HIGH PERFORMANCE INTEGRATION OF DATA PARALLEL FILE SYSTEMS AND COMPUTING

Zhenhua Guo

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

May 2012

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Geoffrey Fox, Ph.D.
(Principal Advisor)

Judy Qiu, Ph.D.

Minaxi Gupta, Ph.D.

David Leake, Ph.D.

Acknowledgements

Abstract

The ongoing data deluge brings parallel and distributed computing into the new data-intensive computing era, where many assumptions made by prior research on grid and High-Performance Computing need to be reviewed to check their validity and explore their performance implication. Data parallel systems, which are different than traditional HPC architecture in that compute nodes and storage nodes are not separated, have been proposed and widely deployed in both industry and academia. Many research issues, which did not exist before or were not under serious consideration, arise in this new architecture and have drastic influence on performance and scalability. MapReduce has been introduced by the information retrieval community, and has quickly demonstrated its usefuleness, scalability and applicability. Its adoption of data centered approach yields higher throughput for data-intensive applications.

In this thesis, we present our investigation and improvement of MapReduce. We identify the inefficiency of various aspects of MapReduce such as data locality, task granularity, resource utilization, and fault tolerance, and propose algorithms to mitigate the performance issues. Extensive evaluation is presented to demonstrate the effectiveness of our proposed algorithms and approaches. Besides, we observe the inability of MapReduce to utilize cross-domain grid resources, and propose an extension of MapReduce called Hierarchical MapReduce (HMR). In addition, to speed up the execution of our bioinformatics data visualization pipelines containing both single-pass and iterative MapReduce jobs, a workflow management system Hybrid MapReduce (HyMR) is presented built upon Hadoop and Twister. The thesis also includes a detailed performance evaluation of Hadoop and some storage systems, which provides useful insights to both framework and application developers.

Contents

List of Tables

List of Figures

List of Acronyms

OpenMP Open Multi-Processing

MPI Message Passing Interface

PVM Parallel Virtual Machine

PGAS Partitioning Global Address Space

HPC High Performance Computing

HTC High Throughput Computing

MTC Many Task Computing

HyMR Hybrid MapReduce

HMR Hierarchical MapReduce

DAG Directed Acyclic Graph

DAGMan Directed Acyclic Graph Manager

BOINC Berkeley Open Infrastructure for Network Computing

GFS Google File System

HDFS Hadoop Distributed File System

HiveQL Hive Query Language

SQL Structured Query Language

UDF User Defined Function

NUCC Non Uniform Cluster Computing

GALS Globally Asynchronous Locally Synchronous

UDT UDP-based Data Transfer

SPE Sphere Processing Engine

SJF Shortest Job First

BASE Benefit Aware Speculative Execution

LSAP Linear Sum Assignment Problem

BoT Bag-of-Tasks

BoDT Bag-of-Divisible-Tasks

NFS Network File System

UPC Unified Parallel C

SPMD Single Program Multiple Data

GPFS General Purpose File System

DPSS Distributed Parallel Storage Server

SRB Storage Resource Broker

FTP File Transfer Protocol

PVFS Parallel Virtual File System

PBS Portable Batching System

VM Virtual Machine

APU Atomic Processing Unit

LATE Longest Approximate Time to End

POSIX Portable Operating System Interface

HTTP Hypertext Transfer Protocol

IU Indiana University

XML Extensible Markup Language

MDS Multidimensional Scaling

MI-MDS Majorizing Interpolation Multidimensional Scaling

SWG Smith Waterman Gotoh

PSA Pairwise Sequence Alignment

EM Expectation Maximization

1

Introduction and Background

1.1 Introduction

After the eras of experimental, theoretical and computational science, we now enter the new era of data-intensive science. Tony Hey's book *The Four Paradigm* depicts the characteristics of real data-intensive applications from various fields, including ocean science, ecological science and biology. The key characteristic is that scientists are overwhelmed with large data sets collected from many sources such as instruments, simulation, and sensor networks. Modern instruments such as Large Hadron Collider, next-generation gene sequencers and survey telescopes are collecting data in a unprecedented rate. For example, High Energy Physics experiments produce tens of Petabytes of data annually, and Large Synoptic Survey Telescope produces data at a rate of 20 Terabytes per day.

The goal of e-Science is to transform raw data to knowledge, and advanced tools are necessary to accelerate the time from data to insight. To process the ever-growing data requires large amounts of computing power and storage space, which far exceeds the processing capability of individual computers. Supercomputers, clusters, clouds, and data centers have been built to facilitate science discovery. These systems adopt different architectures aligned with their specific design goals.

Supercomputers are mostly used by scientists to run a small number of critical science applications efficiently. For example, the Earth Simulator, a highly parallel vector supercomputer developed by Japan, runs global climate models to evaluate the effects of global warming. They have low-latency interconnects to

accelerate the communication among computers. In grid computing, clusters from multiple domains are federated to enlarge available resource sets. These systems took a compute centered approach where compute nodes and storage systems are separated. As a result, parallel applications need to stage in data from storage nodes to compute nodes and store final results back to storage nodes (shown in Figure 1.1). In other words, its scheduling mechanism is bringing data to compute. Data grids were proposed to address the issue of how to manage large volume of data across multiple sites. They adopt a hierarchical architecture in which participant sites are usually under the control of different domains and each site manages a portion of the whole data set. Still data and compute are not closely integrated.

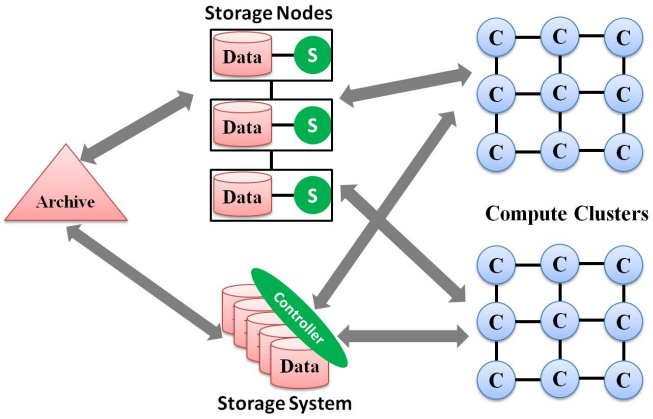


Figure 1.1: Architecture of traditional HPC systems

Data-intensive applications operate on large volume of data. The progress of network and storage technologies cannot keep pace with the rate of data growth, which is already a critical problem. Compute centered approach is not efficient because the cost to bring data to compute becomes significant and data transfer time may dominate the total execution time. For example, in grid the interconnection links between storage nodes and compute nodes are oversubscribed at a high ratio. That implies that the aggregate bandwidth of all nodes is way higher than the capacity of the interconnection links connecting storage and compute nodes. So only a few compute nodes/cores can concurrently fetch data from storage nodes at full speed. As a result, the interconnection links become bottleneck for data intensive applications. This fact renders it inefficient to bring data to compute.

For data intensive applications, the ratio of CPU instructions to I/O instructions is reduced so that data I/O plays a more important role than before. To support scalable data intensive computing, data parallel systems

have been developed among which MapReduce [47] and Dryad [66] are the most prominent technologies. These systems take a data centered approach where data affinity is explicitly supported. For example, local data accesses are favored over cross-rack data fetch. These systems can run some massively parallel information retrieval and web data processing applications on thousands of commodity machines, and achieve high throughput. Dryad was developed by Microsoft. But in late 2011, Microsoft dropped Dryad [6] and moved on to MapReduce/Hadoop.

1.2 Data Parallel Systems

Data parallel systems, which are natively designed for data-intensive applications, has been adopted widely in industry. The architecture is shown in Figure 1.2. The typical systems are GFS [55]/MapReduce[47], Cosmos/Dryad [66], and Sector/Sphere [57]. In these systems, the same set of nodes is used for both compute and storage. So each node is responsible for both computation and storage. This brings more scheduling flexibility to explore data locality compared with the traditional architecture. For instance, the scheduler can bring compute to data, bring compute close to data or bring data to compute. Ideally, most of network traffic should be confined to the same rack/chassis to minimize cross-rack traffic. Traditionally, parallel file systems make data access transparent to user applications by hiding the underlying details of data storage and movement. For data parallel systems, this is insufficient as the runtime needs to know more information to make optimal scheduling decisions. Thus location information of data is exposed to distributed runtimes.

1.2.1 Google File System (GFS)

GFS is a distributed file system designed to run on commodity hardware. It is optimized for write-once-read-many accesses of large files. Hadoop Distributed File System (HDFS) [101] is an open implementation of GFS. In GFS/HDFS, data availability is improved by maintaining multiple replicas. Files are split into equally-sized blocks distributed across nodes. GFS and HDFS use different terms to represent the partitioned data. The term is “chunk” in GFS and “block” in HDFS. To eliminate possible confusion, I will use “block” consistently in following text.

Fig. 1.3 shows the architecture of HDFS. A central *namenode* maintains the metadata of the file system.

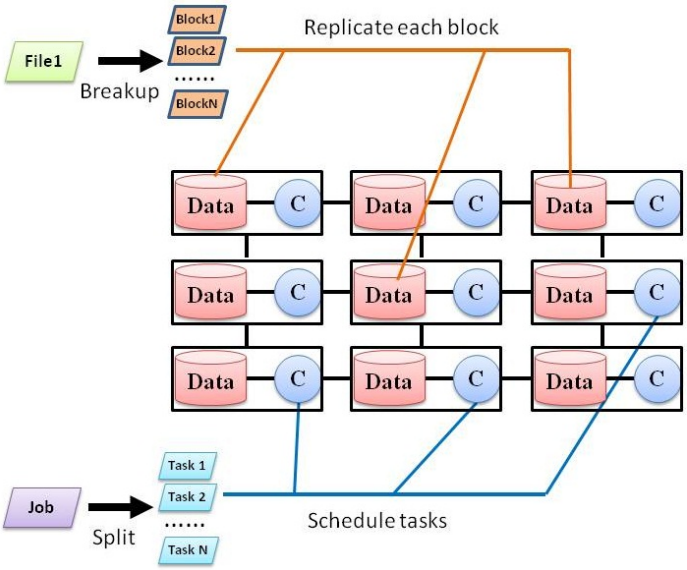


Figure 1.2: Architecture of data parallel systems

Its main responsibilities include block management and replication management. Real data are stored on *datanodes* managed by the *namenode*. HDFS allows administrators to specify network topology and thus is rack-aware. Metadata requests from HDFS clients are processed by *namenode*. Clients directly communicate with *datanodes* to read and write data. The *namenode* is a single point of failure. All metadata are lost if it fails permanently. To mitigate the problem, a secondary *namenode* runs simultaneously which performs periodic checkpoints of the image and edit log of the *namenode*.

1.2.2 MapReduce

MapReduce is a programming model and an associated implementation for processing large data sets.

1.2.2.1 MapReduce model:

In MapReduce, input data are organized as key-value pairs. MapReduce supports two primitive operations: *map* and *reduce* (shown in Table 1.1), which was inspired by Lisp and other functional languages. Each *map* operation is applied to a single key-value pair and generates a set of intermediate key-value pairs. Each *reduce* operation processes intermediate key-value pairs sharing the same key and generates final output. Because the

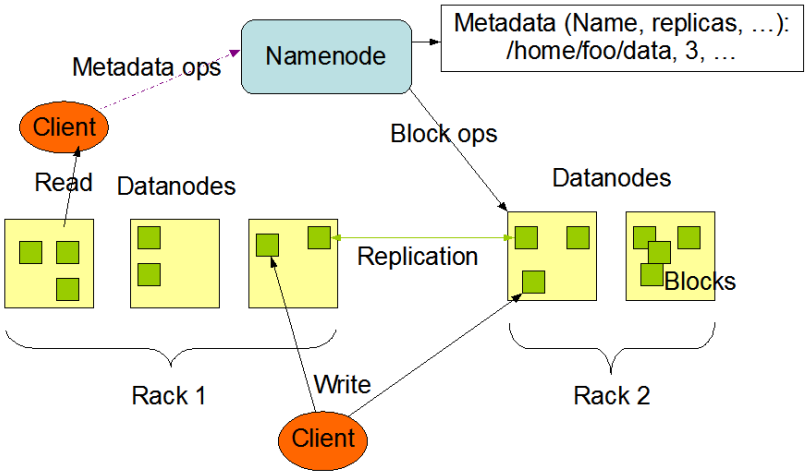


Figure 1.3: HDFS Architecture

* Copied from http://hadoop.apache.org/common/docs/current/hdfs_design.html

intermediate data produced by *map* operations may be too large to fit in memory, they are supplied to *reduce* operations via an iterator. It is users' responsibility to implement map and reduce operations. Although this model looks simple, it turns out many applications can be expressed easily such as distributed grep, reversing web-link graph, distributed sorting and inverted indexing.

map	(k1,v1) $\rightarrow$ list(k2,v2)
reduce	(k2,list(v2)) $\rightarrow$ list(v2)

Table 1.1: map and reduce operations

1.2.2.2 MapReduce runtime:

The MapReduce runtime allows the programs written in MapReduce style to be automatically parallelized and executed on clusters. The runtime is designed to run on large clusters of commodity hardware. Hadoop is an open source implementation of MapReduce.

