodes in Yahoo Japan, has deployed with Hadoop 2.X, where multiple types of jobs such as MapReduce, HBase, PIG, and HIVE are executed concurrently. In such kind of clusters, maintaining a monolithic scheduler means the scheduler has to be updated with new job logic every time a new job type is added, which increases the complexity of implementation of the scheduler.

2.2.2 Offer-based schedulers

Offer-based schedulers that known as Two-level schedulers are designed to tackle this problem by separating the resource allocation and task logic. YARN [16] and Mesos [17] are the typical schedulers of this type. In an offer-based schedulers, a centralized resource manager first allocates computing resources to different application-specific schedulers (make an offer), and each of those schedulers then makes their own task allocation based on its own policies.

YARN scheduler is the default scheduler included in Hadoop 2.X. In a typical YARN scheduler setting, there is one Resource Manager and Multiple Node Managers. Resource manager resides on a server node, which is merely responsible for allocating resources, while Node Managers are responsible for managing an application-specific scheduler called Application Master and a set of resource called Container. When a job is allocated, it will be assigned an Application Master. Application Master will first negotiate with Resource Manager to acquire Container(s). On success, it will ask Node Manager to initiate the actual task.

Mesos scheduler is an open source platform for fine-grained resource sharing between multiple diverse cluster computing frameworks. As an offer-based scheduler, it works in a similar way to YARN: A Mesos master to allocate the resource, multiple slaves to manage application-specific schedulers called Framework.

The offer-based scheduler decoupled the resource allocation and task logic, which offered more flexibility when adding new features. However, due to the fact that the resource was allocated before the task allocation, the task-specific scheduler now can only select best task placement from available resources given by resource manager from the previous step, which may lead to low resource utilization.

2.2.3 Shared-state schedulers

Shared-state schedulers address the problem by allowing each application-specific scheduler maintains its own cluster state as well as a centralized shared-state cluster table. When an application-specific scheduler wants to use some resources and update its own cluster state, it issues an concurrent transaction to update the shared-state cluster table.

The most famous shared-state scheduler is Omega [18] scheduler introduced by Google. Omega maintains a shared-state cluster table called Cell State. Each application-specific scheduler maintains a local copy of Cell State, which is utilized to making scheduling decisions. When the scheduler made a decision, it tries to update the shared Cell State atomically. When there is a contention, one of the scheduler will succeed to get required resource, and the rest will have to retry.

2.3 Torque Scheduler and Gfarm File System

In this section, the details of the Torque Resource Manager and Gfarm File System are briefly summarized, the former being used as the basis for my implementation, and the latter having certain key features utilized in the proposed work.

2.3.1 Torque Scheduler

Torque scheduler is one of most accepted scheduler in HPC environment. It is an open-source product derived from the original PBS project. By default, it has an FCFS scheduler module called pbs_sched. However, in many production environments, pbs_sched was substituted with Maui [19] scheduler.

Torque scheduler is a typical monolithic scheduler. The main architecture is shown in Fig 2.3. It consists of three main components, i.e., pbs_server, pbs_sched, and pbs_mom.

pbs_mom resides on execution hosts, as is responsible for controlling the job, transferring output, and collecting load information for the

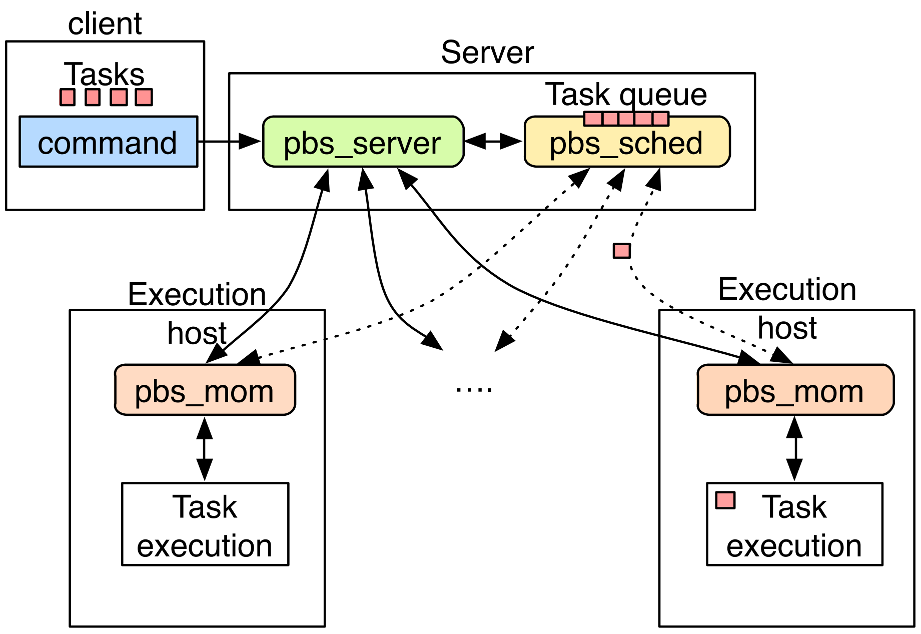


Figure 2.3: Architecture of Torque scheduler

pbs_server.

pbs_sched accepts commands from pbs_server, maintains the latest information on the execution hosts, and makes scheduling decisions which will be sent back to pbs_server.

pbs_server is the central part of the entire system, accepts tasks from clients, initiates scheduling on pbs_sched, and monitors the jobs on pbs_mom.

In Torque, a task is first submitted to pbs_server with *qsub* and queued. At the same time, the pbs_server will send a signal to pbs_sched to initiate a scheduling circle, in which pbs_sched gets all details about jobs and execution nodes from pbs_sched, and then makes decisions on task allocation. Finally, decisions are sent back pbs_server and be carried out.

2.3.2 Gfarm File System

The Gfarm file system is a globally distributed file system used to share data and support distributed data-intensive computing. It is an open source project maintained by The University of Tsukuba.

Instead of setting a dedicated storage node, it federates the local file systems of compute nodes, and manages them using a single namespace. The Gfarm file system has multiple instances of gfsd as the storage daemons which access the local file system, and a master-slave gfmd as the metadata daemon, which manages the file metadata including the hierarchical namespace, file properties, directory structure, and replica information.

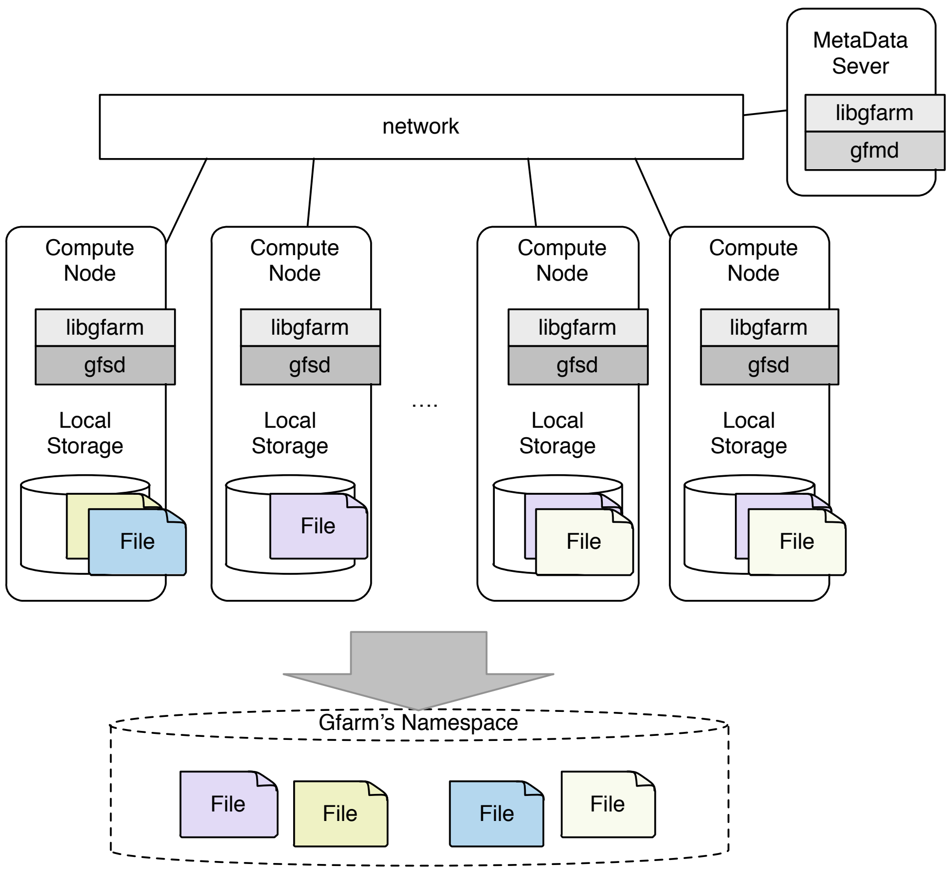


Figure 2.4: Basic structure of Gfarm

Gfarm file system is NUSA file system because each storage node can be utilized as compute node, and it can access its local storage using gfsd, as well as its remote storage using gfsd on other nodes via the network (see Figure 2.4). Gfarm does not divide one file into multiple nodes, but it achieves a scalable performance by duplicating the replica in several nodes. Gfarm can be accessed through the Command Line Interface (CLI), API, and FUSE [20] through a plug-in called gfarm2fs.

Gfarm is designed in a way to maximize the locality. When opening a file descriptor, the libgfarm will first communicate with metadata server and acquire information, such as RTT, CPU load level, and available space on a storage node, which will be used to select the most suitable storage node. Normally, The node tries to open the file will be selected to maximize the locality.

2.4 Motivation and Targeted System

Some of the well-known schedulers are shown in Figure 2.5. Google File System (GFS) and HDFS [21] shown in the figure are the most widely deployed NUSA file system which are the underlying file systems of MapReduce. The JobTracker and YARN are the default schedulers for MapReduce 1 and MapReduce 2, respectively. Since GFS and HDFS are NUSA file systems, the locality is a crucial concern. However, schedulers and optimizations for improving locality for such file systems are often based on unique characteristics of MapReduce, which makes them inapplicable for NUSA file system designed for batch queuing systems like Gfarm. For example, Delay Scheduling, one of the most cited works for optimizing locality in MapReduce, is not directly applied to Gfarm, because a clear definition of 'local' node does not hold true in that file system.

In addition, there are some widely deployed task schedulers in HPC environment which are batch queuing systems–Torque, PBS Pro, LSF, and Slurm, etc. Most of these schedulers had only assumed local storage access when they are designed. The main information they depend on is CPU load and available memories.

Those schedulers are suitable for uniform storage access file systems mainly because there is no need to consider the file location at the task dispatch.

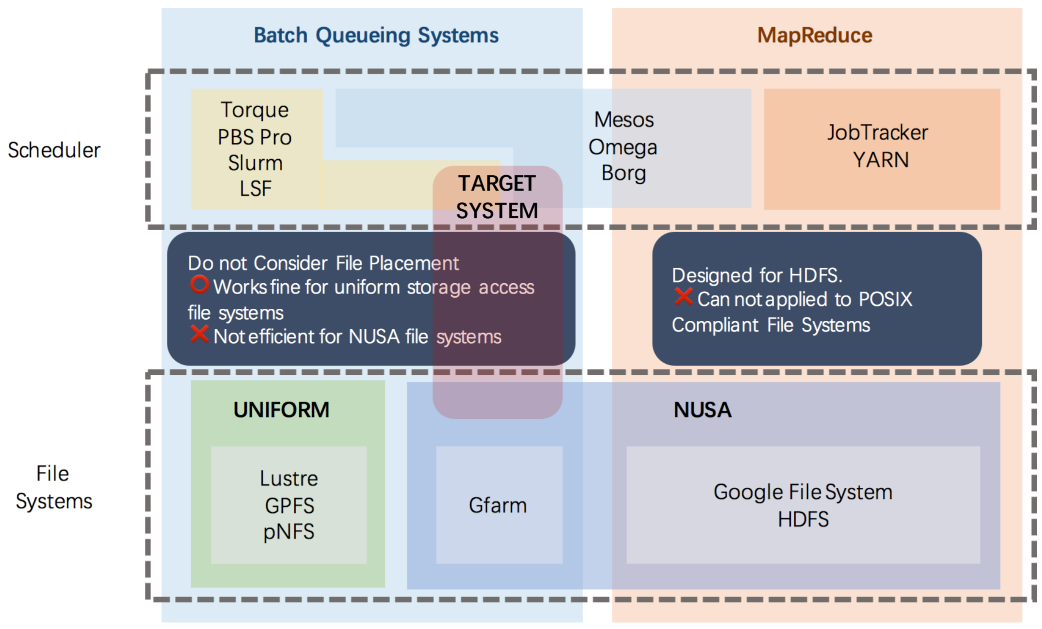


Figure 2.5: Target System

The access performance is nearly identical regardless of which compute node the task is dispatched to.

However, such schedulers may not be ideal for a NUSA file system because if a task is dispatched without consideration of the file placement, it might be assigned to a node where the file cannot be accessed locally, thereby leading to a drop in performance.

Moreover, some of schedulers like Mesos, Omega and Borg [22] are capable of handling workloads of both batch queueing systems and MapReduce. However, they usually assume the prerequisites of MapReduce while designing their approaches for improving the file locality. For instance, the Mesos scheduler applied Delay Scheduling in their implementation while Delay Scheduling is not directly applicable to Gfarm.

In summary, to effectively execute tasks on NUSA file systems for batch queueing systems, the data placement has to be taken into consideration and local access should be exploited. There are many approaches for Hadoop/MapReduce but they are not able to be applied to batch queuing systems directly. The main purpose of this study is to propose a scheduling strategy for batch queueing systems on the top of NUSA file systems that emphasize the exploitation of high-performance local access.

Chapter 3

Design of Data-Aware Scheduling

In this study, two designs of data-driven scheduling algorithms have been proposed: Data-Aware Dispatch (DAD) introduces a parameter $fileLocality$ to indicate the cost of accessing data and a global parameter to strike a balance between $fileLocality$ and CPU load, and Improved Data-Aware Dispatch (IDAD) enables a per-task parameter setting instead global one. In IDAD, each task specifies a parameter called RDR that indicates the performance degradation if the task is located on the remote node and thus is more adaptive to different types of tasks.

3.1 Prerequisites

Before proceeding to the algorithms, I would like to make prerequisites clear in this section first. As we stated in Section 2.3, the Gfarm file system and Torque scheduler are used in this study, some key features which affect the design of algorithm will be summarized here.

Machine Specification

- The cluster is homogeneous, i.e. nodes in the cluster have similar specs, and thus have similar performance.

- The remote accesses between nodes have similar performance since the network latency is quite small in HPC environment. For the simplicity of the model, I assume the nodes in the cluster are connected to one switch directly.
- The connection bandwidth between nodes are much slower than I/O bandwidth of local storage, thus exploiting file locality is a crucial concern.

File System

- The file system is POSIX compatible, which allows vast of existing scientific applications to be executed.
- In the file system, a file is not divided into multiple chunks, instead, it can be replicated and distributed to multiple storage nodes.
- In the file system, a storage node acts as a compute node at the same time. When a task is dispatched to a compute node, local storage is preferred.

Task Scheduler

- The scheduler is a monolithic or a shared-state scheduler. Hence it can have the whole cluster view to make optimal task allocation among all nodes in a cluster.
- Tasks scheduled by the scheduler are general-purpose, which means they may refer to an arbitrary number of files.

3.2 Data-Aware Dispatch (DAD)

Most of the existing works have not assumed a NUSA file system as their underlying system, therefore, they do not take file placement into consideration, and thus are not suitable for NUSA file systems. Other works, especially those of Hadoop, depend on some particular constraints, such as a fixed number of replicas or a fixed task size. However, these assumptions do not hold true for some general purpose NUSA file systems such as Gfarm. In this section, Data-Aware Task Dispatch (DAD) [23] is proposed to exploit local access regardless of those conditions.

3.2.1 File Locality and Score

The traditional scheduler takes the CPU load-average as the primary factor when selecting a compute node for a specific task. In contrast, DAD introduces $fileLocality$, a parameter that indicates the difficulty of accessing the dataset and combines it with load-average as a comprehensive $Score$ to determine the most suitable node. These two parameters are described in detail below.

fileLocality

The $fileLocality(t, h)$, which indicates the difficulty of accessing the dataset referenced by task t when it is dispatched to a specific compute node h, is defined as follows:

$$\begin{aligned} fileLocality(t,h) &= [\frac{\sum_{y=1}^{n} locality(f_y,h)}{\sum_{y=1}^{n} sizeof(f_y)} + 1]/2 \\ locality(f_i,h) &= \begin{cases} -sizeof(f_i) & \text{if } on(f_i,h) \\ sizeof(f_i) & \text{other} \end{cases} \end{aligned} \tag{3.1}$$

where the $locality(f_i, h)$ is a value determined by the size of file f_i and whether compute node h has a replica of f_i. If one of the replicas of f_i is on h, the cost of accessing it will be smaller, and therefore the file size of f_i will be subtracted to make the "cost" smaller, and vice versa. The $fileLocality(t, h)$ is the normalized sum of the $locality(f_y, h)$ ranges $[0, 1]$.