Primitive *map* and *reduce* operations are organized into schedulable map and reduce tasks. Each MapReduce job is comprised of some number of map and reduce tasks. By default each map task processes the

data of one block. Each node has a configurable number of map and reduce slots, which limit the maximum number of map and reduce tasks that can concurrently run on the node. When a task starts to execute, it *occupies* one slot; and when it completes, the slot is *released* so that other tasks can use it. Conceptually, each slot can only have one task assigned at most at any time. There is a master node where *Job Tracker* runs. The Job Tracker manages all slave/worker nodes and runs a scheduler that assigns tasks to idle slots. When a slave node sends a heartbeat message and says it has available map slots, the master node first tries to find a map task whose input data are stored on that slave node. This is made possible because data location information is exposed by the underlying storage system GFS/HDFS. If such a task can be found, it is scheduled to the node and node-level data locality is achieved. Otherwise, Hadoop tries to find a task for rack-level data locality where input data and execution are on the same rack. If it still fails, Hadoop randomly picks a task to dispatch to the node. The default Hadoop scheduling policy is optimized for data locality.

To sum up, MapReduce integrates data affinity to facilitate data locality aware scheduling. It considers different levels of data locality, such as node level, rack level and data center level. Node level data locality means a task and its input data are co-located on the same node. Rack level data locality means a task and its input data are on different nodes that are located on the same rack.

Fig. 1.4 shows the execution breakdown of MapReduce application *wordcount* which counts the number of occurrences of each word. The input text is split into three blocks. The implementations of map and reduce operations are shown near bottom left corner. Each block is fed into the map operation which tokenizes the text and emits value 1 for each encountered word. After all map tasks complete, the intermediate data are grouped by key and shuffled to the corresponding reduce task. A *partitioner* is used to compute the mapping from intermediate keys to reduce tasks. After a reduce task collects all its input data, the reduce operation is applied which adds up the intermediate values and produces the final word count.

1.3 Motivation

Among the proposed data parallel systems [47] [57] [66] [50], MapReduce has attracted a great deal of attention because of its ease of use, scalability, fault tolerance and so on. Since the initial publication of the

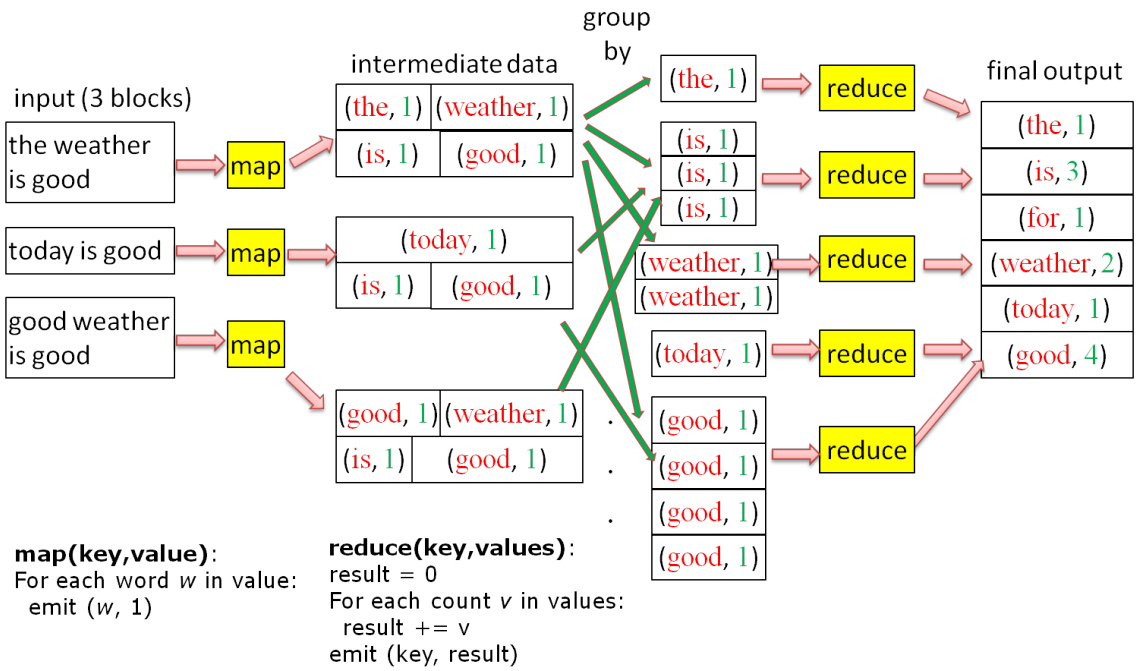


Figure 1.4: Break down the execution of MapReduce application *wordcount*

MapReduce paper back in 2004, it has been cited by more than 4000 papers. MapReduce has been used to run large-scale data processing applications [92, 42, 47]. However compared with traditional systems (e.g. MPI), they are still relatively new and have not undergone substantial research. Both theoretical and practical investigation is needed to fully reveal their efficiency and limitations. For example, default Hadoop scheduler favors data locality, but it is not clear yet which factors impact data locality and to what extent, and whether good data locality can be sustained if a system runs many jobs with diverse workload from multiple users. Tuning runtime parameters to maximize performance is difficult and requires many trials before satisfactory settings are found. For example, Hadoop has around 200 parameters that can be tuned by users and it is non-trivial to figure out optimal settings. Much of prior research on grid computing does not help here because they have different assumptions about the system architecture. This thesis aims to investigate in depth various important aspects of MapReduce that have not been carefully examined including data locality, task granularity, resource utilization and speculative execution, and propose enhancements to MapReduce model to improve its applicability and efficiency.

1.4 Problem Definition

Data parallel systems have been used to tackle large scale data processing and shown promising results for data-intensive applications. However, we noticed the following issues in MapReduce model and runtimes which we strive to solve in this thesis.

- The performance of underlying storage systems has direct impact on the execution of upper-level parallel applications. A detailed performance evaluation of Hadoop and contemporary storage systems has not been conducted. We hope to reveal their performance advantages/disadvantages and show how efficient they are.

- Hadoop assumes the work done by each task in a parallel job is similar to simplify implementation. However, this assumption does not always hold and thus severe load unbalancing can occur if tasks are intrinsically heterogeneous. We intend to figure out the ways to adjust tasks so that they are well balanced and thus job turnaround time is reduced.

- From our preliminary research, we have identified that in Hadoop resource utilization is constrained by the relationship between the numbers of task slots and tasks. By incorporating prior research from the HPC community, we wish to address the challenge of maximizing the efficiency of resource usage without interfering with native scheduling.

- As we have shown, in data-intensive computing to move data around is not feasible and data locality becomes critical. The important of data locality has drawn attention in not only the MapReduce community but also traditional grid computing communities. Current implementations use data-locality favored heuristics to schedule tasks. However, data locality itself has not been carefully analyzed theoretically. We intend to figure out the importance of data locality and the (sub-)optimality of state-of-the-art MapReduce scheduling algorithms.

- During our use of MapReduce runtimes, we found they cannot be deployed on cross-domain clusters and users need to deploy and manage separate MapReduce runtimes. The coordination of multiple MapReduce deployments to run a single parallel job is cumbersome, time-consuming and error-prone. We want to address the challenge of simplifying cross-domain job execution. Besides, for a complex

problem which consists of both MapReduce algorithms and iterative MapReduce algorithms, the user needs to manually interact with different runtimes (e.g. Hadoop, Twister) to run individual jobs. A framework that seamlessly integrates and manages different runtimes is desired.

1.5 Contributions

We summarize the contributions of our research:

- A detailed performance analysis of widely used runtimes and storage systems is presented to reveal both the speedup and overhead they bring. Surprisingly, the performance of some well-known storage systems degrades significantly compared with native local I/O subsystems. Our findings show that substantial tuning and optimization of those systems are needed to fully exploit the power of underlying systems.
- For MapReduce, a mathematical model is formally built with reasonable data placement assumptions. Under the formulation, we deduce the relationship between influential system factors and data locality, so that users can predict the expected data locality.
- The sub-optimality of default Hadoop scheduler is revealed and an optimal algorithm based on Linear Sum Assignment Problem (LSAP) is proposed.
- Based on existing Bag-of-Tasks (BoT) model, a new task model Bag-of-Divisible-Tasks (BoDT) is proposed. Upon BoDT, new mechanisms are proposed that improve load balancing by adjusting task granularity dynamically and adaptively. Given BoDT model, we demonstrate that Shortest Job First (SJF) strategy achieves optimal average job turnaround time with the assumption that work is arbitrarily divisible.
- We propose Resource Stealing to maximize resource utilization, and Benefit Aware Speculative Execution (BASE) to eliminate the launches of non beneficial speculative tasks and thus improve the efficiency of resource usage.
- To enable cross-domain MapReduce execution, Hierarchical MapReduce (HMR) is presented which circumvents the administrative boundary of separate grid clusters. To use both MapReduce runtimes

and iterative MapReduce runtimes in a single pipeline of jobs, Hybrid MapReduce (HyMR) is proposed which combines the best of both worlds.

1.6 Dissertation Outline

We present the state-of-the-art parallel programming models, languages and runtimes in chapter 2. What we have surveyed ranges from traditional HPC frameworks to recent data parallel frameworks, from low-level primitive programming support to high-level sophisticated abstraction, from shared-memory architecture to distributed-memory architecture. We focus on the level of abstraction, the target parallel architecture, large-data computation, and fault tolerance. By comparison, we demonstrate the advantage of data parallel systems over HPC systems for data-intensive applications for which "move compute to data" is more appropriate than "move data to compute".

In chapter 3, we evaluate the performance of Hadoop and some storage systems by conducting extensive experiments. We show how important system configurations such as data size, cluster size and per-node concurrency impact the performance and parallel efficiency. This performance evaluation can give us some insights about the current state of data parallel systems. In addition, the performance of local file system, NFS, HDFS and Swift are evaluated and compared.

In following chapters, we use MapReduce/Hadoop as the target research platform. The integration of data locality into task scheduling is a significant advantage of MapReduce. In chapter 4, we investigate the impact of various system factors on data locality. Besides, the non-optimality of default Hadoop scheduling is illustrated and an optimal scheduling algorithm is proposed. We conclude that our algorithm can improve performance significantly. Besides, we evaluate the importance of data locality for different cluster environments (single-cluster, cross-cluster, and HPC-style setup with heterogeneous network) by measuring how data locality impacts job execution time.

Hadoop is not efficient to run jobs with heterogeneous tasks. In chapter 5, the drawbacks of fixed task granularity in Hadoop will be analyzed. Task consolidation and splitting are proposed which dynamically balance different tasks for the scenarios where prior knowledge is either known or unknown. In addition, we consider the multi-job scenario and prove that Shortest Job First (SJF) strategy yields optimal average job

turnaround time.