When calculating $fileLocality$, the total size of local files and remote file are calculated respectively. Next, subtract the total size of local files from the total size of remote files to get a preliminary $fileLocality$. Finally, formalize this value to range $[0, 1]$, which is the $fileLocality$.

For better understanding, an example of calculating $fileLocality$ is given in Figure 3.1. In this example, there is one task that accesses three files, which are 30M, 40M, and 50M respectively. Also, there are four computing nodes, each has a part or all of the files required by the task. Equation 3.1 will be used to calculate the *fileLocality* of all four compute nodes.

Consider the leftmost node in Figure 3.1, it refers to one remote and two local files. As you can see in the figure, the total size of remote and local files are 40M and 80M respectively. Since the remote file has to be accessed through the network, the size 40M is added to the cost. On the other hand, the two local files can be accessed locally, therefore the total size of 80M will be subtracted from the cost. In all, the preliminary *fileLocality* is -40M, and the formalized *fileLocality* is 0.34.

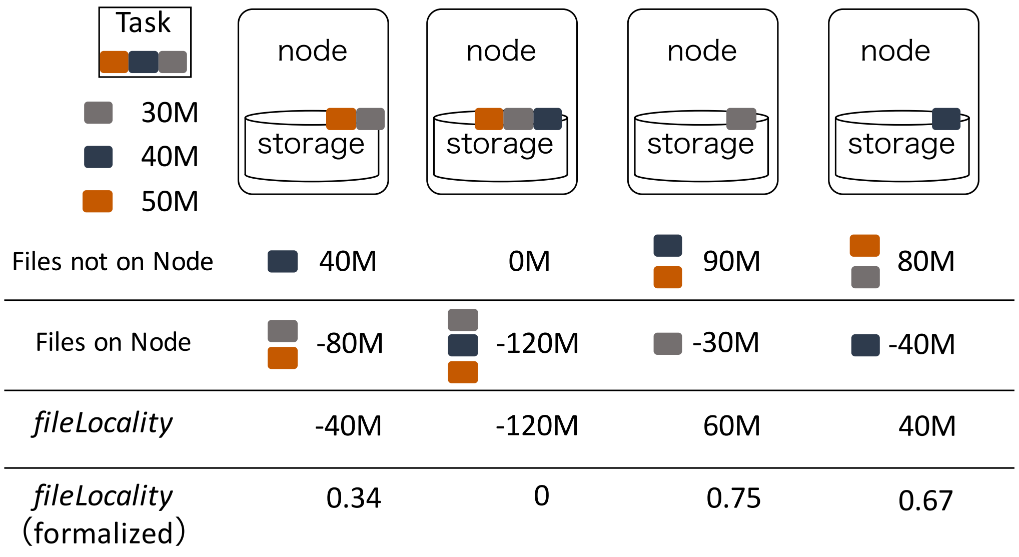


Figure 3.1: Calculating *fileLocality*

Score

The comprehensive *Score* can be calculated in advance using the *fileLocality* as follows:

$$\begin{aligned} Score(t,h) =& fileLocality(t,h) \times \beta \\ &+ load(h) \times (1-\beta) \quad (0 \le \beta \le 1) \end{aligned} \tag{3.2}$$

The *load* (load-average) and *fileLocality* are unified into *Score* using parameter β. Here, β is a modifier used to adjust the strength of DAD. When $\beta = 1$, the scheduler will ignore the CPU load at dispatch. Although there should be a method for acquiring the optimal value of β, I only show the effectiveness of this particular parameter at this stage.

Score can now be used to judge whether a host is desirable for a job execution in the exact way in which the load-average is used in a CPU-focused scheduler, with consideration of both the CPU load and the file locality.

3.3 Improved Data-Aware Dispatch (IDAD)

DAD has a parameter β to strike a balance between *fileLocality* and CPU load, yet β is quite difficult to calculate. Considering the fact that CPU load does not have a major impact on execution time, a more data-centric approach called Improved Data-Aware Task Dispatch (IDAD) [24] is proposed in this section.

3.3.1 Drawback of Data-Aware Dispatch

DAD worked fine in a simple experiment setting where all of the tasks have a similar workload. However, problems arise when dealing with some real applications, which are summarized as follows:

How to Determine β

In DAD, β is a key parameter for striking a balance between *fileLocality* and CPU load. However, in real-world situations, tasks dispatched by a scheduler may have quite different characteristics. Some tasks may be I/O-intensive which requires β to be set to a value close to 1, while others could be CPU-intensive and prefer smaller β values. Since β in Equation 3.2 is a global parameter that affects all tasks dispatched by the scheduler, it is not easy to determine a suitable β that works for all kinds of tasks.

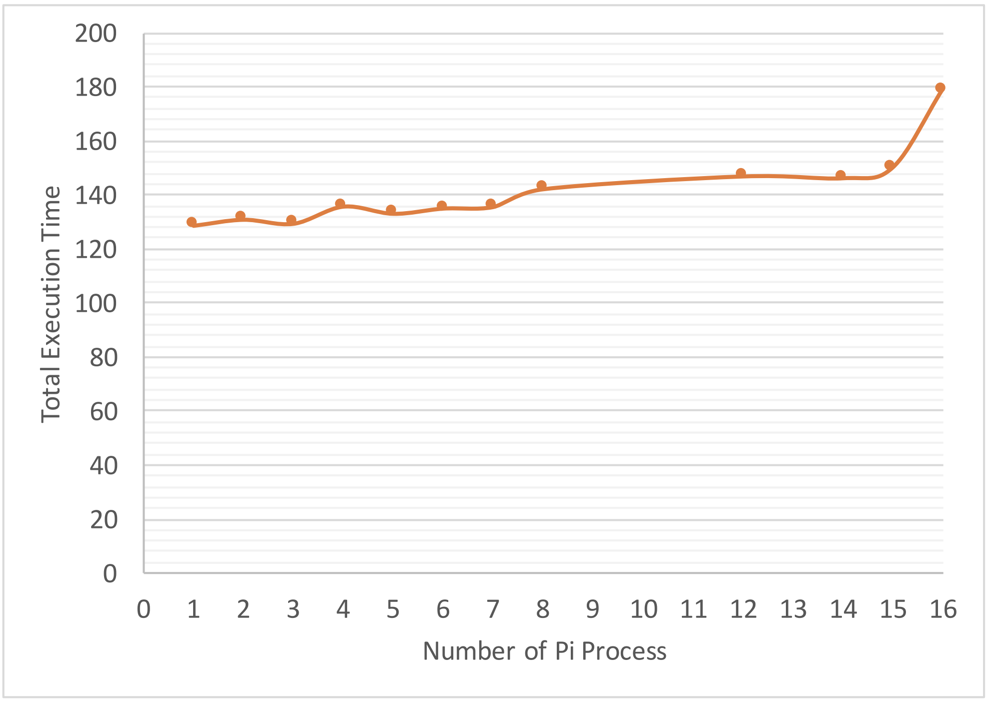


Figure 3.2: CPU impact on BLAST benchmark

Insignificant CPU Impact

A task scheduler has computing slots configured for each compute node, which limits the number of tasks that specific node can run concurrently. In a typical setting, the number of computing slots is smaller than that of CPU cores, which means that the number of concurrent running tasks will not exceed the number of cores on any specific compute node.

We evaluated the impact of CPU load on computing tasks by monitoring the total running time of one set of blastn tasks with a different number of Pi tasks running at the same time. The result is shown in Figure 3.2. As you can see from the figure, the total execution time of blastn only slightly increased as the number of Pi processes increases up to 15. However, it increases dramatically when there are 16 Pi processes, as there are actually 17 processes including the blastn process.

The machine used in this evaluation has 16 cores, thus the CPU loads will not significantly impact execution time if the number of concurrent running tasks is less than this. Therefore, considering CPU load could be beneficial in terms of load balancing, but unlikely to reduce the task execution time.

3.3.2 Design of Improved Data-Aware Dispatch

A traditional scheduler takes the CPU load-average as the main standard for load balancing. Just like DAD, IDAD also defines a *Score* for selecting the best node at dispatch phase. In DAD, the only user-defined parameter is β and it is a global parameter that affects all tasks scheduled. For precise control of each task, I introduce a per-task parameter called Remote Degradation Rate (RDR) to indicate the extent to which a task is data-intensive. In addition, a parameter called RemotePortion is introduced to indicate the portion of files that has to be accessed remotely.

RDR

Unlike the DAD, which utilizes a global parameter to control the effect of data placement, the IDAD enables per-task parameter called Remote Degradation Rate (RDR) to control scheduler in advance.

RDR is defined as follows:

$$RemoteDegradRate = \frac{RemoteTime(t) - LocalTime(t)}{LocalTime(t)} \tag{3.3}$$

where $RemoteTime(t)$ is the execution time when a task t runs on a remote node. Likewise, $LocalTime(t)$ is the execution time when a task t runs on a local node.

The range of RDR is $[0, \infty)$. When $RemoteTime(t)$ equals $LocalTime(t)$, RDR is zero, which means for this specific task t, executing remotely or locally does not affect execution time and therefore it is not a data-intensive job. On the contrary, if the remote running time and local running time differs greatly, the part $RemoteTime(t) - LocalTime(t)$ will be much larger, and thus lead to bigger RDR value.

The definition of RDR is quite important as it not only represents the characteristic of jobs, but also show how such characteristic will affect real execution time in the given environment.

RemotePortion

IDAD uses RDR to characterize the job. RemotePortion, on the other hand, represents the characteristic of file placement for the given job. This idea is quite similar with $fileLocality$ in DAD. The RemotePortion is defined as follows:

$$\begin{aligned} RemotePortion(t,h) &= \frac{\sum_{y=1}^{n} RemoteSizeof(f_y,h)}{\sum_{y=1}^{n} Sizeof(f_y)} \\ RemoteSizeof(f_i,h) &= \begin{cases} 0 & \text{if } on(f_i,h) \\ sizeof(f_i) & \text{other} \end{cases} \end{aligned} \tag{3.4}$$

where the $RemoteSizeof(f_i,h)$ is a value determined by the size of file f_i and whether compute node h has a replica of f_i. Here, only the size of remote the file will be counted, which means if replicas of f_i is not on h, its size will be counted. The $sizeof(f_y)$ is the file size of f_y. In short, the $\sum_{y=1}^{n} RemoteSizeof(f_y,h)$ is the total size of files accessed remotely and $\sum_{y=1}^{n} Sizeof(f_y)$ is the total size of files accessed by the task.

Score

Finally, the $Score(t,h)$ then can be defined as follow:

$$Score(t,h) = RDR(t) \times RemotePortion(t,h) \tag{3.5}$$

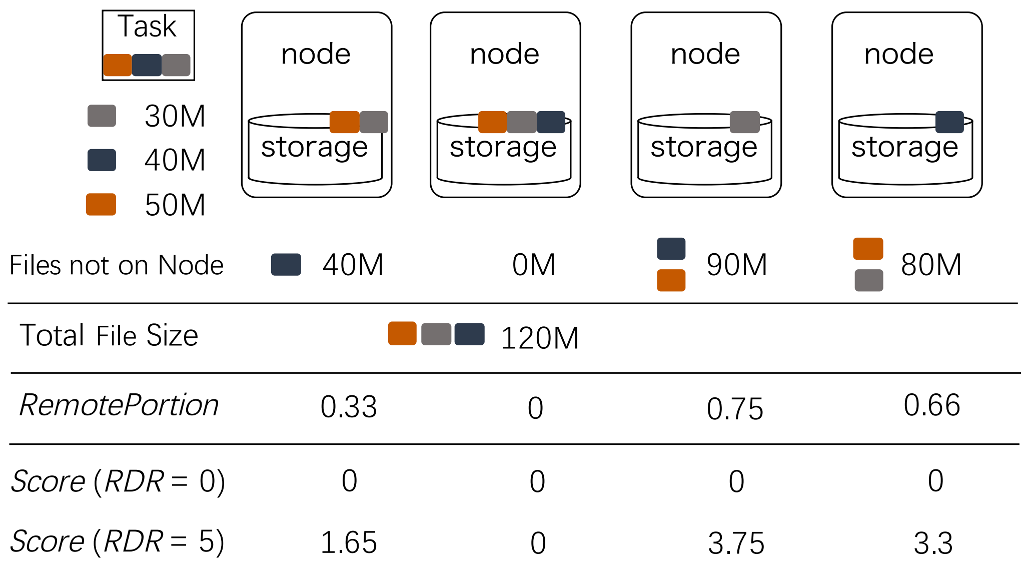
At the dispatch phase, when the scheduler tries to select the best node h for task t, IDAD calculates the $Score(t,h)$ for each available node h; and the node with the lowest $Score$ is chosen as the execution node. As you can see in Equation 3.5, when the $score$ is low, either the RDR is low or the $RemotePortion$ is low. The former means that the task is not that data-intensive and does not require high-level locality, while the latter means that compute node has the majority of the files needed by the task.

For better understanding, an example of calculating $score$ in IDAD is given in Figure 3.3. In this example, the setting is similar to the one in the previous section with DAD, but it has an additional parameter–RDR. Similarly, there is one task that accesses three files, which are 30M, 40M, and

50M respectively. Also, there are four computing nodes, each has a part or all of the files required by the task.

Consider the leftmost node in Figure 3.3, it has one 40M remote file, and the total file size is 120M. Therefore, we can calculate the $RemotePortion$ of this node by Equation 3.4, which is 0.33. When the RDR of a task is 0, according to Equation 3.5, all of the nodes have $Score = 0$ no matter what $RemotePortion$ they have. On the contrary, if the RDR of the task is 5, then the $Score$ will be 1.65 for the leftmost node.

Just like how $Score$ works in DAD, the $Score$ for IDAD will be used to judge whether a host is desirable for a job execution.

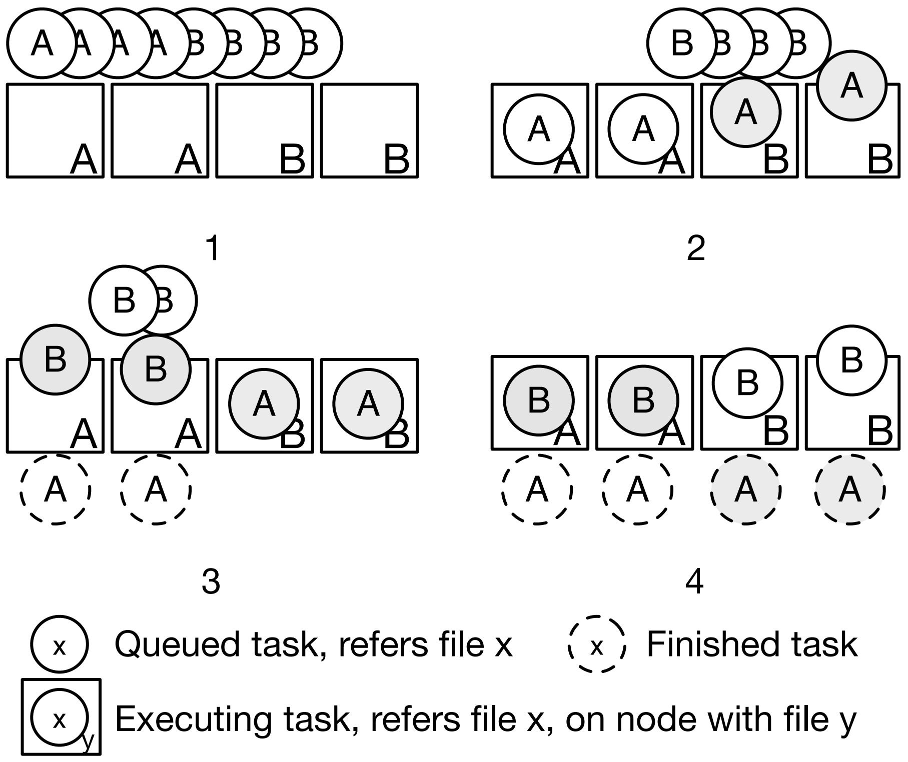


Files not on Node	40M	0M	90M	80M
Total File Size	120M			
RemotePortion	0.33	0	0.75	0.66
Score (*RDR* = 0)	0	0	0	0
Score (*RDR* = 5)	1.65	0	3.75	3.3

Figure 3.3: Calculating $Score$ in IDAD

3.4 Delay Scheduling for Data-Aware Scheduling and Local Threshold

Some task sequence may cause a serious problem in schedulers implemented with DAD or IDAD. In this study, Delay Scheduling is applied to solve this problem. However, Delay Scheduling is designed for Hadoop/MapReduce that has different assumptions from mine, therefore, an adaptation is required. Details on this issue are described below.



1
2
3
4
Queued task, refers file x
Finished task
Executing task, refers file x, on node with file y

Figure 3.4: Task order causing performance degradation: after the first two tasks executed locally (phase 2), the following two tasks will be dispatched to the two available remote nodes. This situation also happens in phase 3. Finally, half of the tasks have to access the file remotely

3.4.1 Delay Scheduling

I found that the order of the tasks might cause a drastic degradation in the performance. An example of this is shown in Figure 3.4, where four task requiring file A; and the other four, file B.

Because there are two nodes for each referenced file, the ideal case is for each node to be dispatched with two local tasks. This can be achieved by the arranging tasks as AABBAABB. However, if the tasks come in the order of AAAABBBB, after the first two tasks are dispatched (phase 2 in Figure 3.4), the following two tasks will be dispatched to the two available nodes remaining without a needed file. Finally, half of the tasks have to access the file remotely, which will cause a significant drop in performance for data-intensive tasks.

I applied Delay Scheduling (DS) to alleviate this issue. DS is a simple idea for a scheduler to achieve locality in the Hadoop file system. In DS, when a task is to be dispatched according to the scheduling policy but has no local node, instead of being executed immediately, it waits for a few slots so that it can be executed locally.

3.4.2 Local Threshold

DS is designed for MapReduce, where DS recognizes three locality level: local, rack, and off switch. In another word, if a task waits long enough, there will always be one or more local node.

However, in some NUSA file systems like Gfarm, a task may refer to more than one dataset distributed in multiple compute nodes. Therefore, there are cases where no node holds all data required and is 100% local to a task.

To solve this problem, in this study, I introduced the concept of local threshold: tasks are classified by comparing the $Score(t, h)$ and a local threshold value. A node h is local to task t if $Score(t, h)$ is smaller than the local threshold *lThreshold*. Otherwise, h is considered remote.

3.5 By Queue Scheduling

Job array submission is a feature which allows a user to submit a large number of tasks based on the same job script with different parameters. Consider that *Score* is calculated based on file sets referenced by a task, submitting a job array in which tasks referring the same datasets might cause excessive scheduling overhead because DS has to handle tasks with the same *Score*, which could be a waste of time. I solve this issue by submitting different job arrays to different queues and dispatching tasks in each queue interleavely,

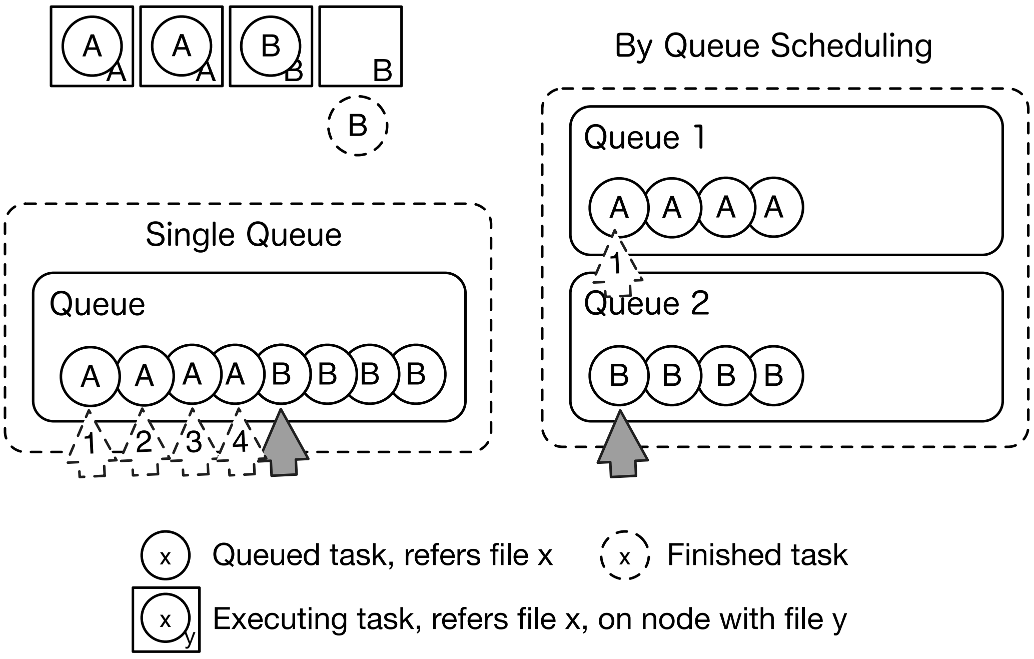


Figure 3.5: Example of By Queue Scheduling (BQS): the scheduler has to judge and skip four times before finding a local task without BQS. That number is one with the BQS

and thus tasks referring to different files were processed in an interleaving manner to avoid excess judgment.

A example of By Queue Scheduling is Given in Figure 3.5. In this example, when a local task refers file B finishes, the scheduler will try to dispatch another local job, which also refers file B to maximize the locality. If there is only one queue, as you can see in the figure, the scheduler has to judge and skip first four task because they all refer file A, and then find a local task. On the other hand, if tasks refer to different file are submitted to different queues, and scheduler process each queue in an inter-leaving manner, after only one failure judge in Queue 1, a local task will be found for execution.

Chapter 4

Implementation of Data-Aware Scheduling

I implemented DAD and IDAD based on the stock Toque scheduler included in Torque package and chose Gfarm as the underlying file system. In this Chapter, some of the details of implementation are discussed.

4.1 Architecture

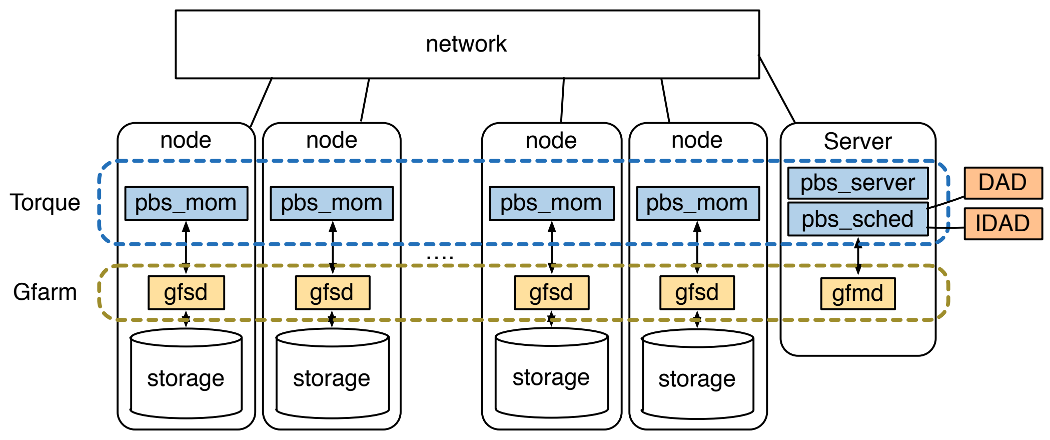


Figure 4.1: Structure of Gfarm on Torque

The entire system consists of one server and multiple worker nodes. Each worker nodes has a gfsd and a pbs_mom, which are storage access daemon of Gfarm and task execution daemon of Torque, respectively. The server, on the other hand, is responsible for controlling the entire system, where the pbs_server, pbs_sched of Torque and metadata server, gfmd, are located.

The system is configured in a way that a task initiated by pbs_mom is able to access its local storage through gfsd on the same node. The two proposed methods are implemented and integrated into the pbs_sched, which is the default scheduler module of Torque.

4.2 Modification on Torque

In this work, the main modifications are made to pbs_sched, whereas pbs_mom remains completely untouched. In addition, the pbs_server and qsub commands are minimally changed to pass information regarding the referenced data of a task. Figure 4.2 show the flow of information, from user to final scheduling module.

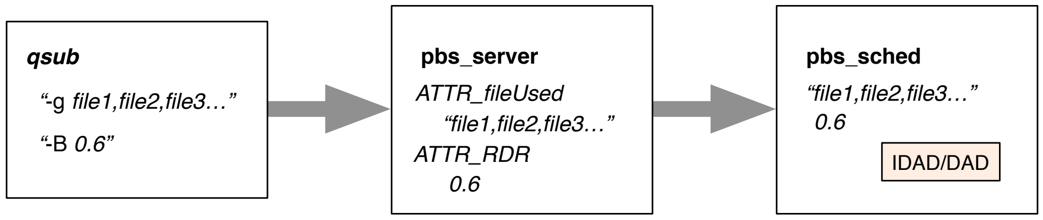


Figure 4.2: The Flow of Job Information

Modification on Client Command

Both DAD and IDAD requires the information about file used by jobs, which has to be specified by the user. As the command to submit a job, the *qsub* are expanded with the ability to specify the file accessed and the Remote Degradation Rate (RDR).

In stock Torque scheduler, a user can submit a batch file through the following command:

```
qsub task.sh
```

We expanded the system using the command-line option -g followed by a comma-separated file list, allowing a user to specify a set of files through the *qsub* command. Similarly, option -B is added to specify the RDR of a task for IDAD.

For example, if we want to submit a task that accesses two files named file1 and file2, and has RDR of 0.6, we can use the following command:

```
qsub -B 0.6 -g file1,file2 task.sh
```

Alternatively, the file referenced and RDR can be specified by writing following directives in the batch script.

```
#!/bin/bash
#PBS -N ExampleJobName
#PBS -g file1,file2
#PBS -B 0.6

actual commands follows...
```

This information will be sent to pbs_sched and stored in its data structure.

Modification on pbs_server

pbs_server is the controlling center of Torque, which receives jobs information from *qsub* and stores them in its own data structure. In my implementation of DAD/IDAD, two variables are added: ATTR_fileUsed for storing file accessed by the job, and ATTR_RDR for storing the RDR of the job.

Modification on pbs_sched

pbs_sched is the scheduling module of Torque. Main modifications are made here. When receiving signals from pbs_server, the pbs_sched will initiate a scheduling cycle, where the latter communicate with the former and get all of the information needed for making scheduling decisions.

Similar to pbs_server, variables are added to store the newly added information for file placement and RDR. This info will be used as the input for DAD and IDAD, which will be described in detail below.

4.3 Workflow of Scheduler Module

As the controlling center of Torque, The pbs_server is responsible for initiating scheduling cycle in pbs_sched by sending some specific signals which are listed below:

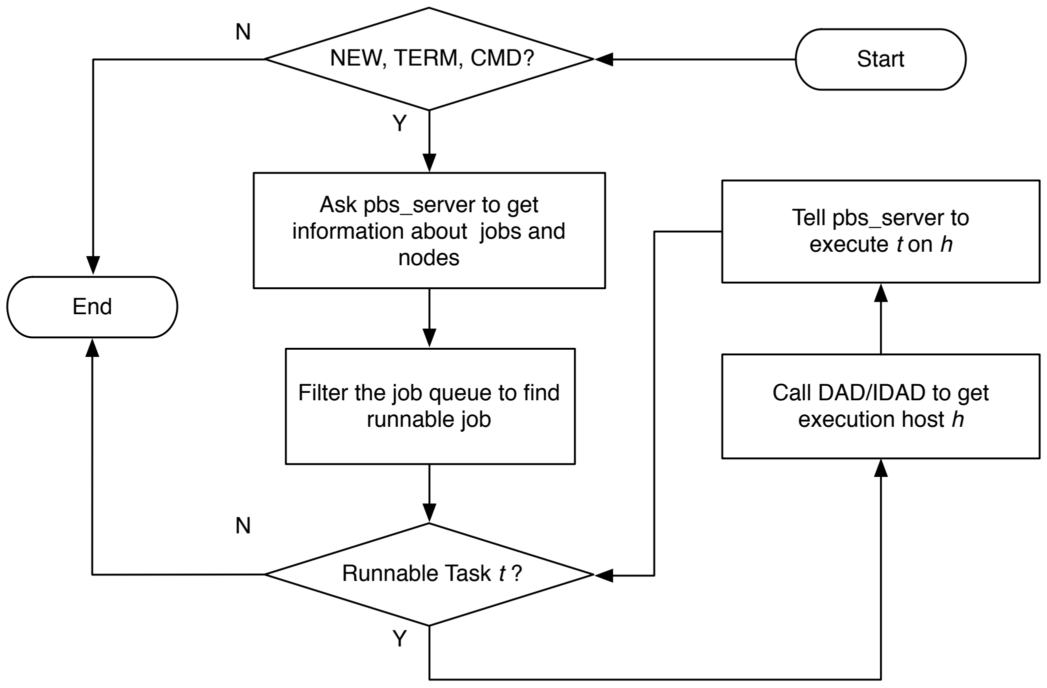

Start
NEW, TERM, CMD?
N
Y
Ask pbs_server to get information about jobs and nodes
End
Filter the job queue to find runnable job
N
Runnable Task *t* ?
Y
Call DAD/IDAD to get execution host *h*
Tell pbs_server to execute *t* on *h*

Figure 4.3: The workflow of Scheduler

1. SCH_SCHEDULE_CMD
   Initiate a scheduling cycle on command
2. SCH_SCHEDULE_NEW
   A new job is queued in pbs_server
3. SCH_SCHEDULE_TERM
   A Running job is terminated
4. SCH_SCHEDULE_FIRST
   First scheduling cycle after pbs_server is started.

In this implementation, when SCH_SCHEDULE_FIRST is received, the scheduler will execute initialization process by allocating memory for hash tables that are used to store information acquired from the metadata server of Gfarm.

When either the CMD, NEW, and TERM is received by the scheduler, it initiates a new scheduling cycle. This process is shown in Figure 4.3. At the beginning, the pbs_sched will get all of the information, jobs' and nodes', from pbs_server, and filter the queue to find runnable jobs. Next, runnable jobs are processed in FCFS manner by DAD or IDAD algorithm. As the output, a selected execution host will be send back to pbs_server.

4.4 Implementation Optimization

Communication Cost

DAD and IDAD need to know the files a job refers to and the nodes where those files reside. The information regarding the replica is managed by the meta-data server of Gfarm, i.e., gfmd. Therefore, the scheduler has to communicate with gfmd before making a decision.

Because only one file can be queried each time, a task refers to many files requires to communicate with gfmd multiple times, which increases communication in the network and is quite time-consuming. We exploit a hash table to store such information and reduce the amount of communication with gfmd. Similarly, the file size is also hashed to avoid redundant communication.

Redundant Wait Time

An issue may arise when naively implementing Delay Scheduling for DAD and IDAD on Gfarm. As stated before, the original Delay Scheduling was designed for the Hadoop file system in which each task can always find a node with all of the access files.

However, in Gfarm, a task can refer to multiple files, and each compute node might hold only a small portion of such files and thus has large *Score*. Therefore, it is possible that neither of the compute nodes satisfies the standard of "local". In this case, waiting for a local node would be a waste of time. DAD will judge whether it is necessary for a job to wait for the next available slot. If a job is not local to either of the compute nodes, it will not be delayed.