The hard partition of physical processing capability to virtual map and reduce slots may limit the utilization of resource when all slots are not occupied. Besides the mechanism to launch speculative tasks in Hadoop is not efficient, which may result in the waste of resources. We mitigate these problems in chapter 6 where resource stealing and Benefit Aware Speculative Execution (BASE) are proposed.

In chapter 7, Hierarchical MapReduce (HMR) and HyMR are proposed which expand the environments where MapReduce can be used and enable the simultaneous use of regular and iterative MapReduce runtimes respectively.

Related work is presented in chapter 8, following which conclusions are drawn and future work is outlined in chapter 9.

2

Parallel Programming Models and Distributed Computing Runtimes

To fully utilize the parallel processing power of a cluster of machines, two critical steps are required. Firstly, domain-specific problems need to be parallelized and converted to the programming model programmers choose. The chosen programming model should match the structure of the original problem naturally. Secondly, a distributed computing runtime is critical to manage highly distributed resources and run parallel programs efficiently.

Parallel computing models and languages relieve programmers from the details of synchronization, multithreading, fault tolerance, etc. So programmers can focus on how to express their domain-specific problems in the chosen language or model. Each language or model has different tradeoffs among expressiveness, usability and efficiency. Many models have been proposed for shared-memory (e.g. OpenMP, multithreading), distributed shared-memory (e.g. PGAS) and distributed-memory platforms (e.g. MapReduce). Which model or language to choose depends on application characteristic (e.g. CPU bound vs. IO bound, MapReduce vs. Directed Acyclic Graph (DAG)), system architecture (e.g. shared memory, distributed memory with low-latency network) and performance goal (e.g. latency vs. throughput).

Depending on the characteristics of hardware and application, the design of distributed computing runtimes has different goals. High Performance Computing (HPC), High Throughput Computing (HTC), and Many Task Computing (MTC) are three well-researched categories. HPC emphasizes the use of large

amounts of parallel computing power for short periods of time (e.g. hours and days) to complete a computation job. Tasks in HPC jobs are tightly coupled. HPC is mainly designed for supercomputers and clusters with low-latency interconnects. Top500 [13] uses LINPACK as the benchmark to measure the performance of the fastest machines in the world which is expressed in flops per second. HTC emphasizes on executing as many tasks/jobs as possible for much longer time spans (e.g. months) and thus it prefers overall system throughput to the peak performance of an individual task. So HTC is measured by the number of tasks/jobs processed per month or per year. HTC can span over multiple clusters across different administrative domains by using various grid computing technologies. MTC lies between HPC and HTC, which emphasizes on utilizing as many resources as possible to accomplish a large number of computational tasks over short periods of time. Usually tasks in MTC are less coupled than those in HPC. In addition, MTC takes into consideration data-intensive computing where the volume of processed data may be extremely large.

Below we survey the main programming languages, models and execution runtimes, the taxonomy of which is shown in Fig. 2.1.

2.1 Programming Models

For most non-trivial applications, sequential computation model quickly overwhelms the processing capability of a single thread/process when the size of input data scales up. Parallelism of different types (e.g. task parallelism, data parallelism) can be exploited to accelerate task execution significantly. Several programming models have been proposed to unify the programming constructs or hide the complexity of managing underlying distributed resources. Overall they ease the development of parallel programs which run on multiple processor cores or multiple machines.

2.1.1 MultiThreading

A process is a runnable unit which owns resources allocated by the operating system. By default, processes do not share address spaces or file handles. A thread is a lightweight process. Usually a process can run multiple threads which share the same address space. Each thread owns a section of the process address

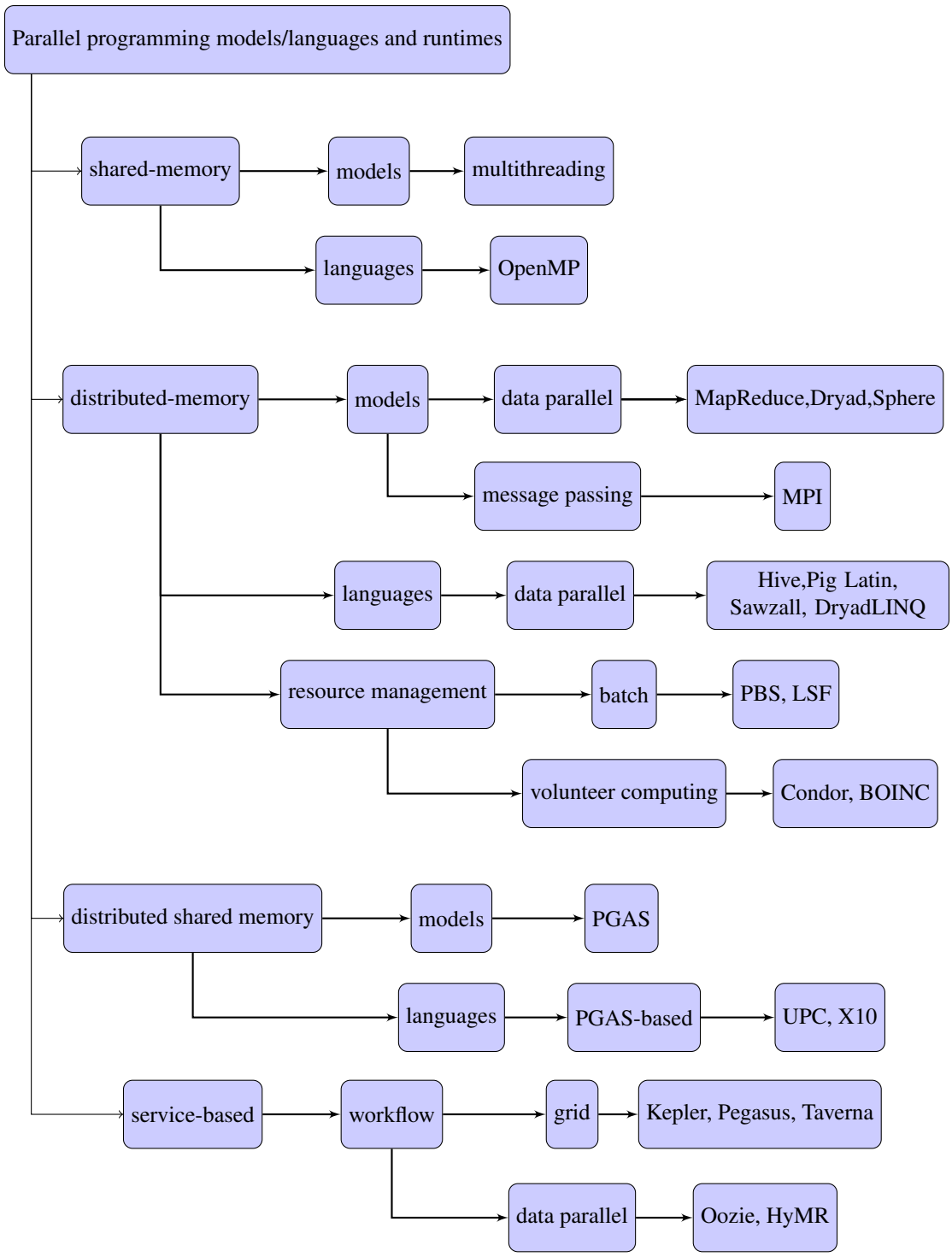


Figure 2.1: Taxonomy

space where thread-local stacks and variables reside. Inter-thread communication within the same process incurs much lower overhead than inter-process communication. The creation and termination of threads is faster than that of processes. Depending on implementation, threads are categorized into kernel threads and user threads. Kernel threads are managed by the kernel scheduler and thus can be scheduled independently. User threads are managed and scheduled in user space and the kernel is not aware of them. It is faster to create, manage and swap user threads than kernel threads. But multithreading is limited for user threads. When a user thread is blocked, the whole process is blocked as well and thus all other threads within the process get blocked even if some of them are ready to run.

For parallel programming, programmers need to explicitly control the synchronization among participant threads to protect the critical memory region accessed concurrently. The commonly used synchronization primitives include locks, mutexes, semaphores, conditional variables, etc.

Despite the conceptual simplicity, threading has several drawbacks. The isolation of thread execution is less rigid. The misbehavior of a thread can crash the whole process and thus result in the termination of the execution of all other threads belonging to the same process. In contrast, process-level isolation is more secure. In addition, multithreading only provides primitive building blocks which theoretically can be used to construct complicated applications. In practice, the complexity of writing and verifying thread-based parallel programs overwhelms the programmers even for modest-sized problems. Following is an excerpt from Lee's paper [77].

> "Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism." – *The Problem with Threads*, Edward A. Lee, UC Berkeley, 2006 [77]

2.1.2 Open Multi-Processing (OpenMP)

For shared memory system, OpenMP provides API that supports multiprocessing on multiple architectures. Its goal is to simplify writing multi-threaded programs. OpenMP is built upon multithreading. It adopts

fork-join paradigm for parallelization. The master thread controls the execution logic and spawns multiple slave threads to collaboratively complete the work when needed. After the work is done, slave threads join back into the master thread, which continues to run until the end of the program. Threads communicate by sharing variables. Sometimes explicit synchronization is still needed to protect data conflicts and control race condition.

OpenMP supports C, C++ and Fortran. For each language, special compiler directives are defined to allow the users to explicitly mark the sections of code that can be run in paralle. Additional library routines are standardized from which each thread can get context information such as thread id.

Fig. 2.2 shows the solution stack of OpenMP. At the bottom is hardware which has multiple processors and maintains a shared address space. On top of it is the operating system with the support of shared memory and multithreading. OpenMP runtime library utilizes the functionalities supported by OS and provides runtime support for upper level applications. Multiprocessing directives, OpenMP library and environment variables, which are the main building blocks to write parallel programs, are exposed to user applications.

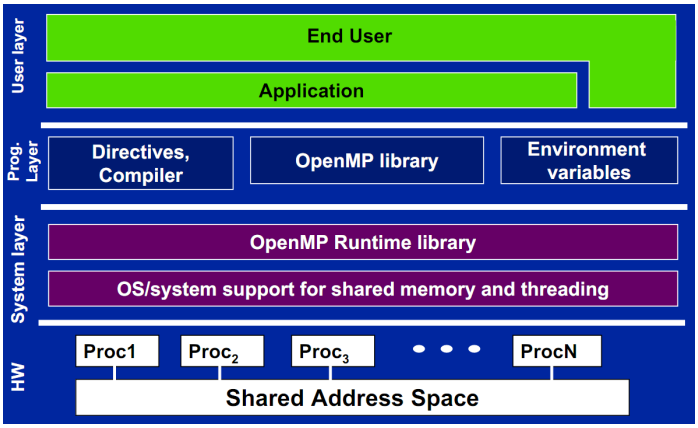


Figure 2.2: OpenMP stack

* Copied from [4]

Fig. 2.3 shows a simple OpenMP C example. Two arrays are declared and initialized in lines 5 and 6. The number of threads is set to 4 in line 8. *omp_set_num_threads* is a routine provided by OpenMP library. The directive in line 10 tells the compiler to parallelize the subsequent for loop which adds up arrays a and b in an element-wise manner. The master thread spawns 4 slave threads each of which adds one pair of elements.

After all slave threads complete, they join into the master thread. Then we calculate the sum of the elements in array a. We use "reduction" directive to "reduce" array a to a single value.

```
#include "omp.h"
#include <stdio.h>

void main() {
  double a[4] = {0, 1, 2, 3};
  double b[4] = {10, 11, 12, 13};
  int i;
  omp_set_num_threads(4);   // set the number of threads to 4

  #pragma omp parallel for  // parallelize for loop
  for(i = 0; i < 4; ++i) {  // add two arrays
    a[i] += b[i];
  }

  double sum = 0.0;
  #pragma omp parallel for reduction (+:sum) // parallelize and reduce
  for(i = 0; i < 4; ++i) {
    sum += a[i];   // the array elements are "reduced" to sum
  }

  printf("%f", sum);
}
```

Figure 2.3: OpenMP C program example

2.1.3 Message Passing Interface (MPI)

Multithreading and OpenMP are designed mainly for shared memory platforms. MPI is a portable, efficient, and flexible standard specifying the interfaces that can be used by message-passing programs on distributed memory platforms. MPI itself is not an implementation, but a vendor-independent specification about what functionalities a standard-compliant library and runtime should provide. MPI interfaces have been defined for languages C/C++ and Fortran. There are a variety of implementations available in public domain

(e.g. OpenMPI [10], MPICH [7, 8]). Usually a shared file system (e.g. General Purpose File System (GPFS)) is mounted to all compute nodes to facilitate data sharing.