4.5 Pseudocode of IDAD

Algorithm 1 Improved Data-Aware Dispatching

```
1: function GETFILESIZE($filePath$)
2:     Communicate with gfmd and get file size of $filePath$, then store the result to hash table;
3: end function
4: function GETFILEEXIST($filePath, nodeName$)
5:     if $eHash[filePath + nodeName]$ exits then
6:         return $eHash[filePath + nodeName]$;
7:     end if
8:     $found \leftarrow false$
9:     $repNode[] \leftarrow$ nodes with replica, from gfmd
10:    for $node$ in $repNode[]$ do
11:        $eHash[filePath + repNode] \leftarrow exists$
12:        if $nodeName = node$ then
13:            $found \leftarrow true$
14:        end if
15:    end for
16:    if $found = false$ then
17:        $eHash[filePath + nodeName] \leftarrow notexists$
18:    end if
19:    return $eHash[filePath + nodeName]$
20: end function
21: function GETREMOTEPORTION($job, nodeName$)
22:    $fileMisMatch, fileTotalSize \leftarrow 0$
23:    for $f$ in $job.fileUsed[]$ do
24:        $size \leftarrow$ GETFILESIZE($f$)
25:        if GETFILEEXIST($f, nodeName$)=FALSE then
26:            $fileMisMatch \leftarrow fileMisMatch + size$
27:        end if
28:        $fileTotalSize \leftarrow fileTotalSize + size$
29:    end for
30:    return $fileMisMatch/fileTotalSize$
31: end function
```

```
function ImpDataAwareDispatch($job$)
    $lShold \leftarrow$ local threshold;
    $wLimit \leftarrow$ max delay time;
    $minScore \leftarrow$ FLOAT_MAX;
    $pNode, gNode \leftarrow NULL$; //possible and good node.
    $ifWait \leftarrow false$; //if need to wait for a good node
    for each execution node $h$ do
        $FileRemoteRate \leftarrow$ getRemoteRate($job, h$);
        $RDR \leftarrow job.RDR$;
        $Score \leftarrow RDR \times FileRemoteRate$;
        if $h$ is not free $\wedge$ $Score < lShold$ then
            $ifWait \leftarrow true$;
        end if
        if $h$ is free $\wedge$ $Score < minScore$ then
            $pNode \leftarrow h$;
            $minScore \leftarrow Score$;
            if $Score < lShold$ then
                $gNode \leftarrow h$;
            end if
        end if
    end for
    if $pNode = NULL$ then
        return $NULL$;
    end if
    if $gNode \neq NULL$ then
        return $gNode$;
    end if
    if $job.wTime < wLimit \wedge ifWait = true$ then
        $job.wTime + +$;
        return $NULL$;
    end if
    return $pNode$;
end function
```

Chapter 5

Performance Evaluation

The performance evaluation of DAD and IDAD will be described in this chapter. There are two parts in this chapter: the evaluation of $Score$ defined by IDAD, and evaluation of schedulers integrated with DAD or IDAD as a whole.

5.1 Test Environment

A cluster in The University of Tsukuba, consists of five nodes named Chris20 to Chris24, is used in this evaluation. Each node has 8GB $\times$ 8 Memory and two NUMA nodes, which adds up to 64GB of Memory and 16 CPU cores with hyper-threading disabled.

In addition, the nodes are connected to a switch (PowerConnect 6248) through Gigabit Ethernet (1000BASE-T full duplex). Infiniband FDR 4x is also available, but it does not match the prerequisite that the network speed is much slower. Therefore it is not used in this experiment.

The storage device used in the experiment are four SAS HDD drives linked to a RAID controller with 1G NVRAM, while RAID is disabled and each drive works as a single device.

More details of software and hardware are listed in Table 5.1.

Table 5.1: Test Enviroment

Hardware	
CPU	Intel Xeon E5-2665 (2.4GHz 20MCache 8Core) x2
HDD	146GB (6Gbps SAS 15,000rpm) x4
Memroy	64GB (8GB 1600MHz) x8
Network	1Gbps Ethernet
Software	
OS	CentOS release 6.8 (Final)
Gfarm	2.7.0
Torque	4.2.6.1

5.2 Evaluation of *Score*

In this section, the evaluation of *Score* defined by IDAD (Equation 3.5 in Chapter 3) is described. This evaluation is quite important because it is crucial to understand the case where a task access multiple files and they are distributed on different nodes.

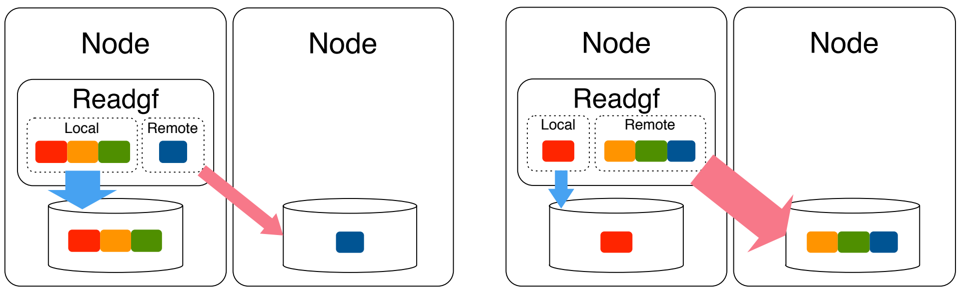
In this section, two benchmarks, Readgf and BLAST benchmark, are utilized to evaluate the effectiveness of *Score*. Readgf is a micro benchmark designed to profile the scheduling process, and BLAST benchmark is a very important set of programs in bioinformatics to finds regions of similarity between biological sequences.

5.2.1 Evaluation of *Score* with Readgf Benchmark

Readgf benchmark is a benchmark designed for profiling the task execution behavior on Gfarm by recording the start time and execution length of each task, and makespan of the entire job. The main workload is reading one or multiple files on Gfarm, which can be specified with the following command:

```
readgf file1 1024 (access size in byte) file2 512 ....
```

In this experiment, one Readgf task that access four files of 512MB are submitted to one compute node, and the number of remote files is altered



(a) Three Local and One Local Files (b) One Local and Three Local Files

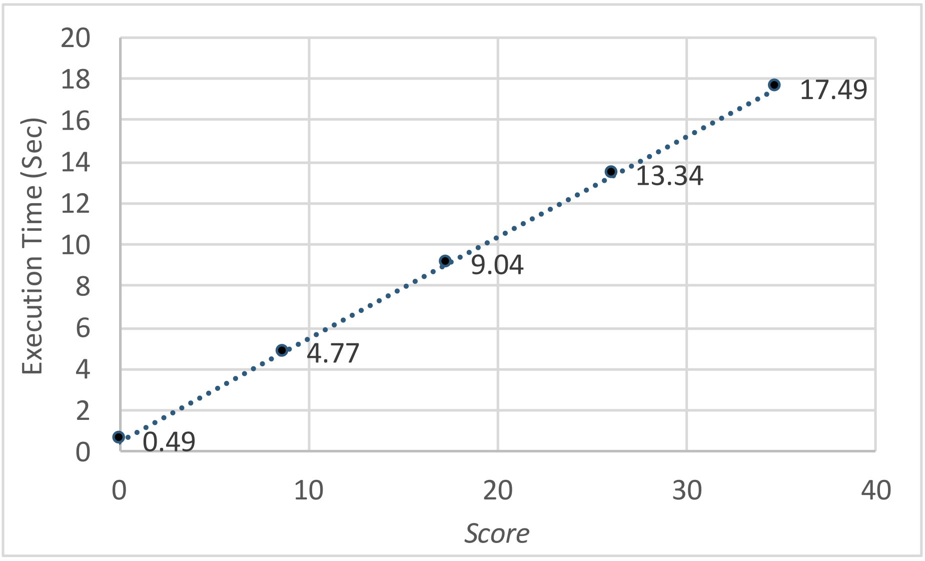
Figure 5.1: Readgf Setting for *Score* Evaluation

from one to four to generate different *Score* and the execution time of each case is recorded accordingly. Figure 5.1a and Figure 5.1b show the situation that one and three files are located in a remote node, respectively. In the figure, the blue and red arrow are local and remote access.

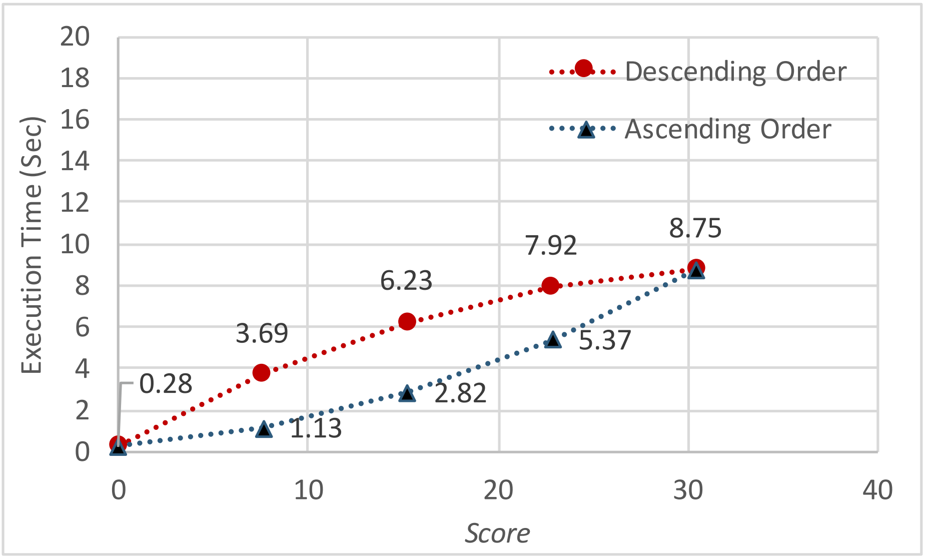
Table 5.2: Calculation of *Score* with Readgf

Remote/Total	Remote/Total Size (MB)	*RemotePortion*	*Score*
0/4	0/2048	0	0
1/4	512/2048	0.25	8.67
2/4	1024/2048	0.5	17.34
3/4	1536/2048	0.75	26.02
4/4	2048/2048	1	34.69

First of all, the *Score* of each case is calculated in Table 5.2. Because four files are same in size, the *RemotePortion* of each case is 0, 0.25, 0.5, 0.75, and 1 respectively. In addition, from the observed data, when all files are located in local, the execution time is 0.49 second. While, when all files are located in remote, the execution time goes up to 14.49 sec. Therefore, according to the definition of RDR (Equation 3.3), the RDR of Readgf task in this test environment is 34.69. In all, *Score*s can be acquired with Equation 3.5 and listed in the Table.



(a) Same Access Size



(b) Different Access Size

Figure 5.2: *Score* Evaluation with Readgf

I plotted the *Score* and execution time in Figure 5.2a. As you can see from the figure, as the *Score* of a compute node increases, the task execution time increases accordingly. These plotted points almost form a straight line, which indicates the increase of execution time is quite proportional to that of

Score. The reason is that this experiment runs in an ideal condition where all four files are same in size, and sizes accessed by Readgf are identical as well.

To evaluate the effectiveness of *Score* when only a part of a file is accessed, I conducted another experiment in which the file size and placement are kept the same with previous setting, but the sizes of accesses are altered. Instead of accessing the whole 512 MB of files, Readgf only accesses a part of them (100/512MB, 200/512MB, 300/512MB, and 400/512MB).

In this experiment, the *RemotePortion*s are the same with previous one because the file size and placement are not changed. However, since the sizes of accesses are altered, the complete remote execution time and complete local execution time have changed. In this case, they are 8.75 and 0.28 second, which means RDR is 30.47.

I ran this experiment two times with different sequence of placing files to a remote node. Firstly, the files are moved to a remote node in ascending order of their access portion size (i.e., the file that 100MB is accessed is moved to the remote node first). In the other setting, the files are moved in descending order of their access portion size.

The result is shown in Figure 5.2b. One key difference when comparing Figure 5.2a and Figure 5.2b is that the maximum execution time decreased from 17.49 to 8.75. This is because in the former case, the total access size is 2048MB (512MB $\times$ 4), while in the latter case, the total access size is 1000MB (100MB + 200MB + 300MB + 400MB).

In Figure 5.2b, the blue and red line indicates the case of ascending and descending order, respectively. Since the file sizes are the same among four files, each relocated remote file increases *RemotePortion* by 0.25, and thus increases *Score* by 7.6175 (0.25 $\times$ 30.47). In the case of ascending order, because the access portion of files relocated to remote is smaller at first, the impact to execution time was not that significant in the beginning, which results in the blue line similar to a convex function. On the contrary, In descending case, the influential files are moved first, which result in the orange line similar to a concave function.

5.2.2 BLAST Benchmark

BLAST

Sequence similarity searching is one of the most important components in bioinformatics. Basic Local Alignment Search Tool (BLAST) is an "approach to rapid sequence comparison, directly approximates alignments that optimize a measure of local similarity" [25]. It finds regions of similarity between biological sequences. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance.

NCBI-BLAST+

BLAST+ [26] is a set of command-line applications to run BLAST. It is released in 2009 and maintained by National Center for Biotechnology Information (NCBI). Different functionalities of BLAST+ are organized by separate programs such as blastn, blastp, tblastn, blastx, tblastx, etc. The main components and their functionalities are summarized below in Table 5.3.

Table 5.3: Some Applications Included in BLAST+

Application	Search Type
blastn	Nucleotide queries to nucleotide databases
blastp	Proteins queries to protein databases
blastx	Translated nucleotide queries to protein databases
megablast	Faster nucleotide queries to nucleotide databases
tblastx	Translated nucleotide queries to Translated nucleotide databases
tblastn	Protein queries to translated nucleotide databases

BLAST benchmark

BLAST benchmark [27] is a benchmarking tool for evaluating the relative performance of BLAST+ running on different hardware and different BLAST implementations. It simulates typical workloads obtained from an analysis of several hundred thousand runs. This benchmark consists of four parts: Databases, Queries, Tasks, and an Executing Shell.

The databases in the current version of BLAST benchmark is "nt.01" and "nr.01", which are a subset of "nt" and "nr" database that accounts for more than 80% of all searches in NCBI. "nt.01" database is a nucleotide database which contains 655,319 sequences and 3,670,101,299 letters in total. On the other hand, "nr.01" database is a protein database which contains 2,509,695 sequences and 856,437,321 letters.

The entire benchmark contains 100 blastn, 40 blastp, 24 blastx, 24 megablast, 10 tblasn, and 10 tblastx tasks. The execution logic of those tasks is described in Makefile. A user can run entire benchmark by executing `make`. Alternatively, one or more specific type(s) of applications can be evaluated separately by specifying their name. For example, `make blastn` only executes blastn.

5.2.3 Evaluation of *Score* with BLAST Benchmark

BLAST benchmark contains multiple sub-benchmarks such as blastn, blastp, and blastx, etc. In this experiment, the blastn have been chosen because it is the most I/O-intensive application, which is ideal to show the effectiveness of *Score*.

A blastn application fires a nucleotide query on a nucleotide database. In the current version of BLAST benchmark, blastn utilizes "nt.01" database included in its package. "nt.01" consists of ten files. They all starts with "nt.01" but have different suffixes, in this study, these suffixes will be used to specify the files. (e.g, nsq stands for database file "nt.01.nsq").

In this experiment, I changed the database's path to an FUSE-mounted Gfarm directory by modifying the Makefile script of BLAST benchmark. Afterward, I recorded the total execution time of all 100 blastn tasks with one of the files in remote, while keeping all other files local. In this way, each *Score* of a node without one particular file can be calculated. The result is shown in Table 5.4. The result is sorted in descending order based on *Score* of nodes.

In this table, the row "nsq" in the means 1) that the *score* of a node without "nt.01.nsq" is 5.2390, and 2) that when all 100 blastn tasks are executed on

Table 5.4: blastn Execution Time on Node with Different *Score*

Remote File	File Size	*Score* of Nodes	Time Difference (Sec)
nsq	878M	5.2390	877.84
nhr	108M	0.6417	12.96
nsd	23M	0.1337	0.00
nhd	11M	0.0656	0.00
nin	7.5M	0.0448	80.80
nnd	5.3M	0.0315	4.77
nog	2.5M	0.0149	0.00
nsi	535K	0.0031	0.01
nhi	256K	0.0015	0.00
nni	22K	0.0001	0.21
(All Remote)	1033M	6.1759	970.44
(All Local)	N/A	0	0

this specific node, the execution time would be 877.84 sec longer than that of an entirely local node. The nsq file is the main sequence database file, which is largest and thus has the largest score.

It is conspicuous that when five files (nsd, nhd, nog, nsi, nhi) are placed to a remote node, the execution time does not seem to be affected. I used `strace` to track the file I/O behavior of a blastn task and found that these files are not opened during the execution. The row ”nin” deserves more than a passing notice because it only has *Score* of 0.0448 but affects the execution time greatly. This is because nt.01.nin is index file of the database, and it will be accessed multiple times during an analysis.

5.2.4 Analysis of *Score* Evaluations

In this section, *Score* defined in IDAD has been evaluated with two benchmarks: Readgf and BLAST benchmark.

In the case of Readgf, the access pattern is close to the ideal condition –the access is equal or proportional to the file size, and all files are accessed only once. This setting makes *Score* quite accurate in representing the I/O load of a task.

On the other hand, in the case of BLAST benchmark, a mixed-load real-world application is utilized. Since one of the referenced files is quite large, the *Score* will not be affected by other files greatly. Moreover, blastn ' s accesses to each file are imbalanced that some files are frequently accessed, while other files are not even accessed a single time. The access sizes also heavily depend on different queries. These facts make *Score* not that accurate to represent the I/O loads sometimes.