Programmers identify the parallelism and use MPI constructs to write parallel programs. During runtime, each MPI program may concurrently run multiple processes. Each communication may involve all processes, or a portion of processes. MPI defines *communicators* and *groups* which define the communication context and are used to specify which processes may communicate with each other. The processes within a process group are ordered and each process is identified by its rank in the group assigned automatically by the MPI system during the initialization. The identifiers are used by application developers to specify the source and destination of messages. The pre-defined default communicator is *MPI_COMM_WORLD* which includes all processes. MPI-1 supports both point-to-point and collective communication. MPI guarantees messages do not overtake each other. Fairness of communication handling is not guaranteed in MPI, so it is the users' responsibility to prevent starvation.

Point-to-Point communication routines: The basic support operations are *send* and *receive*. Different types of routines are provided including synchronous send, blocking send/receive, non-blocking send/ receive, buffered send, combined send/receive and "ready" send. Blocking send calls do not return until the message data have been safely stored so that the sender is free to modify the send buffer. However, the messages may have not been sent out. The usual way to implement it is to copy the data to a temporary system buffer and thus it incurs the additional overhead of memory-to-memory copying. Alternatively MPI implementations may choose not to buffer messages for performance reasons. In this case, a send call does not return until the data have has been moved to the matching receiver. In other words, the sender and receiver may or may not be loosely coupled depending on implementations. Synchronous send calls do not return until a matching receiver is found and starts to receive the message. Blocking receive calls do not complete until the message is received. A communication buffer should not be accessed or modified until the corresponding communication completes. To maximize performance, MPI provides nonblocking communication routines that can be used to make communication and computation overlap as much as possible. A nonblocking send call initiates the operation and returns before the message is copied out of the send buffer. The program can continue to run while the message is copied out of the send buffer simultaneous in background by MPI runtime. A nonblocking receive call initiates the operation and returns before a message is received and stored into the receive buffer.

Collective communication routines: Only two processes can be involved in point-to-point communications. Collective communication mechanisms allow more than two processes to communicate. The collective communication mechanism supported by MPI include *barrier* synchronization, *broadcast*, *gather*, *scatter*, *gather-to-all*, *reduction*, *reduce-scatter*, *scan*, etc. When reaching the *barrier* synchronization point, each process blocks until all processes in the group reach the same point. *Broadcasts* send a message from a "root" process to all other processes in the same group. *Scatter* distributes data from a single source process to each process in the group, and each process receives a portion of the data (i.e. the message is split into n segments and the i-th segment is sent to the i-th process). The *gather* operation allows a destination process to receive messages from all other processes in the group and store them in rank order. *Gather-to-all* distributes the concatenation of the data across processes to all processes. *Reduce* applies a reduction operation across all members of a group. In other words, it operates on a list of data elements stored in different processes and produces a single output stored in the specified process. One example is sum calculation across all data distributed across processes. *Reduce-scatter* applies element-wise reduction on a vector and distributes the result across the processes.

Process Topologies: MPI allows programmers to create a virtual topology and map MPI processes to positions in the topology. Two types of topologies are supported - Catesian (grid) and Graph. MPI does not define how to map virtual topologies to the physical structure of the underlying parallel system. Process topologies are usually used for the purposes of convenience or efficiency. Domain-specific communication patterns can be expressed easily with process topologies and ease the application development. For most parallel systems, the communication cost is not constant for all pairs of nodes (e.g. some nodes are "closer" than others). Process topologies can help MPI runtime to optimize the mapping of processes to physical processors/cores based on the physical characteristics and structures.

MPI I/O: MPI I/O adds parallel I/O support to MPI. It provides a high-level interface to describe data partitioning and data transfers. It lets users read and write files in synchronous and asynchronous modes. Accesses to a file can be independent or collective. Collective accesses allow for read and write optimization on various levels. MPI data types are used to express the data layout in files and data partitioning among processes. There has been substantial research on how to improve the performance of parallel IO [106, 76, 107, 40, 113].

Summary: Overall, MPI provides powerful communication primitives that can be used by application

developers to coordinate different tasks of a single parallel program. In MPI 1.0 and 2.0, fault tolerance is not supported, and it is the users' responsibility to recover their programs from faults (e.g. hardware failure, process hang, network paritition). There have been some MPI extensions that support process checkpointing and recovery [62, 63] but they have not been standardized. In addition, data affinity/locality is not incorporated, which makes it inappropriate to run massively parallel applications on commodity clusters.

2.1.4 Partitioning Global Address Space (PGAS)

In shared memory model, data sharing is easy because the same memory region can be made accessible to multiple processes. In contrast, distributed memory platforms involve more hassle when intermediate data need to be shared among multiple processes. Usually, the programmer needs to write code to explicitly move data around for sharing. PGAS adopts distributed shared memory model. The whole memory address space is partitioned into two portions: shared area and private area. The shared area is a global memory address space which is directly accessible by any process. So they can be used to store globally shared data. On the implementation side, the shared area is partitioned and physically resides on multiple machines. It is the PGAS runtime that creates the "illusion" of shared memory on top of distributed memory architecture. Therefore, the performance may vary depending on where the accessed data is stored physically. Each process has affinity with a portion of the shared area and PGAS can exploit the reference locality. Shared data objects are placed in memory based on affinity, Private area is local to the corresponding process and thus not accessible by other processes. So any process can access global memory while only the local process may reference private area.

2.1.4.1 Unified Parallel C (UPC)

UPC is an parallel extension of ANSI C based on PGAS model. It was the successor of the early research projects such as Split-C, AC, and PCP. Currently, it is being maintained by Berkeley lab. UPC adopts Single Program Multiple Data (SPMD) model where the same task is run concurrently with different input to speed up the execution. Static and dynamic memory allocation are supported for both private and shared memory. UPC provides standard routines to support data movement from/to shared memory. UPC does not enforce

implicit synchronization among processes. Instead, it provides synchronization mechanisms such as barriers, locks, and fences.

2.1.5 MapReduce

MapReduce [47] was initially proposed by Google and has gained popularity quickly in both industry and academia. MapReduce and its storage system GFS have been discussed in detail in section 1.2.2 and 1.2.1. Upon MapReduce model, some enhancements such as Map-Reduce-Merge and MapReduce online have been proposed with more features that enable MapReduce to be applied to more types of applications.

2.1.6 Iterative MapReduce

A large collection of science applications is iterative in that each iteration processes the data produced in last iteration and generates intermediate data that will be used as input by the next iteration until the result converges. Two typical examples are K-means and Expectation Maximization (EM). Iterative MapReduce can be implemented by pipelining multiple separate MapReduce jobs. However, this approach has significant drawbacks. Firstly, for a logically individual problem, it involves multiple jobs the number of which is determined by convergence condition, so the additional overhead of starting, scheduling, managing and terminating jobs is inevitably incurred. Secondly, the intermediate data are serialized into disks and deserialized back into memory across iterations, which results in substantial overhead of disk accesses that are much slower than memory accesses. Iterative MapReduce runtimes have been developed to mitigate the issues.

2.2 Batch Queuing Systems

Batch queuing systems are usually used to manage resources in supercomputers and dedicated clusters. They manage only compute resources, and need to efficiently handle the resource requests of both small sequential programs running on a single node and massively parallel programs running on thousands of nodes. Batch queuing systems need to guarantee fairness while maximizing resource utilization. Fairness means real resource allocation should match the configured rations as closely as possible. From the perspective of system

owners, to keep the system as busy as possible (but not overloaded) makes full use of resources and thus is preferred. Compared with data centered approach, storage nodes and compute nodes are separated and batch queuing systems adopt "bring data to compute" paradigm. For data-intensive applications, this approach incurs substantial network traffic and thus is inefficient.

In batch queuing systems, jobs are submitted by users to job queues each of which has an associated priority. Each job is specified in a job description language/script by which users can specify how many compute nodes are needed, how long they are needed, and the location of output/error files. Based on the running jobs and jobs in queue, batch queuing systems calculate the availability of resources and reserve the specified number of nodes for a period of time. Mostly, compute nodes share a mounted global file systems. Input data can be stored in the shared file system, or programmers need to write scripts to explicitly stage in data. Usually, the order of job execution is determined based on submission time, the priority of the submitter's account, and resource use history. To strictly follow the rules may result in significant resource fragmentation. For example, in a system comprising 5 nodes, task A is running and using 3 nodes while task B and C are in queue which require 5 nodes and 2 nodes respectively. Task A will run for 1 hour before completion and tasks B and C will run for 2 hours and 30 minutes respectively. Because task B is placed in front of task C, C will not run until B starts to run. However, task C can be scheduled to run immediately without impacting the execution of task B at all (anyway task B can start only after task A completes). Backfilling [84] allows small jobs to leapfrog the large waiting jobs in the front of the queue without incurring significant delay on other jobs when there are sufficient idle resources to run those small jobs.

2.3 Data Parallel Runtimes

Data parallel runtimes explicitly exploit the data parallelism of the computation and provide simple programming models where users can plug in their sequential implementation and obtain parallelism automatically at run time. Once domain-specific problems are cast into the model, the runtime is able to automatically parallelize the execution and schedule tasks. As a result, developers do not need to care about resource allocation, threading, concurrency control, synchronization and fault tolerance which are known to be difficult to program with.

2.3.1 Hadoop

Hadoop is an open source implementation of MapReduce developed under the umbrella of Apache Source Foundation. Hadoop community is developing MapReduce 2.0 which dramatically changes the architecture to i) separate resource management and task scheduling/management, ii) mitigate the performance bottleneck of a single master node.

Some higher level projects have been built on top of Haddop and add additional functionalities. For example, Apache Mahout implements many widely used data mining and machine learning algorithms in a parallel manner so that data of larger-scale can be processed efficiently. Hive is a data warehouse software that supports querying and managing large data sets. Queries are converted to MapReduce jobs which are run on Hadoop.

2.3.2 Iterative MapReduce Runtimes

Some frameworks and enhancements to MapReduce have been proposed for iterative MapReduce applications. HaLoop [29] modifies Hadoop to provide various caching options and reuse the same set of tasks to process data across iterations (i.e. tasks are loop-aware). Twister [50] and Spark [119] reuse the same set of "persistent" tasks to process intermediate data across iterations. Significant performance improvement over MapReduce has been shown for these frameworks.

2.3.3 Cosmos/Dryad

MapReduce model is limited in the expressiveness. Although many applications can be artificially transformed to MapReduce form (e.g. split the application into multiple MapReduce jobs), the transformation i) may be unnatural and complicate the writing of programs, ii) may have significant performance implication. Iterative MapReduce is one example. Dryad is an execution engine for data-parallel applications. It provides a DAG model which can encode both computation and communication and thus is more powerful and expressive than MapReduce. Each vertex in the graph corresponds to a task that can be executed on available nodes after its prerequisite tasks have finished and the input data have been staged in. Dryad scheduler maps vertices to compute nodes with the goal of maximizing concurrency. So independent vertices can be run

simultaneously on multiple nodes or multiple cores on a single node. Difference types of communication channels are supported such as files, TCP pipes and shared-memory FIFOs. Dryad automatically monitors the whole system and recovers from computer or network failures. If a vertex fails, Dryad reruns the task on a different node. A version number is associated with each vertex execution to avoid conflicts. If the read of input data fails, Dryad reruns the corresponding upstream vertex to re-generate the data. In initial version, greedy scheduling was adopted by the job manager with the assumption that it is the only job running in the cluster. Dryad applies run-time optimization that dynamically refines the graph structure according to network topology and application workload. Dryad runs on top of Cosmos which is a distributed file system that facilitates sharing and managing distributed data sets across the whole cluster. Although Dryad provides more advanced features compared to MapReduce, its use in both industry and academia is really limited and thus its scalability and performance for running diverse parallel applications have not been demonstrated.

Fig. 2.4 shows the Dryad ecosystem. Many languages such as Nebula and DryadLINQ have been extended to integrate the processing capability of underlying Dryad systems. Existing sequential programs can be easily modified to become parallel.

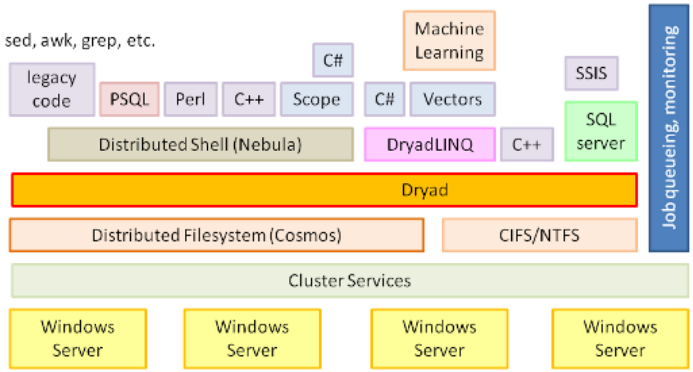


Figure 2.4: Dryad ecosystem

* Copied from http://research.microsoft.com/en-us/projects/dryad/

2.3.4 Sector and Sphere

Sector [57] is a user-space distributed file system. Sector files, which are stored in the local file systems of one or more slave nodes, are not split into blocks. So if files are too large, users need to manually split

them into multiple files of smaller size. Sector can manage data distributed across geographically scattered data centers. One assumption made by Sector is that nodes are interconnected with high-speed network links. Sector is network topology ware, which means network structure is considered when data are managed. Data in Sector are replicated and per-file replication factor can be specified by users. Sector allows users to specify where replicas are placed (e.g. on local rack, on a remote rack). Permanent file system metadata is not required. If file system metadata is corrupted, the metadata can be rebuilt from real data. Data transfer is done with a specific transport protocol called UDP-based Data Transfer (UDT) which provides reliability control and congestion control. UDT has been shown to be fast and firewall friendly, and used by both commercial companies and research projects. Fig. 2.5 shows the architecture of Sector. Sector adopts a master-slave architecture. The Security Server maintains user accounts, file access permission information and authorized slave nodes. The Master Server maintains file system metadata, monitors slave nodes and responds to users' requests. Real data are stored on slave nodes.

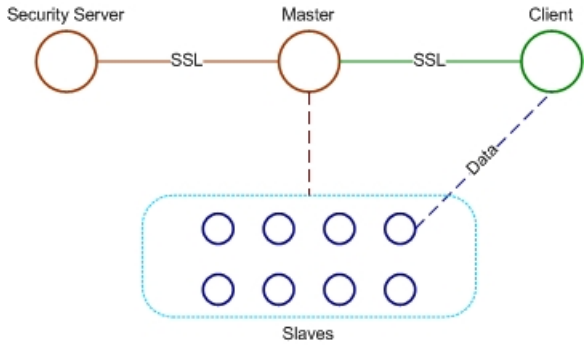


Figure 2.5: Sector system architecture

* Copied from [57]

Sphere [57] is a distributed runtime built upon Sector. Data are processed by Sphere Processing Engines (Sphere Processing Engines (SPEs)) each of which processes a segment of data records. Multiple input streams can be processed simultaneously. In-situ processing is achieved with best efforts by processing data near where it resides. User specified User Defined Functions (UDFs) can be plugged in and applied to data processing. In this sense, it is more generic than MapReduce model.

2.4 Cycle Scavenging and Volunteer Computing

Besides the massive computing power provided by dedicated supercomputers and clusters, personal computers can also provide a large amount of aggregate processes capability and storage space. The utilization of most PCs is low because they are mainly used for simple daily non compute-intensive work (e.g. web surfing, email send/receive). In other words, they have a large number of spare resources if combined. To efficiently harvest those idle CPU cycles and spare storage capacity can provision resources to complex science applications without incurring significant additional cost. Such systems are categorized into Volunteer Computing where computing resources are donated by owners. Because the contributed resources are highly distributed, network connection among them is drastically heterogeneous and each machine may join or leave at any time. As a result, the applications suitable for volunteer computing i) should not be IO-intensive, ii) have no or little communication among tasks, iii) are flexible for job completion time.

2.4.1 Condor

Condor [79], a HTC runtime developed by University of Wisconsin, runs on large collections of distributively owned resources. It can integrate dedicated clusters and personal computers seamlessly into a single environment. Condor provides match-making mechanism to match the resource requirement of jobs and the physical resource capacity. Condor allows users to run existing applications with little or no modification. Condor supports both sequential and parallel jobs. Applications can be re-linked with Condor's I/O library to facilitate data I/O and job checkpointing. For parallel jobs, three mechanism are supported: MPI, Parallel Virtual Machine (PVM) and its own Master-Worker library. Condor supports fault tolerance with checkpointing and migration. Failed tasks are automatically retried for a number of times. Condor can integrate the resources that are not under its control through Condor-G. Currently, Condor can talk to grid systems (e.g. Globus) and cloud resources (e.g. Amazon EC2) in addition to traditional batch queuing systems (e.g. Portable Batching System (PBS)). Condor daemons can be run on grid resources which join a temporary Condor pool, so that Condor jobs can "glide in" and run on grid resource as if they were native Condor resources. *Directed Acyclic Graph Manager (DAGMan)* is meta-scheduler for Condor that manages dependencies among job through a DAG and schedules and monitors jobs submitted to the underlying Condor system. Overall, Condor is a powerful execution engine for parallel jobs, but it does not pay much attention

to large data applications.

2.4.2 Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC [1, 23], founded by University of California Berkeley, is an open source volunteer computing middleware that harvests the unused CPU and GPU cycles. Several of BOINC-based projects have been created including SETI@home, Predictor@home, and Folding@home. The BOINC resource pool is shared by many projects, so there is always work to be done and thus the whole system is kept busy and fully utilized. Hundreds of thousands computers are contributed, which provides nearly three petaflops processing power. BOINC provides support for redundant computing to identify and reject erroneous results (e.g. caused by mal-functioning computers, or malicious users).

2.5 Parallel Programming Languages

High level programming languages with native support of parallel data processing can lower the entry barrier to parallel programming. Programmers can directly utilize the constructs provided by these languages to achieve various types of parallelism.

2.5.1 Sawzall

Sawzall [90] is a procedural programming language that supports query and aggregation of data distributed over hundreds or thousands of computers. Sawzall exploits the combined computing capability of multiple machines where data reside. Input data are organized as records. Sawzall splits computation into two phases - filter and aggregation. In filtering phase, an operation is applied to each record and intermediate values are emitted. In aggregation phase, all intermediate values are collected and a "reduction" operation is applied to produce the final results. The natively supported aggregators include collection, sampling, summation, maximum, quantile, top and unique. Developers can define new aggregators and plug in them to Sawzall runtime. Aggregators are shared among all tasks in the same job. In addition, an aggregator can be indexed where Sawzall runtime automatically creates individual aggregators for each index. For commutative and associative operations, the order of record processing and aggregation is not important, based on

```
1  count: table sum of int;
2  sum: table sum of float;
3  ele: float = input;
4  emit count <- 1;
5  emit sum <- x;
6  emit mean <- sum / count;
```

Figure 2.6: Sawzall example

which Sawzall makes additional optimization (e.g. coalesce intermediate data). Sawzall is built upon existing Google infrastructure such as Google File System and MapReduce and has been used by Google to process their log files.

Fig. 2.6 shows a simple Sawzall example that calculates the mean of all the float-point numbers in input files. Lines 1-2 declare three aggregators which are marked explicitly by keyword *table*. Their aggregator types are both *sum* which adds up the values emitted to it. Each record in input is converted to type *float* and stored in the variable *ele* declared in line 3. The *emit* statement in line 4 sends value 1 to the aggregator *count* whenever a record is encountered. The *emit* statement in line 5 sends each data value to the aggregator *sum*. So after all records are processed, variable *count* stores the number of records and variable *sum* stores the sum of all records. The mean is calculated in line 6.

2.5.2 Hive

Hive [109, 108] is an open source data warehousing platform built on top of Hadoop. It defines Hive Query Language (HiveQL) which is a Structured Query Language (SQL)-like declarative language. Hive adopts the well-understood concepts in databases such as tables, rows, columns and partitions. In additional, HiveQL allows users to plug their MapReduce programs into HiveQL queries so that complex logic can be directly expressed with MapReduce paradigm. The system architecture is shown in Fig. 2.7. HiveQL programs are compiled into MapReduce jobs that run in Hadoop. Hive also provides a system catalog *Metastore* where data schemas and statistics are stored. The information stored in *Metastore* can be used for data exploration, query optimization and query compilation. Hive has been used intensively in Facebook for reporting and ad-hoc data analysis.

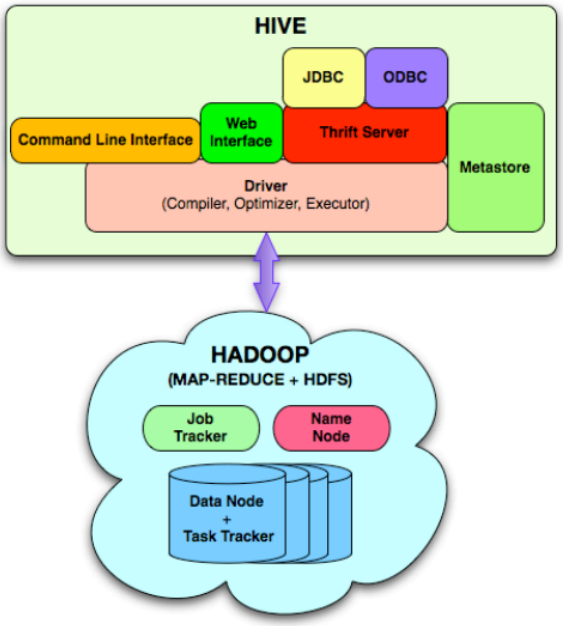


Figure 2.7: HIVE system architecture

* Copied from [108]

2.5.3 Pig Latin

Pig Latin [88] is a data processing language proposed by Yahoo!. It was designed based on the observation that i) SQL is declarative and thus unnatural to procedural programmers; ii) MapReduce model is too low-level and the application code is hard to maintain and reuse (e.g. multiple jobs may need to be pipelined for a single query). Pig Latin tries to find the sweet point between declarative SQL and procedural MapReduce. Each Pig Latin program comprises a sequence of steps each of which is expressed in SQL-like syntax and carries out a single high-level data transformation. A Pig Latin program acts like an execution plan as a whole while the runtime can parallelize the execution of each data transformation. The execution of different steps can be reordered to maximize performance. The supported data transformation includes per-tuple processing, filtering, grouping, join, aggregation and so on. Pig Latin programs are converted to MapReduce jobs which run on Hadoop in parallel. With *load* operation, input data are deserialized into Pig's data model. Output data are serialized with *store* operation. Pig Latin supports a nested data model that allows developers to write UDFs based on their concrete computation requirements.

2.5.4 X10

X10 [34], a language developed by IBM, is designed specifically for parallel programming based on PGAS model. X10 is built on the foundation of Java, but overcomes its lack of lightweight and simple parallel programming support. It is intended to increase programming productivity for Non Uniform Cluster Computing (NUCC) without sacrificing performance. Main design goals are safety, analyzability, scalability and flexibility. X10 is a type-safe object-oriented language with specific support of higher performance computation over dense and sparse distributed multi-dimensional arrays. X10 introduces dynamic, asynchronous activities as fundamental concurrency constructs, which can be created locally or remotely. Globally Asynchronous Locally Synchronous (GALS) semantics is supported for reading and writing mutable data.