Though not as accurate in Readgf benchmark, however, the *Score* mostly reflects the I/O load in for most files, because it is fair to assume the largest files specified by a user will likely to be accessed and thus affect the execution time. If it is not the case, a user can just omit that file to get more accurate *Score*.

Introducing per-file weight may be a solution for this issue. Allowing users to specify the importance of each file individually may help scheduler make better decision indeed. However, It would significantly increase the user cost. Users from other disciplines may have difficulties in analyzing the access pattern of their own applications. It is a trade-off between accuracy and usability.

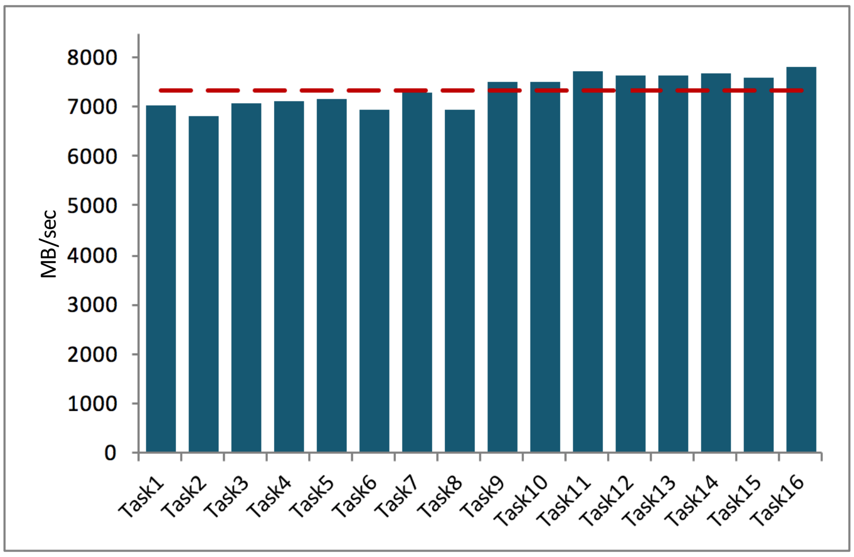
In all, under current constraints, I conclude that *Score* defined by IDAD is effective to express the I/O cost of a task allocation.

5.3 Evaluation of Data-Aware Scheduling

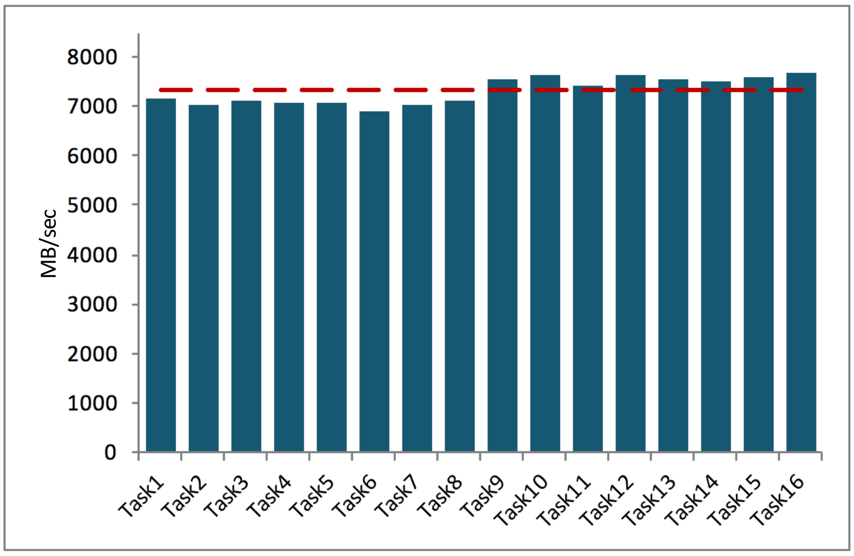
In this section, a Torque scheduler integrated with DAD and IDAD module will be evaluated. It will be compared with stock Torque scheduler using three benchmarks: thput-gfpio, Readgf, and BLAST benchmark.

5.3.1 thput-gfpio

The thput-gfpio benchmark is included in the Gfarm file system package and is used for evaluating the read, write, and copy performance of Gfarm. The block size and access size can be specified. This benchmark is utilized to show the I/O throughput gain made by two data-aware algorithms.



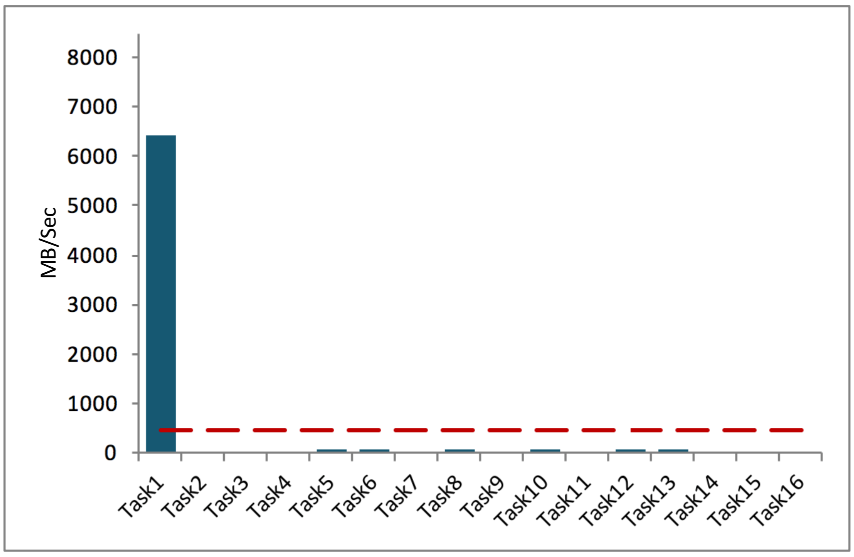
(a) Data-Aware Dispatch



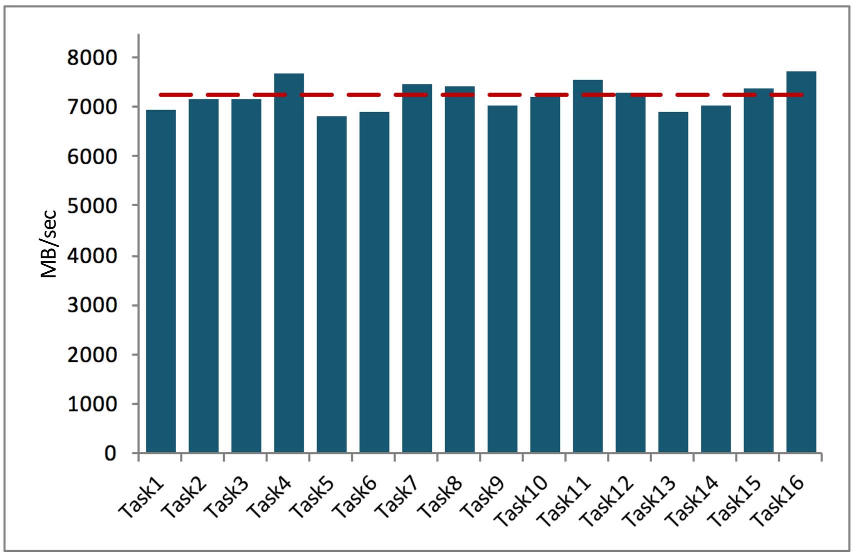
(b) Improved Data-Aware Dispatch

Figure 5.3: Evaluation result using thput-gfpio (Data-Aware)

In this experiment, 16 different 1 GB files were generated beforehand and distributed evenly to four compute nodes. In addition, 16 thput-gfpio tasks were submitted to read each file accordingly.



(a) Stock Torque



(b) Stock Torque with task order rearranged

Figure 5.4: Evaluation result of thput-gfpio (Stock Torque)

The results of two data aware approaches are shown in Figure 5.3. Figure 5.3a and Figure 5.3b are the results of DAD and IDAD, respectively. In these figures, each bar represents the read throughput of a task, and the red line is the average throughput of all 16 tasks. As you can see in two figures, the results of two Data-Aware approaches have quite a similar result. This is because the workload of this benchmark is mainly reading files on Gfarm file system, while DAD and IDAD are quite similar in handling I/O-only workload. In summary, the average throughput of DAD and IDAD are 7340.88MB/s and 7319.74MB/s respectively.

On the other hand, two results of stock Torque scheduler are shown in Figure 5.4. In the first case (Figure 5.4a), only one task has relative high read throughput, while in the other case (Figure 5.4a), the stock Toque scheduler shows a similar result with two Data-Aware approaches. The reason for this huge difference is explained blow:

As mentioned above, 16 files are made before the evaluation and distributed evenly to four compute node. For ease of description, we name the 16 tasks Task1 to Task16, which access 16 files named file01 to file16. In addition, we assume the first node has file01 to file04, the second node has file05 to file08, and so on. The setting is listed in Table 5.5.

Table 5.5: Evaluation Setting of thput-gfpio

Task	File Accessed	Location	Task	File Accessed	Location
Task01	file01	node1	Task05	file05	node2
Task02	file02		Task06	file06	
Task03	file03		Task07	file07	
Task04	file04		Task08	file08	
Task09	file09	node3	Task13	file13	node4
Task10	file10		Task14	file14	
Task11	file11		Task15	file15	
Task12	file12		Task16	file16	

In the first case where throughput degraded greatly, tasks are submitted in alphabetical order to read file1 to file16 accordingly (Task01, Task02, ..., Task16). Since the stock scheduler is not aware of data placement, Task1 to Task4 will be executed on four different nodes first (Task01 on node1, Task02 on node2, Task03 on node3, Task04 on node4). i.e., four tasks that access files on the same node will be executed at the same time. In this case, only one task can access the file locally (Task01 on node1). In addition, the network bandwidth of the node with all four files (node1) will be shared by other three tasks, which will significantly reduce the performance of both local (Task01) and remote accesses (Task02, Task03, Task04). Hence, you may notice that the performance of the only local task is also lower than that in two Data-Aware approaches.

In the second case, tasks are submitted in a way that all four concurrently running tasks are able to access file distributed in all four nodes (Task1, Task5, Task9, Task13,..., Task4, Task8, Task12, Task16). When first four task (Task1, Task5, Task9, Task13) are being executed, all tasks can perform local access because that files accessed (file1, file5, file9, file13) are distributed in four nodes.

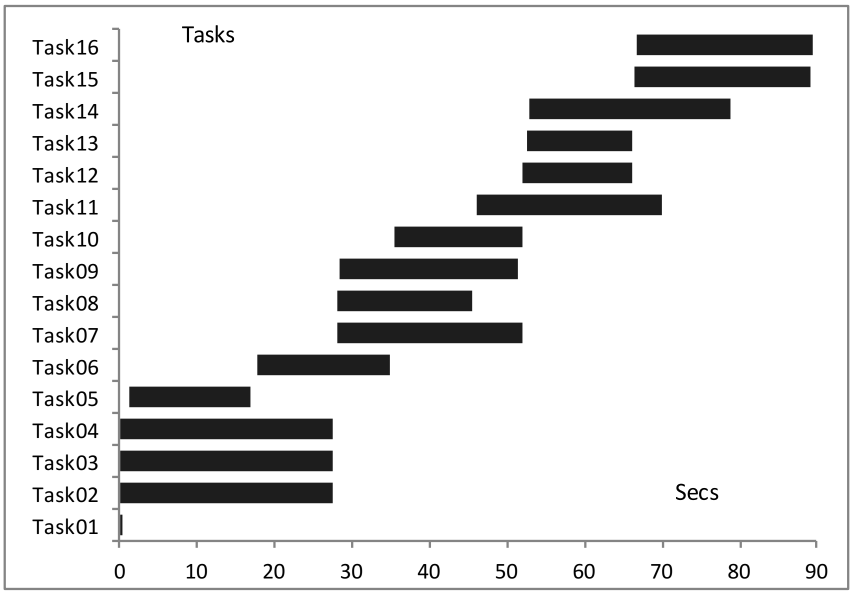
The result in Figure 5.4b shows that with the proper task sequence, the stock Torque scheduler may perform as well as two data-aware approaches. However, such rearrangement of tasks is only practical in a simple test case. On the other hand, two properly configured data-aware approaches are able to perform well regardless of the task sequence.

5.3.2 Readgf

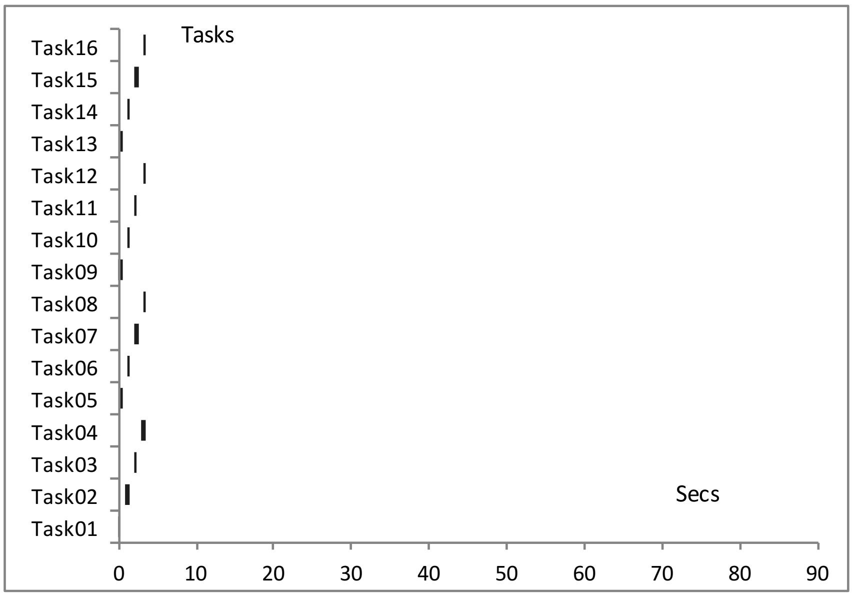
An increase in throughput performance can be clearly observed in the previous benchmark. However, the behavior of the tasks remained unknown. The Readgf benchmark was developed to reveal detailed task scheduling information such as the start time, execution length, and makespan. The main workload of Readgf is reading files on Gfarm.

The file placement setting in this evaluation is identical to that of thput-gfpio benchmark: 16 different 1 GB files were made beforehand and distributed evenly to four compute nodes. In addition, 16 Readgf tasks were submitted to read each file accordingly. The results are shown in Figure 5.5.

The two subfigures in Figure 5.5 are Gantt charts. In a Gantt chart, the names of tasks are listed on the left of the y-axis and the x-axis is the time elapsed since the start of the first task. Each task is represented by a bar; the position and length of the bar reflect the start time, duration and end time of the task. A longer bar means more time is consumed during execution, and thus less effective. On the other hand, shorter bar means less time, and thus more effective.



(a) Stock Torque Scheduler



(b) Data-Aware Dispatch

Figure 5.5: Evaluation Results of Readgf

Since the workload of Readgf benchmark is reading files on Gfarm file system, the DAD and IDAD work in a similar way. Therefore, the Gantt chart of IDAD is omitted here.

As shown in Figure 5.5a, using the stock Torque scheduler, only one of the tasks has a significantly short bar, which means that the interval between the start and end time was short and that the task was read efficiently from the local node. All other tasks, on the other hand, have relatively long bars and are less effective. Conversely, using DAD, as shown in Figure 5.5b, all of the tasks had short lines, indicating that they performed efficient local accesses.

Another noticeable difference between stock Torque scheduler and DAD is that the order of task execution is altered. X-axis in a Gantt chart is the timing axis, therefore, the task sequence can be acquired by finding the left side of bars, from left to right.

The stock Torque scheduler does not alter the task sequence, the starting point (left side of bars) of tasks appears in exact same order to the sequence of task submission. On the contrary, the DAD automatically altered the execution order of tasks by applying delay scheduling, which guarantees local access for tasks.