2.6 Workflow

2.6.1 Grid workflow

A workflow comprises a sequence of steps concatenated through data or control flow. A workflow management system defines, manages and executes workflows on distributed resources. It allows users to build dynamic applications that orchestrate distributed resources that may span multiple administrative domains. Some workflow management systems have been proposed and developed such as Pegasus [49], Taverna [87], and Kepler [22] with different features. A taxonomy of workflow is presented in [115] which categorizes workflow management systems based on five elements: a) workflow design, b) information retrieval, c) workflow scheduling, d) fault tolerance, and e) data movement. Workflow in grid systems is discussed in [54]. Workflow management systems should quickly bind workflow tasks to the appropriate compute resources. The efficiency also depends on the data movement mechanisms between tasks. Data is moved either via a data transfer service or a script, or through data channels directly between the involved tasks. Most workflow management systems assume that workflow tasks are long running and data movement cost is negligible. However, for data-intensive applications that process extreme volumes of data, to move data around is performance prohibitive. When composing workflows, users need to pay special attention to data locality.

2.6.2 MapReduce workflow

For complex problems, a single two-phase MapReduce job is insufficient and multiple jobs need to be constructed and connected. Oozie [9] is an open source workflow service managing jobs for Hadoop. It supports Streaming, MapReduce, Pig and HDFS jobs. Users use a DAG to express jobs and their dependency relationship. Supported flow control operations include chaining, fork, join and decision. Because cycles are not supported, for-loop like iteration cannot be expressed if the number of iterations cannot be determined statically. Oozie is transactional in that it provides fault tolerance support (e.g. automatic and manual retry). HyMR is a workflow management system proposed by us that integrates MapReduce framework Hadoop and iterative MapReduce framework Twister. The key observation is that Hadoop is good at fault tolerance and data sharing while Twister achieves better performance for iterative applications but lacks a data sharing mechanism and fault tolerance. HyMR combines the best of both worlds. HDFS is used to share data across different jobs in a workflow. Twister is used to run iterative applications while Hadoop is used to run regular MapReduce applications. See section 7.2 for more details.

3

Performance Evaluation of Data Parallel Systems

In this chapter, we present the extensive experiments conducted to measure the efficiency of Hadoop and various storage systems including local IO subsystem, NFS, HDFS and Swift. The results obtained in this chapter will provide valuable insights for optimizing data parallel systems for data-intensive applications.

3.1 Swift

Swift is a new project under the umbrella of OpenStack. I would like to discuss its design specifically because it is more recent compared to other well-known systems to be evaluated. Swift is a highly available, distributed, eventually consistent object store. Swift is not a file system and typical Portable Operating System Interface (POSIX) semantics is not used. Swift clients interact with the server using HTTP protocol. Basic HTTP verbs such as PUT, GET and DELETE are used to manipulate data. The main components of Swift architecture are shown below.

- **Proxy Server**: The Proxy Server is the gateway of the whole system. It responds to users' requests by looking up the location of account, object or container and dispatching the requests accordingly. The Proxy Server can handle failures. If an object server becomes unavailable, it routes requests to a

handoff server determined by the ring. All object reads/writes go through the proxy server which in turn interacts with internal object servers where data are physically stored.

- **Object Server** The Object Server is a object/blob storage server that supports storing, retrieval and deletion of objects. It requires file's extended attributes (xattrs) to store metadata along with binary data. Each object store operation is timestamped and last write wins.
- **Container Server** The Container Server stores listings of objects called $container$. The Container Server does not store the location of objects. It only manages which objects are in each container.
- **Account Server** The Account Server manages listings of container servers.
- **Ring** A ring represents a mapping from logical namespace to physical location. Given the name of an entity, its location can be found by looking up the corresponding ring. Separate rings are maintained for accounts, containers, and objects.

Swift supports large objects. A single uploaded object is limited to 5GB by default. Larger objects can be split into segments which are uploaded along with a special manifest file that records all the segments of a file. When downloaded, segments are concatenated and sent as a single object. In our Swift tests below, data size is way larger than 5GB and thus large object support is used.

Swift is designed for the scenarios where writes are heavy and repeated reads are rare. Since a Swift object server has so many files, the possibility of cache hits becomes marginal. Based on these principles, to buffer disk data in memory does not benefit much. So in implementation, in-memory data cache is purged aggressively elaborated below.

- For write operations, cache is purged every time 512MB data are accumulated in cache.
- For read operations, cache is purged every time 1MB data are accumulated in cache

So for the same amount of data, read operations incur many more cache purge calls than write operations (512x). The additional function calls result in higher overhead. One result of cache purge is that any read may incur data fetch from physical disk with high possibility.

- For write operations, data are first written to memory, accumulated, and then flushed to disk later. So optimizations for bulk copy from memory to disk can be used.
- For read, each operation likely touches physical disks.

3.2 Testbeds

FutureGrid is an international testbed supporting new technologies at all levels of the software stack. The supported environments include cloud, grid and parallel computing (HPC). Table 3.1 summarizes the currently supported tools.

PaaS	Hadoop, Twister, ...
Iaas	Nimbus, Eucalyptus, ViNE, OpenStack, ...
Grid	Genesis II, Unicore, SAGA, ...
HPC	MPI, OpenMP, ScaleMP, PAPI, ...

Table 3.1: FutureGrid software stack

Our performance evaluation has been carried out on FutureGrid clusters. Table 3.2 lists the basic specifications of all FutureGrid clusters. Most of the experiments below were conducted in Bravo whose detailed specification is shown in Table 3.3. Unless stated specifically, Gigabit Ethernet is used.

3.3 Evaluation of Hadoop

Firstly, we evaluate the performance of Hadoop. Hadoop has many parameters that can be tuned. In our evaluation, three main parameters below are considered:

- the size of input (denoted by D)
- the number of nodes (denoted by N)
- the number of map slots per node (denoted by S). It restricts the maximum number of concurrently running tasks on a node.

Name	System Type	# Nodes	# CPUs	# Cores	TFlops	RAM(GB)	Storage(TB)	Site
india	IBM iDataPlex	128	256	1024	11	3072	335	IU
sierra	IBM iDataPlex	84	168	672	7	2688	72	SDSC
hotel	IBM iDataPlex	84	168	672	7	2016	120	UC
foxtrot	IBM iDataPlex	32	64	256	3	768	0	UF
alamo	Dell Power Edge	96	192	768	8	1152	30	TACC
xray	Cray XT5m	1	168	672	6	1344	335	IU
bravo	HP Proliant	16	32	128	1.7	3072	60	IU

Table 3.2: FutureGrid clusters

(copied from https://portal.futuregrid.org/manual/hardware)

Machine Type	Cluster
System Type	HP Proliant
CPU type	Intel Xeon E5620
CPU Speed	2.40GHz
Number of CPUs	128
Number of nodes	16
RAM	192 GB DDR3 1333Mhz
Total RAM (GB)	3072
Number of cores	128
Operating System	Linux
Tflops	1.7
Hard Drives	6x2TB Internal 7200 RPM SATA Drive
Primary storage, shared by all nodes	NFS
Connection configuration	Mellanox 4x DDR InfiniBand adapters

Table 3.3: Specification of Bravo

(copied from https://portal.futuregrid.org/hardware/bravo)

The first metric we choose is job turnaround time T which is the wall-clock time between the time when a job is submitted and the time when it completes. Absolute job turnaround time cannot reveal the efficiency of scaling out. When the number of nodes or map slots per node is increased, performance is expected to improve. Given a fixed amount of work to do, ideally the speedup of T should be $S \cdot N$ compared with the baseline configuration where S and N are both 1. But the efficiency of scaling-out may be low in practice. For example, it is possible that job turnaround time is reduced by 2 times while number of nodes is increased by 10 times. So we normalize absolute job turnaround time to calculate parallel efficiency.

Let I_1 denote the normalized job turnaround time with respect to the number of nodes (i.e. I_1 is the time taken to process one unit of data using one node). I_1 can be calculated with (3.1).

$$T = \frac{I_1 \cdot D}{N} \quad \Rightarrow \quad I_1 = \frac{T \cdot N}{D} \tag{3.1}$$

We know that $I_1 = T \cdot N/D$ given a fixed S. Now we take S into consideration. Let I_2 denote the normalized job run time with respect to both the number of nodes and the number of slots per node. So I_2 is the time taken to process one unit of data on one node by one map task. We assume all map slots are utilized during processing. I_2 can be calculated using (3.2).

$$I_1 = \frac{I_2}{S} \quad \Rightarrow \quad I_2 = I_1 \cdot S = \frac{T \cdot N \cdot S}{D} \tag{3.2}$$

In our tests, we varied the number of slave nodes between 10, 20, 30 and 40; varied the number of map slots per node between 1, 4, 8, 10, 16, 32 and 64; and varied the size of input data between 100GB, 200GB, 300GB and 400GB.

Firstly, the size of data was fixed to 400GB. Table 3.4 shows both job turnaround time and speedup with respect to N and S. For speedup calculation, the test where S was 1 and N was 10 was chosen as the baseline reference configuration. Then we fixed N to 20 and varied S and D, Table 3.5 shows the results. These raw values are hard to interpret. We visualize and analyze the impact of different factors below in detail.

Scheduling	S	**Time** (num. of nodes is varied)				**Speedup** (num. of nodes is varied)			
		40	30	20	10	40	30	20	10
default	1	1431†	1904	2852	5711	4.0	3.0	2.0	1.0*
	4	608	787	1180	2554	9.4	7.2	4.8	2.2
	8	422	463	756	1868	13.5	12.3	7.6	3.0
	10	331	434	699	1594	17.2	13.1	8.2	3.6
	16	261	354	572	1187	21.9	16.1	10.0	4.8
	32	222	259	440	988	25.7	22.0	13.0	5.8
	64	296	294	443	955	19.3	19.4	12.9	6.0
0.5 random	1	1460	1910	2856	5739	3.9	3.0	2.0	1.0
	4	667	880	1298	2686	8.6	6.5	4.4	2.1
	8	481	612	931	1822	11.9	9.3	6.1	3.1
	10	399	530	787	1643	14.3	10.8	7.2	3.5
	16	305	382	548	1193	18.7	14.9	10.4	4.8
	32	226	254	394	1018	25.3	22.5	14.5	5.6
	64	281	288	408	1040	20.3	19.8	14.0	5.5
random	1	1459	1905	2853	5754	3.9	3.0	2.0	1.0
	4	706	957	1370	2761	8.1	6.0	4.2	2.1
	8	506	645	1026	1932	11.3	8.9	5.6	3.0
	10	440	557	836	1869	13.0	10.2	6.8	3.0
	16	328	418	634	1231	17.4	13.7	9.0	4.6
	32	245	271	461	1107	23.3	21.1	12.4	5.2
	64	299	278	534	1050	19.1	20.5	10.7	5.4

Table 3.4: Job run time w/ S and N varied (fixed 400GB input)

*: Baseline system configuration: $S = 1$, $N = 10$
†: Time is in second

3.3.1 Job run time w.r.t the num. of nodes

We increased the number of nodes, and measured the job run time of different scheduling algorithms with S set to 8 and 32. Fig. 3.1 shows the results. All job run time is normalized against the reference test where S is 8, N is 10 and default scheduling is used.

default				0.5 random				random			
S	400G†	100G‡	Slowdown*	S	400G	100G	Slowdown*	S	400G	100G	Slowdown*
1	5711	1329	4.29	1	5739	1327	4.32	1	5754	1321	4.35
4	2554	538	4.74	4	2686	582	4.61	4	2761	625	4.41
8	1868	315	5.93	8	1822	404	4.50	8	1932	438	4.41
10	1594	277	5.75	10	1643	360	4.56	10	1869	370	5.05
16	1187	231	5.13	16	1193	248	4.81	16	1231	262	4.69
32	988	151	6.54	32	1018	207	4.91	32	1107	185	5.98
64	955	138	6.92	64	1040	111	9.36	64	1050	168	6.25

Table 3.5: Job turnaround time w/ S and D varied

†: time taken to process 400GB data (in seconds)

‡: time taken to process 100GB data (in seconds)

* Given a fixed S, slowdown = $\frac{time\ to\ process\ 400GBdata}{time\ to\ process\ 100GBdata}$

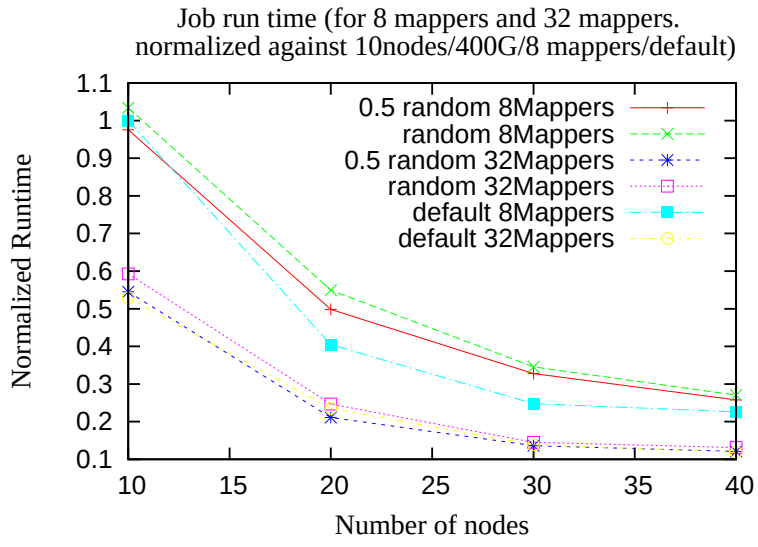


Figure 3.1: Job run time with the number of nodes varied

- As we expect, the job run time is decreased as we increase number of nodes. But the relationship is not

linear. The performance gain becomes less and less significant as we add more and more nodes. Eventually, the reduction of job run time becomes marginal and using more resources gets cost prohibitive.