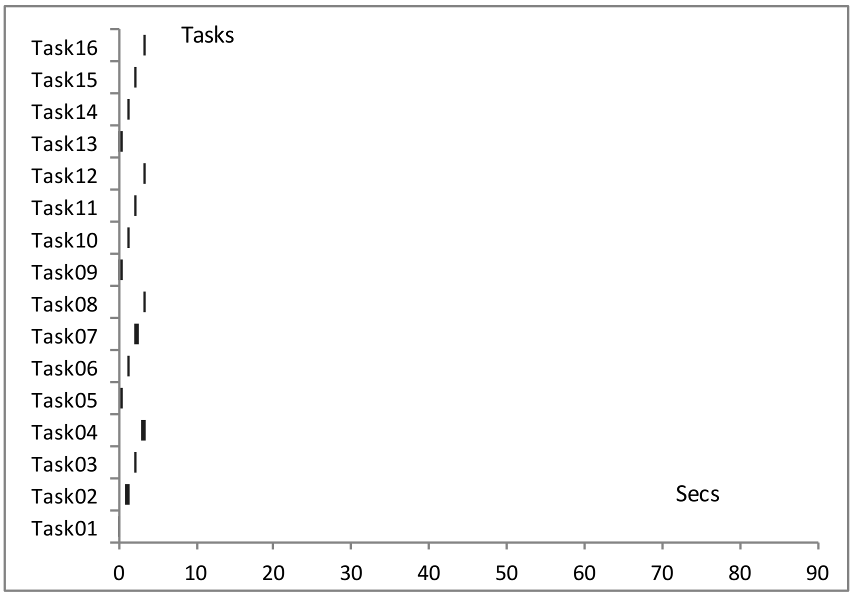


Figure 5.6: Gantt Chart of Readgf with Re-ordered Task Sequence

I also tried to alter the task sequence manually, as I did in thput-gfpio, so that each task can perform the local access. The result is shown in Figure 5.6. In this figure, the task order is changed to Task1, Task5, Task9, Task13..., which has identical execution order with DAD. Therefore, the result is similar to that of DAD.

5.3.3 BLAST Benchmark

I evaluated DAD and IDAD with BLAST benchmarks, comparing it with stock Torque scheduler. BLAST benchmark has a single Makefile responsible for controlling the task execution. However, such setting is not ideal for executing BLAST benchmarks in parallel. Therefore I modified the way of executing tasks of BLAST benchmark so that it will fit in a batch scheduler.

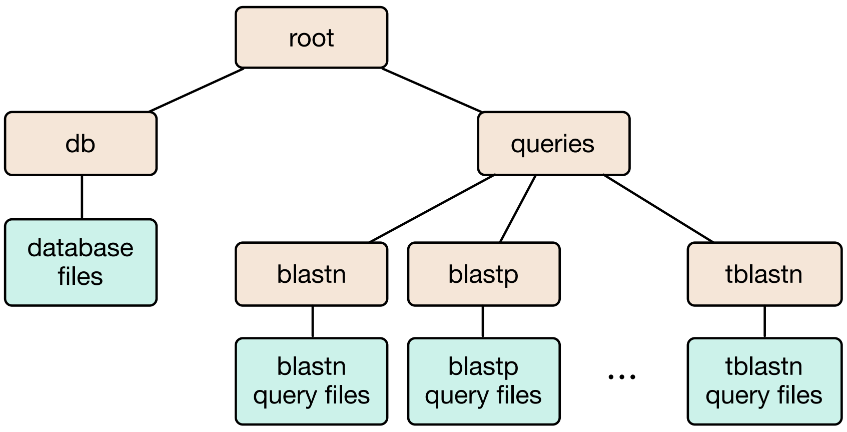


Figure 5.7: Directory of BLAST Benchmark

Submitting BLAST Benchmark to Torque

In the original setting of BLAST benchmark, a Makefile is used to control the task execution. It only provides parallelism on single node level by executing `make -j8`, where the `-j8` indicates that eight processes will be executed in parallel. Obviously, it will not fit in a cluster with multiple nodes.

The Makefile consists of seven sections for each BLAST application: tblastx, tblastn, blastx, blastn, blastp, megablast, and idx_megablast. A typical BLAST application searches databases using a query, which has the following format:

```
  blastn -db DB_FILE -query QUERY_FILE
```

where the blastn is the application, which can be blastn, blastx, tblastx, blastp, etc. DB_FILE is the database required for a query, whereas QUERY_FILE describes the content of a query.

I changed the script so that the tasks in BLAST benchmark can be submitted to Torque scheduler. I will use an example to explain the script. A script to submit `blastn` is listed as follow:

```
#!/bin/bash
#PBS -N BLAST
#PBS -q blastn
#PBS -g /db/nt.01.nhr,/db/nt.01.nni,/db/nt.01.nnd,
/db/nt.01.nsq,/db/nt.01.nin
#PBS -t 1-100
#PBS -B 6.18
file=`ls /home/risyomei/blastbenchmark/queries/blastn/
         | head -n $PBS_ARRAYID | tail -n 1`

echo `date +%s%N`:starttime $file BLASTN$PBS_ARRAYID
  `hostname --short`

blastn -db /work/risyomei/mnt/db/nt.01 -query
  /home/risyomei/blastbenchmark/queries/blastn/$file

echo `date +%s%N`:endtime  $file
```

The directive “ #PBS ” indicates the parameter will be sent to Torque scheduler. The parameters used in this script is listed in Table 5.6.

"-N" specifies the name of the job, which will be displayed in the scheduler and used to rename the output. "-q" specifies the queue this task is sent to. "-g" is parameter added in this study, which specifies the files accessed by the task and will be used to calculate the *fileLocality* in DAD and RemotePortion in IDAD. "-B" is for RemoteDegradationRate, which is a per-task parameter to show the extent to which a job is data-intensive. Finally, the "-t" stands for the number of subtasks of job array.

Job arrays in Torque scheduler are an easy way to submit multiple similar jobs. in the script, “-t 1-100” indicates that 100 sub-tasks will be submitted through this script, which corresponds the number of blastn tasks in BLAST benchmark. Each sub-task submitted through job array will be assigned a unique $PBS_ARRAYID among all subtasks.

Table 5.6: Parameters Used in Script

Parameter	Description
-N	The name of job
-q	Execution queue the task is sent to
-g	Files accessed by the task
-B	RemoteDegradationRate
-t	Array job numbers

In the script, PBS_ARRAYID is used to specify query file in the sub-task. The following command finds the right query files in query directory: the n *th* sub-task will execute the n *th* query in the right directory.

```
file=`ls /home/risyomei/blastbenchmark/queries/blastn/
         | head -n $PBS_ARRAYID | tail -n 1`
```

In addition, the “date” and “hostname” command are used to output the start and end time of the task, and the execution host of the task, respectively. This information will eventually be used to plot a Gantt chart.

The different BLAST applications have quite different characteristics. Therefore, I first executed different applications remotely and locally to see the time difference between local and remote execution. This information is summarized in Table 5.7.

I have chosen blastn and blastx for sub-benchmark because they refer to different databases and are I/O-intensive jobs. Furthermore, the extent of CPU intensity differs across the two sub-benchmarks.

Table 5.7: Pre-Evaluation of BLAST Benchmark

Application	Data Base	Num.	Remote Time	Local Time	RDR
blastn	nt.01	100	1130.42	157.53	6.18
blastp	nr.01	40	1204.08	797.46	0.51
blastx	nr.01	24	926.70	746.52	0.24
megablast	nt.01	24	537.58	290.78	0.85
tblastn	nt.01	10	530.49	471.03	0.13
tblastx	nt.01	10	2851.66	2780.74	0.03

Evaluation of DAD with BLAST Benchmark

In this evaluation, two types of databases are replicated two times and distributed to four nodes as shown in Figure 5.8. 100 blastn tasks and 10 blastx task are sent to these nodes for execution. During the submission, tasks of blastn and blastx are submitted to two different queues to avoid excess judgment regarding the DS.

Since there is no viable way of calculating the optimal β at the moment, I have evaluated DAD with different values of β and *Delay* to acquire the best result.

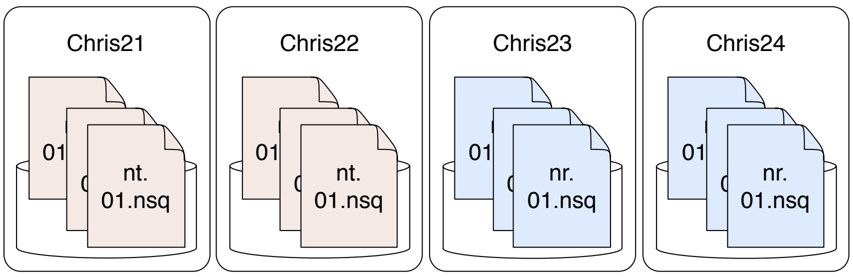


Figure 5.8: BLAST Benchmark Setting

The results are shown in Figure 5.9. As you can see from the figure, when $\beta < 0.6$ or the $Delay < 1$, the performance of DAD degrades significantly. This is because when β is smaller than 0.6, even the $fileLocality$ calculated from file placement is 1, The *Score* will be smaller than the default local threshold defined in DAD, which disables Delay Scheduling. Similarly, if the

Delay = 0, the Delay Scheduling will not work. Moreover, you may also notice that when the delay is larger than two, there will be no significant performance improvement.

In all, the best case for the DAD scheduler ($\beta = 0.8$, with a *Delay* of 1, as indicated by the dark bar in Figure 5.9) had a total execution time of 169.039 sec, whereas the total execution time using the stock Torque scheduler was 241.93 sec. Hence, the DAD reduced the makespan by 30.13% in comparison with stock Torque scheduler.

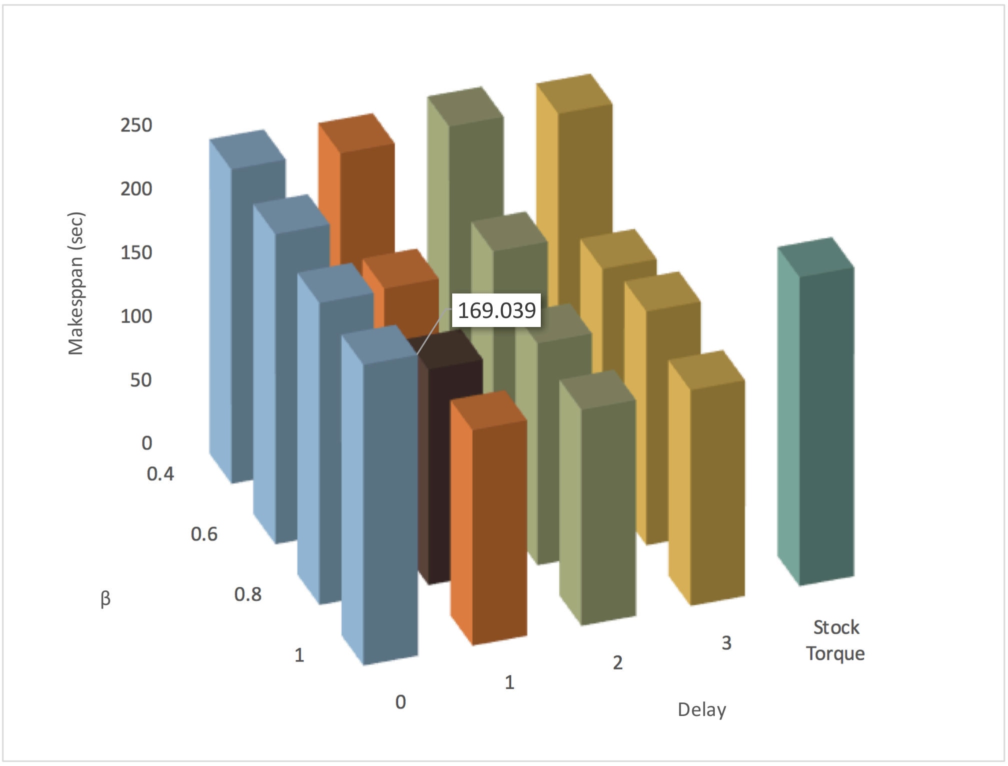
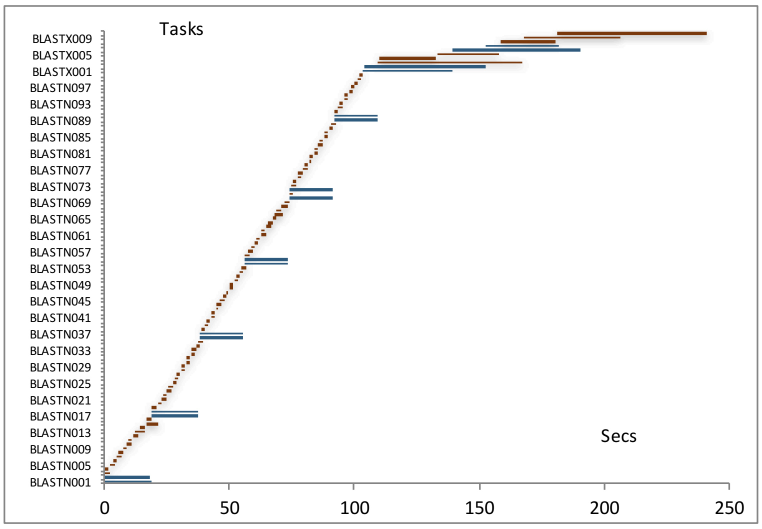


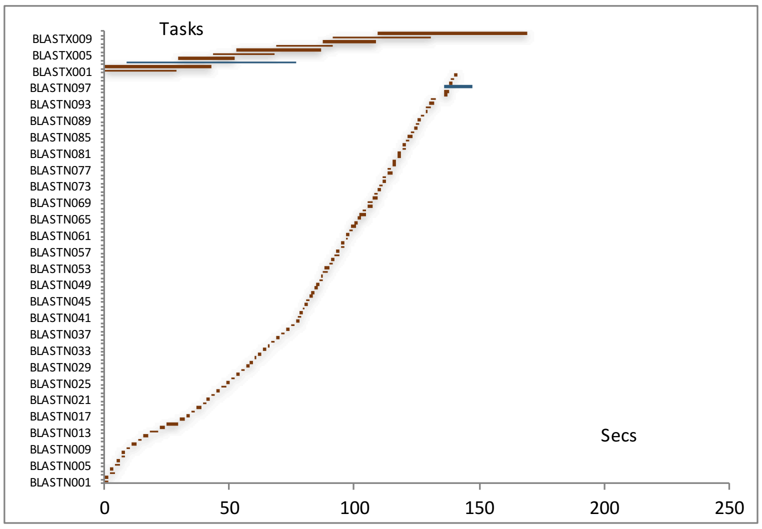
Figure 5.9: Makespan Comparison of DAD with BLAST Benchmark

The Gantt chart of the stock Torque scheduler and the best case for DAD are shown in Figure 5.10. Figure 5.10a is the result of stock Torque scheduler and Figure 5.10b is the best case of DAD with $\beta = 0.8$, *Delay* = 1. In the figure, the red and blue bar represent the task has performed local and remote access respectively.

One obvious difference is that there is only one clear line for the Torque



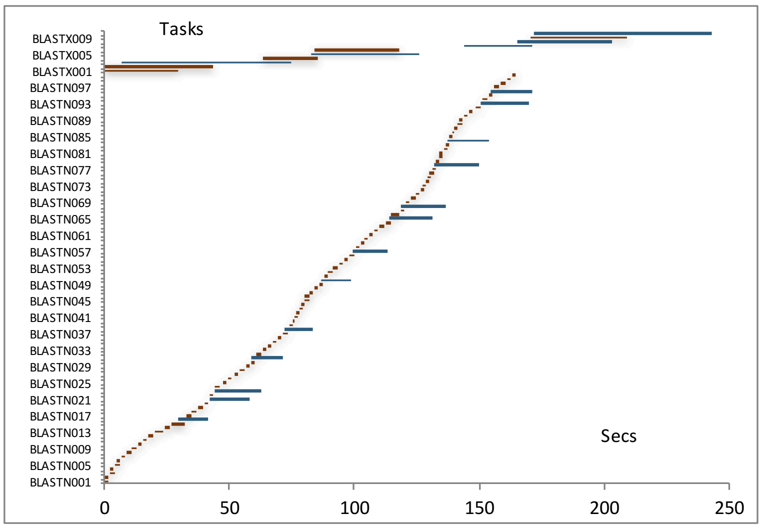
(a) Stock Torque Scheduler



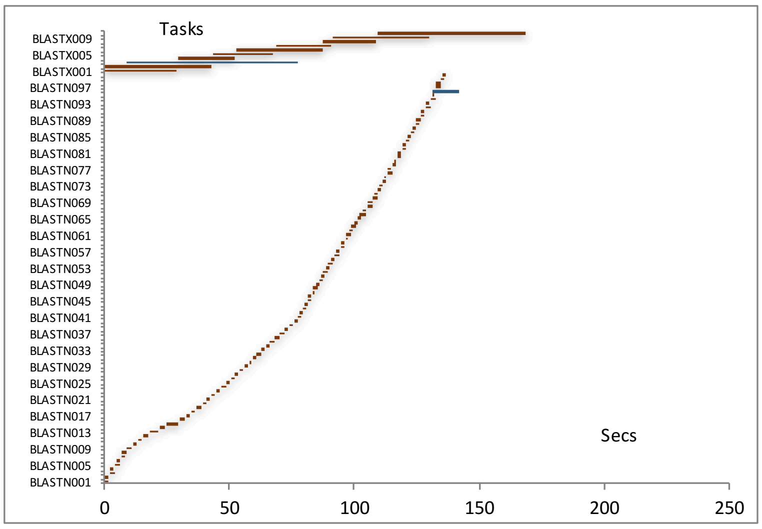
(b) Data-Aware Dispatch ($\beta = 0.8$, $Delay = 1$)

Figure 5.10: Gantt Chart of Stock Torque and DAD

scheduler, but two for DAD. This is because the stock Torque scheduler sees two queues as one large queue, whereas DAD dispatches the task in each queue once each time (top, blastx; bottom, blastn). Moreover, by counting the blue bars in two graphs, the number of remote tasks with stock Torque scheduler is 16, whereas that number is 2 with the best case of Data-Aware Dispatch, which means the effectiveness of the file access is improved.