- Default scheduling has highest speedup. Random scheduling has lowest speedup. 0.5 random falls in between.

- Running 32 mappers per nodes yields much better performance than 8 mappers per node.

3.3.2 Job run time w.r.t the number of map slots per node

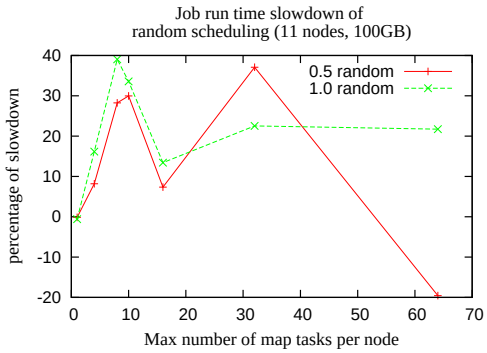
Theoretically, increasing S is a double-edged sword. On the one hand, it increases concurrency by P times. On the other hand, it increases overhead by Q times. P and Q need to be evaluated through experiments. The final outcome depends on the effects of above two factors. If $P > Q$, it decreases job run time. If $P < Q$, it increases job run time. Speedup is P/Q. Fig. 3.2 shows job run time with S varied between 1, 4, 8, 10, 16, 32, and 64. We summarize our observations:

- The relationship is NOT linear. Ideally, job run time should be inversely proportional to S. But the plots show this is not the case.

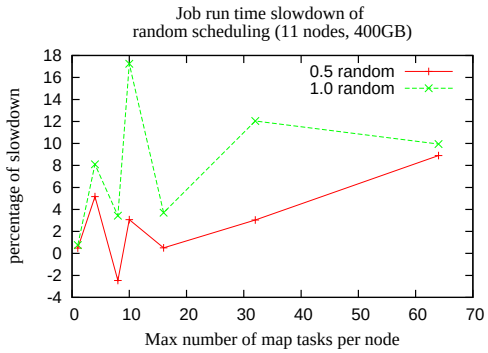
- When S is small, increasing S can bring significant benefit so that job run time is decreased drastically. This is because increasing S also improves concurrency so that the overlap between computation and IO, and the processing power of the multi-core processors can be better explored. The benefit of higher concurrency exceeds overhead.

- When S becomes big (more than 32), job run time does not change much. For some tests, to increase S from 32 to 64 even results in the increase of job run time. This means the contention of resource use becomes comparable to the benefit of higher concurrency.

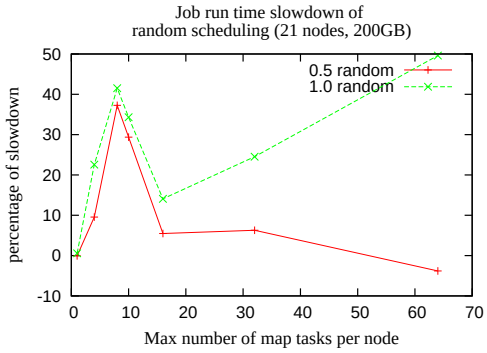
- There is a specific value of S that maximizes the speedup. The concrete value depends on both hardware and applications workload. In an environment where the workload of submitted jobs is quite diverse, to find the optimal S is non-trivial which may need to be adjusted dynamically.



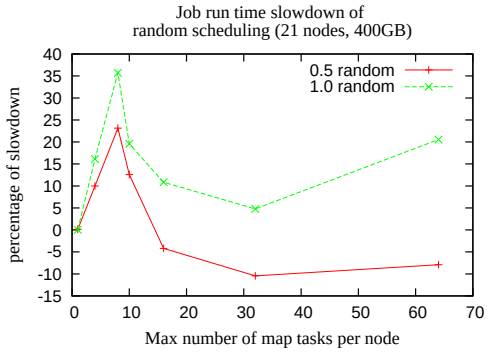
(a) 11 nodes, 100GB data



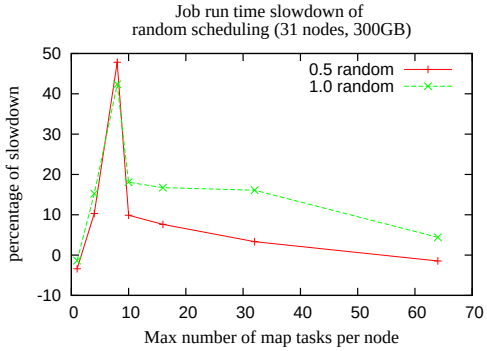
(b) 11 nodes, 400GB data



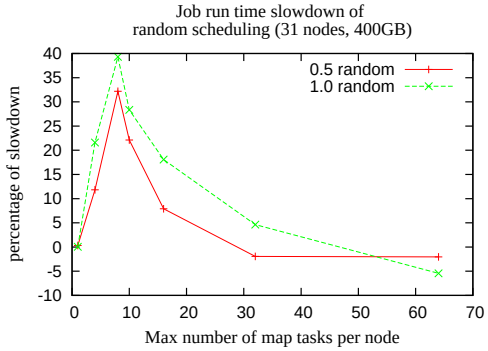
(c) 21 nodes, 200GB data



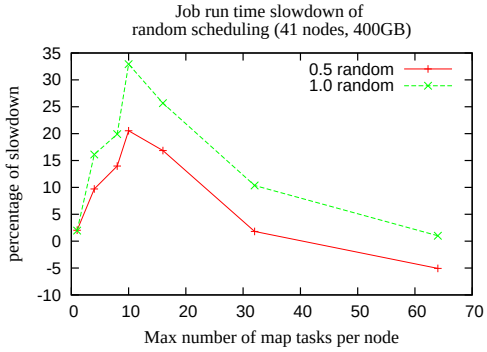
(d) 21 nodes, 400GB



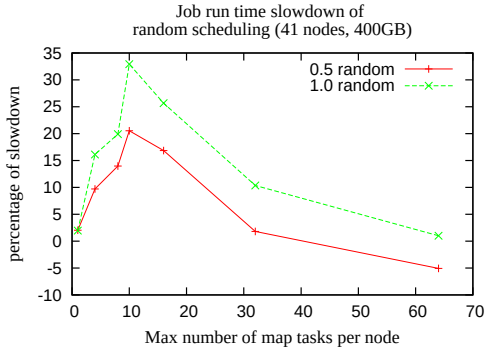
(e) 31 nodes, 300GB data



(f) 31 nodes, 400 data



(g) 41 nodes, 400GB data



(h) nodes, data

Figure 3.2: Job run time for different configurations

- Random scheduling performs worse than default scheduling.

Besides absolute job turnaround time, we also calculate parallel efficiency. We have the following observations regarding normalized job run time I_1 (to eliminate the factor of N):

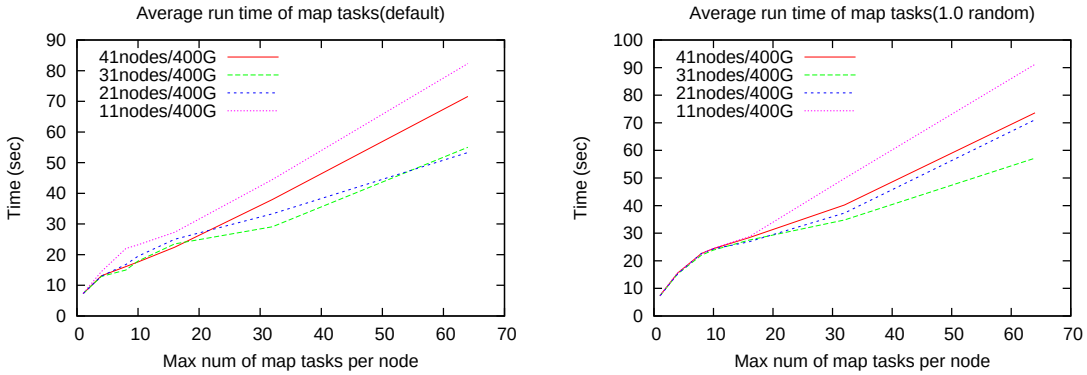
- When S is small (e.g. between 1 and 4) the size of input data does not have significant impact on I_1.
- When S becomes big, the more data is processed, the bigger I_1becomes.
- When S is small, no matter how many nodes the system has and how many data are processed, I_1 is pretty close. As S is increased, the difference between I_1 with different input data sizes is increased.

In addition, following observations have been made by us for normalized job run time I_2 (to eliminate the factors of N and S)

- I_2 increases monotonically as S is increased. This means increasing concurrency by increasing S always results in efficiency deterioration.
- In the worst case, I_2 is 10x slower than ideal speedup. When S is small, no matter how many nodes the system has and how many data are processed, I_2 is pretty close. As S gets larger (beyond 16), different system factors including the number of nodes and the amount of input data result in more diverse I_2.

3.3.3 Run time of map tasks

Given a fixed number of nodes, the larger the input data is, the longer each map task executes. This is obvious because each task processes more data. Fig. 3.3 shows the average run time of map tasks for default and 1.0 random scheduling. As S is increased, both mean and standard deviation (not shown in the plots) of the run time of map tasks increase monotonically. So one consequence of increasing S is the increase of task run time, which is caused by the increasing resource contention. From the plots, we can see the run time is approximately linear with S when S becomes larger than 8. However, the slope is different for varied numbers of nodes.