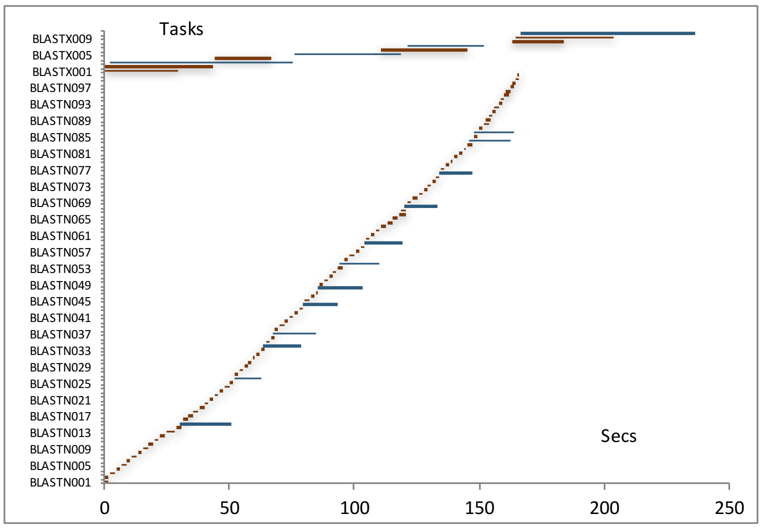
(a) $\beta = 0.4$ and $Delay = 1$



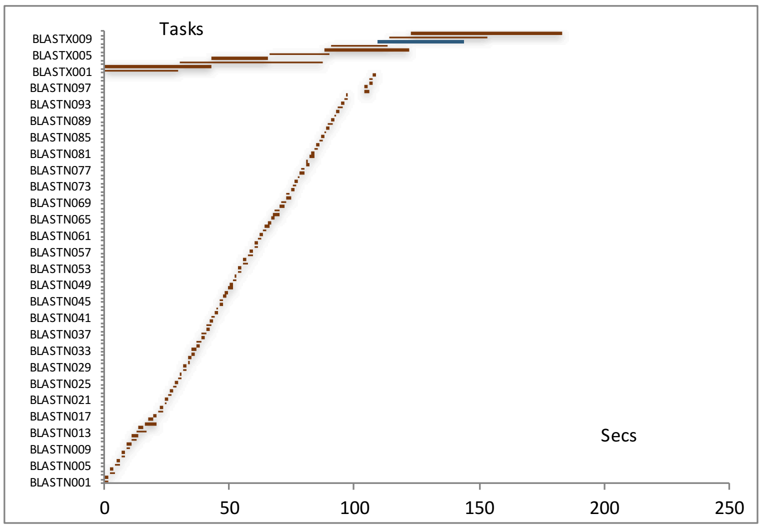
(b) $\beta = 1$ and $Delay = 1$

Figure 5.11: Gantt Chart of DAD ($Delay$ fixed)

To see how β and $Delay$ affect DAD, I fixed one parameter and changed another. Firstly, I fixed the $Delay$ to 1 and changed β to 0.4 and 1.0, corresponding to the Figure 5.11a and Figure 5.11b, respectively. When the β is set to 0.4, consider that the CPU load will not be high because only one task is executed on each node, even the $fileLocality$ is its maximum value–1, the $Score$ should still be smaller than the predefined $lThreshold = 0.5$.



(a) $\beta = 0.8$ and $Delay = 0$



(b) $\beta = 0.8$ and $Delay = 3$

Figure 5.12: Gantt Chart of DAD (β fixed)

Therefore, Delay Scheduling will unlikely to be initiated in this case, where DAD finds the best node among available ones, but does not try to ensure it is suitable for execution, and thus lead to bad performance.

On the other hand, when β is set to 1, the DAD shows a similar performance as the best case of DAD. From the comparison between Figure 5.10b and Figure 5.11b, the task order is identical between these two cases. This is mainly because the CPU load is not high enough influence the result because only one task is executed on each node.

Next, when β is fixed to 0.8 and *Delay* is changed to zero and 3, the results correspond to Figure 5.12a and Figure 5.12b, respectively. When *Delay* is set to zero, DAD does not delay a task when no local nodes are available, thereby incurring a long line on the chart and ending up with a longer total execution time.

On the other hand, when *Delay* is set to 3, tasks have a greater chance to be executed locally. As you can see from the graph, all of the blastn tasks are executed locally. However, in this case, after blastn tasks finishes, the remaining blastx tasks have no choice but run on remote nodes. In contrast, in the best case, some longer tasks of blastx is executed earlier, resulting in a shorter makespan.

Evaluation of IDAD with BLAST Benchmark

In the evaluation of IDAD with BLAST benchmark, I kept the setting same with the previous benchmark. Therefore, the results from two evaluations are comparable.

The blastn and blastx are used for this benchmark, and the file placement of Database file is same with evaluation in DAD, which is shown in Figure 5.8. Moreover, tasks of blastn and blastx are also submitted to two different queues to avoid excess scheduling judgment.

According to Table 5.7, the RDRs of blastn and blastx are 6.18 and 0.24, respectively. The RDR of blastn is quite large, which means the time difference when blastn application is executed on remote and local node is significant. On the other hand, the RDR of blastx is only 0.24, which is much smaller when compared with its counterpart. Hence, in this evaluation, *lThreshold* (local threshold) is set to 0.3 because I want to classify blastx as a CPU-intensive job and blastn as an I/O-intensive job. In the following section, *lThreshold* will be set to a large and then a small value to see how it affects the result.

The makespans of IDAD with different *Delay* and the stock Torque scheduler are shown in Figure 5.13. Since the evaluation of DAD and IDAD share

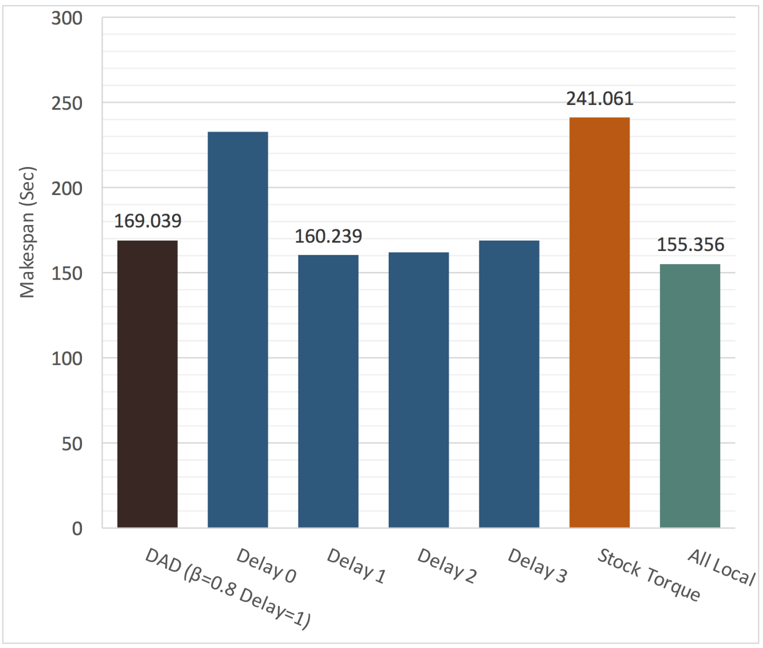
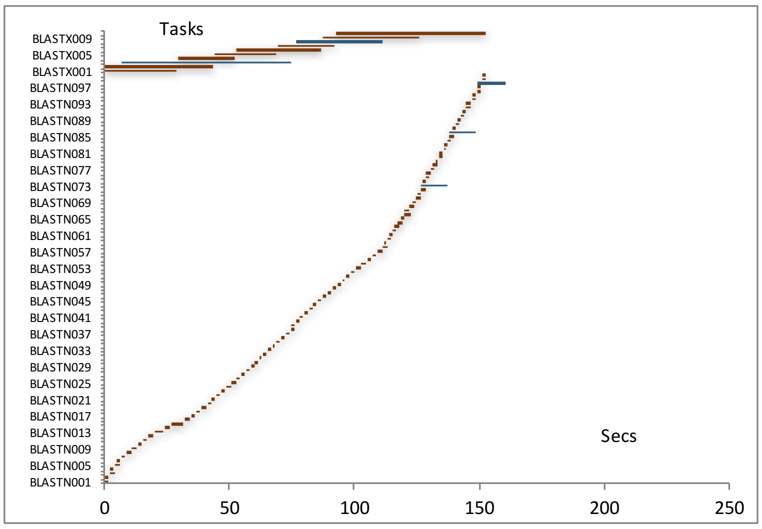


Figure 5.13: Makespan Comparison of IDAD with BLAST Benchmark

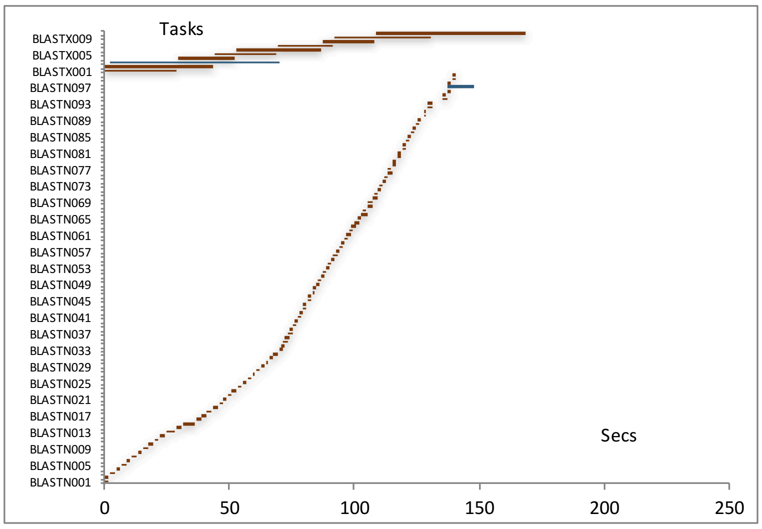
the same setting, the result of the best case of DAD ($\beta = 0.8$, $Delay = 1$) is also plotted in this figure. "All Local" in the figure is obtained under the condition that all nodes hold all datasets required by the task, which means whichever node the task is dispatched to, it will access files locally. Therefore, it can be considered the lower bound for the evaluation. This lower bound is not always reachable, since there simply may not be enough local nodes for tasks.

In Figure 5.13, the best makespan using IDAD is 160.239 sec ($Delay = 1$), whereas the stock Torque scheduler's makespan is 241.061, which is a 33.53% time reduction. When the $Delay = 0$, the IDAD performs as bad as stock Torque scheduler because a task will be dispatched to a node with the lowest $Score$, regardless of access cost. The makespan decreases significantly when $Delay$ is greater than zero. Interestingly, as the $Delay$ increases from 1, the makespan increases slightly. To clarify this situation, I plotted the Gantt chart of IDAD ($Delay = 1$) and IDAD ($Delay = 3$) in Figure 5.14.

The Figure 5.14a and Figure 5.14b may look quite similar at first glance. However, the number of remote tasks (i.e. blue bars) in these two cases are



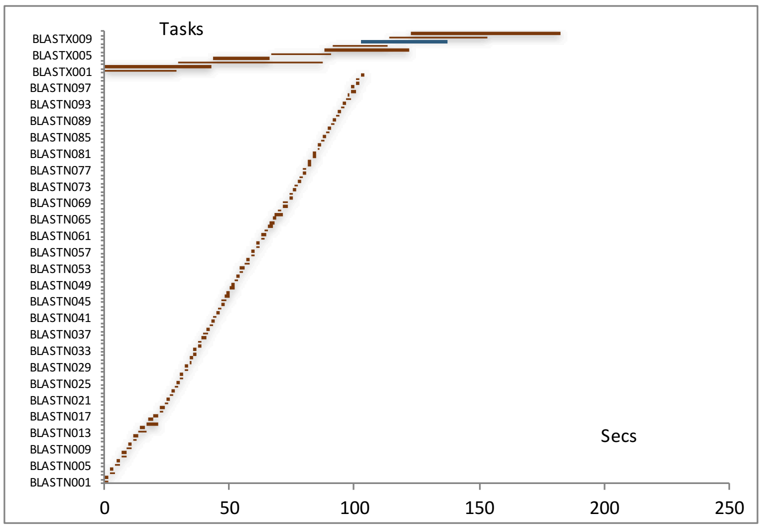
(a) IDAD ($Delay = 1$)



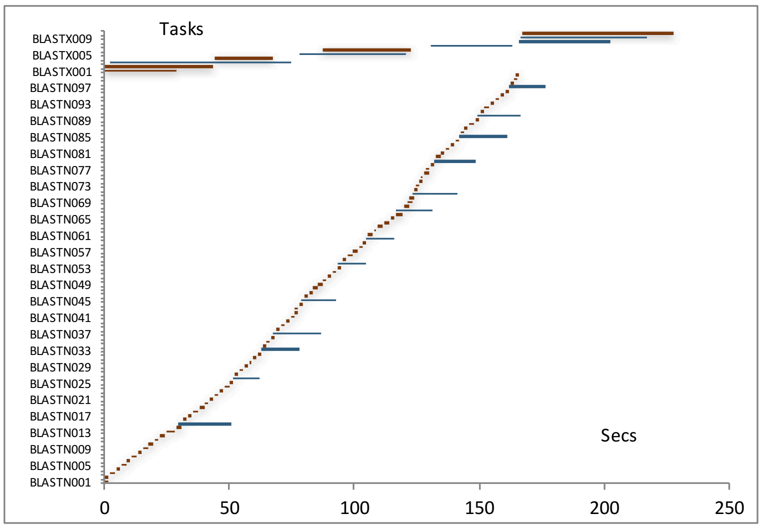
(b) IDAD ($Delay = 3$)

Figure 5.14: Gantt Chart of DAD ($Delay = 1$ vs $Delay = 3$)

different. Result of IDAD ($Delay = 3$) has significantly smaller number of remote tasks because larger $Delay$ value allows the task to wait for more free slots when no local slot available, increasing its possibility of local execution. In this test case, a significantly long blastx task is likely to finish at last, which elongates the makespan. Therefore, it is ideal to start that long blastx task earlier. The IDAD ($Delay = 1$) are not that strict in enforcing locality, which



(a) $lThreshold = 0.1$ and $Delay = 1$



(b) $lThreshold = 7$ and $Delay = 1$

Figure 5.15: Gantt Chart of IDAD ($lThreshold = 7$ and $lThreshold = 0.1$)

allows some of the blastx task start and end earlier. Hence, the influential long task can be started and ended earlier, which leads to shorter makespan.

The situation is quite similar in the comparison between the best case of DAD (Figure 5.10b) and IDAD (Figure 5.14a). In the best case of DAD, the number of remote tasks is two, which is smaller than five of IDAD. However, the IDAD still has shorter makespan, because it executed the last long task of blastx earlier.

In this evaluation, I set $lThreshold = 0.3$, which means blastx ($RDR = 0.24$) tasks will be seen as a local task even if all of its files reside on remote nodes because $RDR < lThreshold$, and are thus more likely to be executed than blastn ($RDR = 6.18$) tasks. Therefore, the final long task can finish earlier, which leads to the reduction of makespan.

Next, I evaluated the effect of $lThreshold$ by altering this value to 0.1 and 7 while keep other parameter settings. When $lThreshold = 0.1$, only local nodes will be selected as execution nodes. On the contrary, if $lThreshold = 7$, all nodes will be regarded as local nodes for both blastx and blastn.

The result of this evaluation is shown in Figure 5.15. When $lThreshold = 0.1$, almost all tasks in Figure 5.15a are executed locally. There are only one remote task in this case, which means strong locality is enforced by the scheduler. However, with all of the blastn task finished locally, the starting time of trailing blastx tasks are postponed, which increased the makespan.

On the other hand, when $lThreshold = 7$, all nodes will be regarded as local nodes. Therefore, when a task can not find a local node, it will be executed remotely. This situation is quite similar to DAD or IDAD with $Delay = 0$. This similarity can be confirmed by comparing the Figure 5.15b (IDAD, $lThreshold = 7$) and Figure 5.12a. In this case, there are many remote tasks whose performance is not that efficient, which will impact the makespan.

Chapter 6

Related Work

Scheduling algorithms have been widely studied from various perspectives. In this section, I focus on those works emphasizing file allocation. In the following section, works on Hadoop/MapReduce and LSF Plugin are summarized.

6.1 Schedulers for Hadoop/MapReduce

Hadoop/MapReduce has completely different computation model with the one in batch queuing systems. In MapReduce, a task can access files on a local worker, as well as its counterpart on a remote node, which makes it an NUSA-based system despite the huge difference in computational models. The default scheduler for Hadoop basically works as an FCFS scheduler.

The YARN scheduler is an upgraded version of the resource manager in Hadoop2, and it recognizes three levels of locality: Node, Rack, and Off-switch.

There are many works that aim to improve locality in Hadoop. The Workload Characteristic Oriented Scheduler [28] introduced the concept of Compute Rate (CR) to denote the extent of CPU intensity of a job. It samples some tasks from a job to acquire their CR and then adjusts its dispatch strategy to improve job locality. [29] introduced the LART scheduler to collocate reduce tasks with the maximum required data, and [30] proposed a sampling-based approach to minimize transmission cost and maximize locality for reduce tasks.

X. Wang et al. presented a locality and energy-aware scheduling method [31] that takes advantage of file locality. They defined a method to calculate energy efficiency and tried to strike a balance between efficiency and locality. The main purpose of this work is to control energy consumption, which is quite different from my goal. Furthermore, this approach is based on the assumption that each file has a fixed number of replicas. This is a basic characteristic of MapReduce but does not hold true for other NUSA file systems like Gfarm.

Delay Scheduling was designed to tackle the conflict between locality and fairness. In DS, when a task should be running according to fairness policy but fails to find a local slot, it waits for some local slots before it is executed remotely. The authors argue that tasks are likely to run locally with little compromise in fairness.

However, DS cannot be applied directly to an arbitrary system because it is based on two assumptions: 1) each job has a nearly identical execution time and will finish relatively quickly and 2) there will always be a local node with all files that the job requires. Both assumptions might fail in other NUSA file systems because the execution time of a task is highly unpredictable in a batch queuing system and a task may access multiple data sets on different nodes.

6.2 Data-Aware Scheduler for Grid and Distributed File System

In [32] and [33], Stork scheduler, a data-aware scheduler for a grid, was proposed. Stork scheduler aims at moving data between file systems, but this work is designed to manage data inside a namespace. W.Tang el [34] proposed an approach to quantify data characteristics by calculating the compute-to-I/O ratio. They argue that a task is more *computationally expensive* if it has the same I/O size but a longer execution time. Based on this value, two strategies (Best-fit or Greedy) are given.

MATRIX [35, 36] is a distributed scheduler that consists of multiple nodes;

each node has an executor, a scheduler, and a metadata controller. MATRIX classifies tasks into local and stealing-ready jobs based on the bandwidth a task use if remotely executed. Each scheduler can steal stealing-ready jobs from the queues of other schedulers.

In [37], a scheduling method for balancing workload when considering locality, network state, and the current workload is presented. In this approach, files are divided into multiple blocks of the same size and distributed and replicated across the nodes. Because each block size is identical, the execution time and load impact of local and remote tasks can be determined. The estimated execution time and load are then used to balance the workload. Because certain NUSA file systems, such as Gfarm, do not divide files into blocks, file sizes may differ from each other widely, and the execution time of each task can therefore not be predicted prior to the execution, which makes this an inapplicable approach.

6.3 Data-Aware Scheduling LSF Plugin for Gfarm

The methods in [38] and [39] are the most relevant approaches that we are currently aware of. The authors proposed two approaches to optimize the creation of a replica: 1) a method for selecting the best node to create the replica when considering the source, destination, and network loads, and 2) a method for categorizing jobs to ensure that the time and performance will not be wasted when creating the replica.

This work and theirs share some characteristics: 1) they are based on Gfarm and take advantage of its effective local file access and 2) they use resource managers to implement the approach. However, their work emphasizes the manipulation of replicas. Tasks are always dispatched to compute nodes with the required file, and the workload is distributed by creating a replica on a new host (such that a task can be dispatched to it).

The main defects of their work are that it fails to handle the case in which a task references multiple files and those files are distributed in different nodes. Both DAD and IDAD offer a method for handling this case. Moreover, the

LSF Gfarm plugin lacks the ability to deal with a "fake" data-intensive job, which refers to a large dataset of which only a small part of it is accessed. For example, both blastx and blastp refer to the same database, but blastx is much more data-intensive. In IDAD, we use RDR to resolve this issue.

Chapter 7

Conclusion

7.1 Summary

As the granularity of simulations increases, the demand for dealing larger datasets is growing accordingly. Handling such databases with Uniform storage access files requires high-throughput networking systems, which incurs high cost in a large environment. Non-uniform storage access (NUSA) file systems federate the local storages of compute nodes, which are able to scale out with lower cost. The locality is quite important for accessing files effectively in NUSA file systems.

In this study, two scheduling algorithms–Data Aware Dispatch (DAD) and Improved Data Aware Dispatch (IDAD) are proposed for NUSA file systems to take advantages of the locality.

In the first approach, DAD introduced a parameter called $fileLocality$ to calculate the cost of accessing files, combined it with CPU load in advance into a comprehensive $Score$ to dispatch tasks with the consideration for both file access and CPU load.

In the second approach, IDAD introduced a per-task parameter called RDR, which is calculated based on time difference when a task is executed remotely and locally. The RDR is used to indicates the I/O intensity of the task, which will be used to better control task behavior during task dispatch.

These two approaches are implemented on the top of stock Torque scheduler and evaluated with three benchmarks. The evaluation consists of two parts: *Score* evaluation, and evaluation of scheduler as a whole.

In the evaluation of *Score*, Readgf and BLAST benchmark are used for evaluation, which shows that the *Score* is able to express the cost of accessing files properly in normal cases. On the other hand, in the evaluation of scheduler as a whole, thput-gfpio, Readgf, and BLAST benchmark are used for evaluation. In the evaluation with thput-gfpio, the two data-aware approaches improved average read throughput from 448MB/s to about 7000MB/s. The reason for this huge difference is unveiled with Readgf benchmark. In the evaluation of BLAST benchmark, DAD and IDAD reduced the makespan for 29.88% and 33.52% in comparison with stock Torque algorithms, respectively.

In this study, the DAD and IDAD are implemented only on Torque scheduler with Gfarm file system. However, they are not only applicable for Gfarm but also for other POSIX-compliant file systems like PPFS [40], as long as the scheduling module can acquire data placement and data size information from the metadata server of the underlying file system.

7.2 Future Work

In DAD, a global parameter β is used to strike the balance between CPU load and IO access. However, no viable method is given to calculate that value. Machine Learning may be a candidate for calculating the optimal β in this algorithm.

On the other hand, in IDAD, tasks with lower RDR are more likely to be executed since they are more likely to satisfy the standard of "local" in comparison with local threshold. It could be an issue when all of the low-RDR tasks finish, then some of the high RDR tasks have to be executed remotely, causing greater performance degradation. Obviously, a more adaptive algorithm should be introduced to rearrange the task execution sequence in the future.

In addition, prevention of human error is a major factor that benefits accuracy of the algorithms. As the referenced files are specified by a user, which eventually will be used to calculate the *Score* in DAD/IDAD, the accuracy of the algorithms will be heavily affected if the files specified are not correct. In a practical environment, a user may specify a large number of files but only some of them are actually accessed, just like the situation of the IDAD Score evaluation with Blast benchmark. This is because the mainstream users of Blast programs are from bioinformatics that have limited knowledge of execution mechanism. They usually regard all of ten files of the database as one integral part. Therefore, a feedback system should be implemented to tell users about the accuracy of the files specified so that they can define their file list properly for following executions.

Acknowledgement

First of all, I am sincerely and heartily grateful to my advisor Professor Osamu Tatebe for his insightful ideas and patience during my research activities. Whenever my research encounters difficulties, he always directed me to right path. Also, as an international student, interactions have not been smooth every time, he repeatedly communicated with me with great patience. Moreover, he supported me in joining various academic and technical conferences, domestically and internationally, which helped me keep up with the latest trend in research. I am really grateful to have him as my advisor.

In addition, I would like to thank the rest of my thesis committee: Professor Taisuke Boku, Professor Shuichi Oikawa, Professor Kento Aida, Associate Professor Hideyuki Kawashima for their efforts during the thesis defense. Their perceptive comments and constructive feedbacks helped me sharpen my thesis.

My sincere thanks also go to Professor Mitsuhisa Sato, Professor Daisuke Takahashi, Associate Professor Yoshiki Yamaguchi, Assistant Professor Hiroto Tadano, Dr. Masahiro Tanaka, and Dr. Norihisa Fujita for their supports and inspirations.

During my daily research activities, I appreciate Dr. Fuyumasa Takatsu and Dr. Mohamed Jabri for their help in solving technical difficulties in evaluation process and advice on my presentations. I also appreciate all other fellow students in the HPCS laboratory, especially those in the Big Data team, for their instructive insights.

Last but not least, I would like to thank my wife and parents for their continuous support and encouragement. This thesis would not have been possible without them.

Bibliography

[1] "European XFEL Data Handling." Available at `http://www.xfel.eu/`.

[2] Frank Schmuck and Roger Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pp. 231–244, 2002.

[3] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters," in *Proceeding of the Linux Symposium*, pp. 380–286, 2003.

[4] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow, "The NFS version 4 protocol," in *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, 2000.

[5] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm Grid File System," *New Generation Computing*, vol. 28, no. 3, pp. 257–275, 2010.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43, 2003.

[7] "Sun Grid Engine." Available at `http://gridscheduler.sourceforge.net`.

[8] "Torque Scheduler." Available at `http://www.adaptivecomputing.com/products/open-source/torque/`.

[9] B. Nitzberg, J. M. Schopf, and J. P. Jones, "PBS Pro: Grid computing and scheduling attributes," in *Grid resource management*, pp. 183–190, 2004.

[10] "Load Sharing Facility (Platform LSF)." Available at `http://www-03.ibm.com/systems/platformcomputing/products/lsf/index.html`.

[11] M. A. Yoo, Andy B.and Jette and M. Grondona, *SLURM: Simple Linux Utility for Resource Management*, pp. 44–60. 2003.

[12] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 265–278, 2010.

[13] R. B. Ross, R. Thakur, *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 391–430, 2000.

[14] "Open Grid Scheduler." Available at `http://gridscheduler.sourceforge.net`.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, 2013.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pp. 295–308, 2011.

[18] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 351–364, 2013.

[19] "Maui Scheduler." Available at http://www.adaptivecomputing.com/products/open-source/maui/.

[20] C. Henk and M. Szeredi, "FUSE: Filesystem in Userspace." Available at http://sourceforge.net/projects/fuse.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pp. 1–10, May 2010.

[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pp. 18:1–18:17, 2015.

[23] X. Li and O. Tatebe, "Data-Aware Task Dispatching for Batch Queuing System," *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–9, 2016.

[24] X. Li and O. Tatebe, "Improved Data-aware Task Dispatching for Batch Queuing Systems," in *Proceedings of the 7th International Workshop on Data-Intensive Computing in the Cloud*, DataCloud '16, pp. 37–44, 2016.

[25] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990.

[26] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. L. Madden, "BLAST+: architecture and applications," *BMC Bioinformatics*, vol. 10, no. 1, p. 421, 2009.

[27] G. Coulouris.el, "Blast Benchmarks." Available at http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchmark.

[28] P. Lu, Y. C. Lee, C. Wang, B. B. Zhou, J. Chen, and A. Y. Zomaya, "Workload Characteristic Oriented Scheduler for MapReduce," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 156–163, Dec 2012.

[29] M. Hammoud and M. F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pp. 570–576, Nov 2011.

[30] X. Tang, L. Wang, and Z. Geng, "A Reduce Task Scheduler for MapReduce with Minimum Transmission Cost Based on Sampling Evaluation," *International Journal of Database Theory and Application*, vol. Vol.8, no. 1, pp. 1–10, 2016.

[31] X. Wang and Y. Wang, "An Energy and Data Locality Aware Bi-level Multi-objective Task Scheduling Model Based on MapReduce for Cloud Computing," in *Proceedings of 2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, vol. 1, pp. 648–655, Dec 2012.

[32] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," in *Proceedings of 24th International Conference on Distributed Computing Systems, 2004*, pp. 342–349, 2004.

[33] T. Kosar and M. Balman, "A New Paradigm: Data-Aware Scheduling in Grid Computing," *Future Generation Computer Systems*, vol. 25, no. 4, pp. 406–413, 2009.

[34] W. Tang, J. Jenkins, F. Meyer, R. Ross, R. Kettimuthu, L. Winkler, X. Yang, T. Lehman, and N. Desai, "Data-Aware Resource Scheduling for Multicloud Workflows: A Fine-Grained Simulation Approach," in *Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, CLOUDCOM '14, pp. 887–892, 2014.

[35] K. Wang, K. Qiao, I. Sadooghi, X. Zhou, T. Li, M. Lang, and I. Raicu, "Load-balanced and Locality-aware Scheduling for Data-intensive Workloads at Extreme Scales," *Concurr. Comput. : Pract. Exper.*, vol. 28, pp. 70–94, Jan. 2016.

[36] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing Load Balancing and Data-Locality with Data-Aware Scheduling," in

Proceedings of 2014 IEEE International Conference on Big Data (Big Data), pp. 119–128, Oct 2014.

[37] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, “BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing,” in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID ’11, pp. 295–304, 2011.

[38] W. Xiaohui, W. W. Li, O. Tatebe, X. Gaochao, H. Liang, and J. Jiubin, “Implementing Data Aware Scheduling in Gfarm (R) Using LSF (TM) Scheduler plugin Mechanism,” in *Proceedings of International Conference on Grid Computing and Applications (GCA’05)*, pp. 3–10, 2005.

[39] J. Jiang, G. Xu, and X. Wei, “An Enhanced Data-Aware Scheduling Algorithm for Batch-mode Dataintensive Jobs on Data Grid,” in *Proceedings of 2006 International Conference on Hybrid Information Technology*, vol. 1, pp. 257–262, Nov 2006.

[40] F. Takatsu, K. Hiraga, and O. Tatebe, “PPFS: A Scale-Out Distributed File System for Post-Petascale Systems,” in *Proceedings of IEEE 2nd International Conference on Data Science and Systems (DSS)*, pp. 1477–1484, Dec 2016.

Appendix A

List of Publications

Journal Papers (Refereed)

1. Xieming Li, and Osamu Tatebe, Data-Aware Task Dispatching for Batch Queuing System, in IEEE Systems Journal, vol. PP, no.99, pp.1-9, 2015.

Conference Papers (Refereed)

1. Xieming Li, and Osamu Tatebe, Improved Data-Aware Task Scheduling Batch for Queuing Systems, DataCloud2016, Salt Lake City, Nov 2016.

Conference Papers (Unrefereed)

1. Xieming Li, and Osamu Tatebe, Data-Aware Dispatch, GPC 2015, Fiji, May 2015

2. Xieming Li, and Osamu Tatebe, Design of Data-Aware Scheduler. PRAGMA25 Student Workshop, Beijing, Oct 2013.

Other Publications

1. 李爕鳴, 建部修見. データ配置を考慮したタスクスケジューリング [J]. 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], 2013, 2013(19): 1-5.

Posters Papers (Refereed)

1. Xieming Li, and Osamu Tatebe, Proposal of task dispatch exploiting data locality for batch queuing system, 10th AEARU Work-shop on Computer Science and Web Technology (AEARU-CSWT 2015), Tsukuba, Feb. 2015.

2. Xieming Li, and Osamu Tatebe, Design of Data-Aware Scheduler. PRAGMA25, Beijing, Oct 2013.

Appendix B

Awards

1. Xieming Li and Osamu Tatebe, Data-Aware Scheduling. PRAGMA25 Student Workshop, Beijing, Oct 2013, Best Technical Award.