(a) Default scheduling

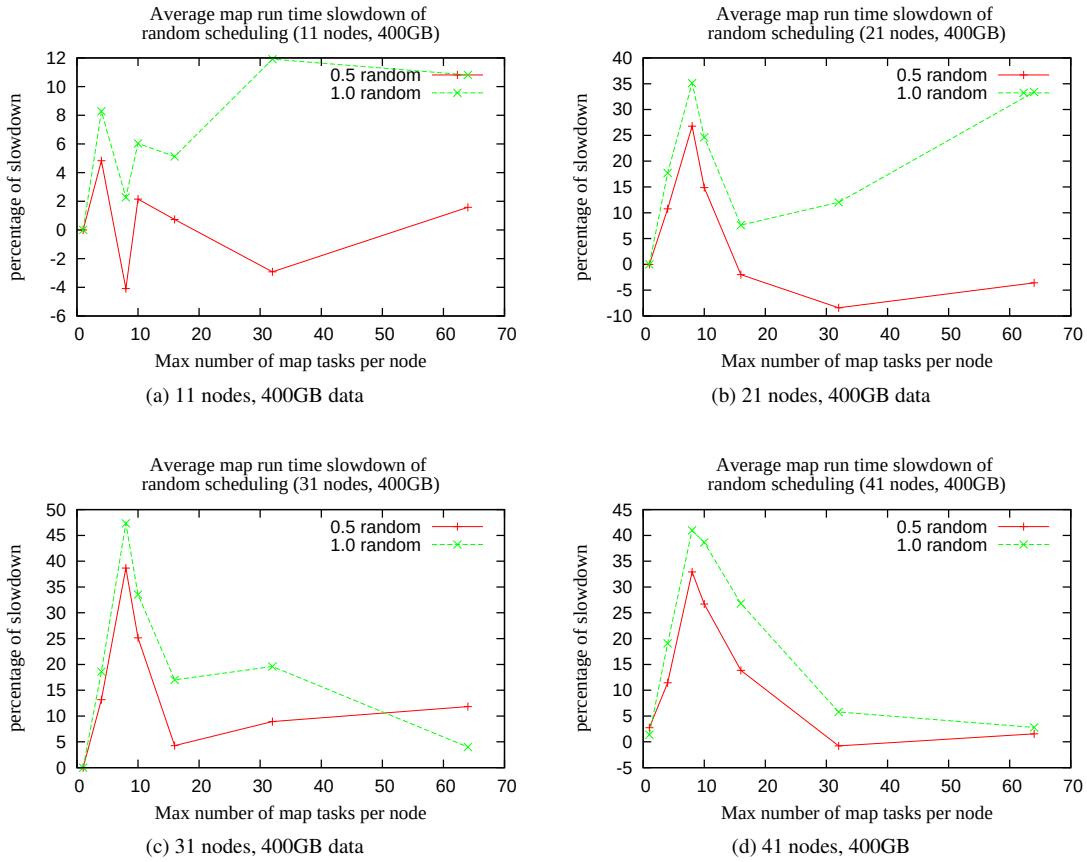
(b) 1.0 random scheduling

Figure 3.3: Average run time of map tasks

Fig. 3.4 shows the slowdown of the average map task run time caused by random scheduling. Hadoop default scheduling is the reference. Overall random scheduling results in longer run time than default scheduling. We observe that when S is increased initially from a small value, random scheduling increasingly slows down the execution of map tasks substantially. When S reaches 8, the slowdown is maximized mostly. Beyond that point, the slowdown is increasingly improved. 1.0 random performs significantly worse than 0.5 random. For some tests, the slowdown trend of 1.0 random changes drastically and thus is more intractable. In addition, when S is no less than 32, the more nodes there are in the system, the less significant the performance difference becomes. When S is increased, the task scheduler can schedule more tasks at once in one scheduling point because there are more available map slots. As a result, the number of scheduling "waves" is reduced, so does the overhead.

3.4 Evaluation of Storage Systems

Storage systems are critical to the performance of overall data parallel systems because they determine the throughput of reading and writing data. We will evaluate how state-of-the-art storage systems perform including local disks, HDFS [101], Shared NFS [96], and OpenStack Swift [11].



(a) 11 nodes, 400GB data

(b) 21 nodes, 400GB data

(c) 31 nodes, 400GB data

(d) 41 nodes, 400GB

Figure 3.4: Average run time slowdown of map tasks

3.4.1 Local IO subsystem

In this test, we measure how local data accesses perform for sequential read and write operations. Both direct IO (i.e. bypass OS buffer/page cache) and regular IO (i.e. use OS caching) were tested. For direct IO tests, 1GB data was written and read in sequential manner. For direct IO, buffer size was set to 512 bytes which is aligned to the disk block size. This setting was mandatory for the tool we used. For regular IO tests, 400GB data was written to and read from a local disk in sequential manner, and buffer size was set to 1MB. Each Bravo node has roughly 200GB memory. Thus, the data size (i.e. 400GB) was twice the amount of memory, so that caching effect became negligible. Results are shown in Table 3.6.

Observations: Regular IO achieved significantly higher throughputs than direct IO did. IO throughput

	Direct IO (buf size is 512B)			Regular IO with OS Caching		
operation	**size(GB)**	**time**	**io-rate**	**size(GB)**	**time**	**io-rate**
seq-read	1	77.7sec	13.5MB/s	400	1059sec	386.8MB/s
seq-write	1	103.2sec	10.2MB/s	400	1303sec	314MB/s

Table 3.6: IO performance of local disks

was degraded by 30 times for direct IO. One influential factor is buffer size. To temporarily cache data allows bulk sync-up. In addition, for regular IO writing data to memory (done by applications) and syncing dirty cached pages to disk (managed by OS) can overlap to some extent. Another observation is that write throughput is lower than read throughput by 24% and 19% for direct IO and regular IO respectively.

3.4.2 NFS

User home directory in Bravo resides on a mounted NFS. The exact same tests as section 3.4.1 were run. Results are shown in Table 3.7.

	Direct IO (buf size is 512B)			Regular IO with OS Caching		
operation	**size(GB)**	**time**	**io-rate**	**size(GB)**	**time**	**io-rate**
seq-read	1	366sec	2.8MB/s	400	3556sec	115.2MB/s
seq-write	1	2688sec	390KB/s	400	3856sec	106.2MB/s

Table 3.7: IO performance of NFS

Observations: As we expect, without the benefit of in-memory caching, direct IO performs much worse than regular IO. Read and write throughput were degraded by 97% and 99% respectively. Communications between NFS clients and server are necessary to sync-up metadata and transfer real data. So its throughput is limited by the capability of network devices and structure. In our tests, Gigabit Ethernet was used, so the maximal throughput was 125 MBytes/s. The throughput of regular IO is close to this limit. For direct IO, not only network throughput but also latency are important.

Compared with local IO subsystem, the throughputs of NFS are much lower. For direct IO, read and write throughputs were degraded by 79% and 96% respectively, while for regular IO read and write throughputs

were degraded by 70% and 66%.

3.4.3 HDFS

In this test, we intend to reveal the additional overhead added by the HDFS implementation. We ran a two-node HDFS cluster in Bravo: one namenode and one data node. Local disks were used by the HDFS data node to store raw data. Replication factor was set to 1. 400GB data was written to and read from HDFS. For write operations, in-memory data source */dev/zero* was used which did not incur disk access for data reading; and for read operations, data sink was */dev/null*. HDFS client was run on the only data node so that all data accesses (except metadata requests) were local through HDFS APIs. Table 3.8 shows the results.

operation	**size(GB)**	**time**	**io-rate**
seq-read	400	3228sec	126.9MB/s
seq-write	400	3456sec	118.6MB/s

Table 3.8: IO performance of HDFS

Observations: Firstly, read and write throughputs are comparable. Secondly, comparing the results with local IO tests (Table 3.6), we can see that HDFS can only achieve roughly 1/3 of the throughput of direct local disk access. Because all data accesses were local in both tests, we can conclude that the additional overhead was brought by HDFS software stack. It implies the version of Hadoop we used is not well optimized or tuned and we expect Hadoop to perform comparably to local IO. One possible "culprit" is the excessive number of memory copying among different Hadoop modules (e.g. encryption module, checksum module, stream module). Another reason is that data accesses in HDFS go through TCP/IP stack no matter whether it is located locally. This eases the implementation, but incurs performance penalty. We believe the performance degradation will catch some attention in the community and be improved over time in future releases.

3.4.4 OpenStack Swift

We deployed a 8-node Swift system (not publicly available yet) in Bravo. It comprised 1 proxy node and 7 storage nodes.

3.4.4.1 Single cluster:

We ran swift client on Bravo to write a 400GB object into Swift and read it back. The segment size was set to 64MB (the same as HDFS block size). For write operations, data source is */dev/zero*. For read operations, data sink is */dev/null*. Both client and service were run in Bravo. Results are shown in Table 3.9.

operation	**size(GB)**	**time**	**io-rate**
seq-read	400	10723sec	38.2MB/s
seq-write	400	11454sec	35.8MB/s

Table 3.9: IO performance of Swift (single cluster)

Observations: Both read and write throughputs are way below the theoretical maximum (i.e. 125MB/s). Because all traffic goes through the proxy server, the slowdown was possibly caused by the inefficiency of the proxy server. If the data transfer between the client and the proxy server and that between the proxy server and the object server are well pipelined, the throughput should be close to 100MB/s at least given the number of transferred segments is large (400GB/64MB = 6400). We conclude that Swift is still in early phase and more optimizations of data transfer are necessary to achieve satisfactory performance.

3.4.4.2 Multiple clusters:

We ran Swift client in FutureGrid cluster Foxtrot deployed in University of Florida. So all data accesses were cross-cluster. We compared the cases where Virtual Machines (VMs) and physical nodes are used. The same tests as above were run and results are shown in Table 3.10.

		Foxtrot Nimbus VM (Infiniband)		HPC in Foxtrot (InfiniBand)	
data size	operation	time	io-rate	time	io-rate
400GB	seq-read	31128sec	13.16MB/s	14647sec	30.0MB/s
400GB	seq-write	8025sec	51.04MB/s	4483sec	91.4MB/s

Table 3.10: IO performance of Swift (cross-cluster)

Observations:

1. InfiniBand can improve write throughput by 2x - 4x while read throughput stays the same. So for write operations network bandwidth of the proxy server (there was only one in our tests) is the bottleneck.

2. Using VMs degraded the performance significantly by 57% and 44% for read and write operations respectively, compared with the case where physical nodes were used. Using VM yielded higher write throughput but lower read throughput compared with single-cluster accesses with Gigabit Ethernet.

3. One interesting observation is that write is faster than read for all tests. One possible cause is the different cache purge policies for read and write operations as discussed in section 3.1.

3.4.5 Small file tests

In this test, we created many files, wrote a small amount of data (1MB) into each file and read the data back. Table 3.11 shows the results.

	System	**Num. of files**	**File size(MB)**	**Time**
Create & Write	**Shared NFS**	1000	1	37.8 sec
	Local Disk	1000	1	5.5sec
	HDFS	1000	1	18.0sec
Read	**Shared NFS**	1000	1	7.9sec
	Local Disk	1000	1	4.5sec
	HDFS	1000	1	13.4sec

Table 3.11: IO performance for small files

Observations: Local IO subsystem is the most efficient. The creation of new files in NFS incurs the most overhead which is 6 times and 2 times of that of local IO and HDFS respectively. So users should avoid creating lots of small files in NFS. For reading of small files, HDFS performs the worst, which matches the results of above HDFS tests. Again further optimization of the data access path in HDFS is needed to make it perform efficiently. Overall, distributed storage systems yield at least 2x slowdown compared to local storage.

4

Data Locality Aware Scheduling

Data locality measures how close compute and its input data are. The "proximity" is related to both network infrastructure and scheduling strategy. In data-intensive computing, the amount of processed data is huge and to move data around overwhelms the limited network resources and thus is quite inefficient. Traditioinal HPC frameworks focus on compute scheduling and do not consider data affinity, while data parallel systems natively integrate data locality into task scheduling.

For typical hierarchical network configuration used in clusters, nodes and network devices are organized in a tree-like structure depicted in Figure 4.1. Nodes are placed in racks and a top-of-rack switch (edge switch) usually comes with each rack. Those switches are connected to aggregate switches which are in turn connected to core switches. As you may have observed, the number of switches drastically decreases when walking up the tree from bottom to top. The switches at high layers are much more powerful (and therefore expensive) than those at low layers. Most of the data centers introduce oversubscription to lower the cost, which means the aggregate throughput of top layers is lower than that of bottom layers. This design has the assumption that the worst case is rare that all nodes use network at full speed simultaneously. This assumption may break for modern clusters. Firstly, if applications are data intensive, they process huge amounts of data in parallel. Concurrent data staging imposes significant load on the network infrastructure. Secondly, current trend is individual compute nodes have more and more cores and processors, and this enables more tasks to run concurrently on each node. Therefore network traffic is increased on average. As a result, in a busy cluster, network oversubscription may result in the significant fluctuation of task execution depending on where input data are stored.

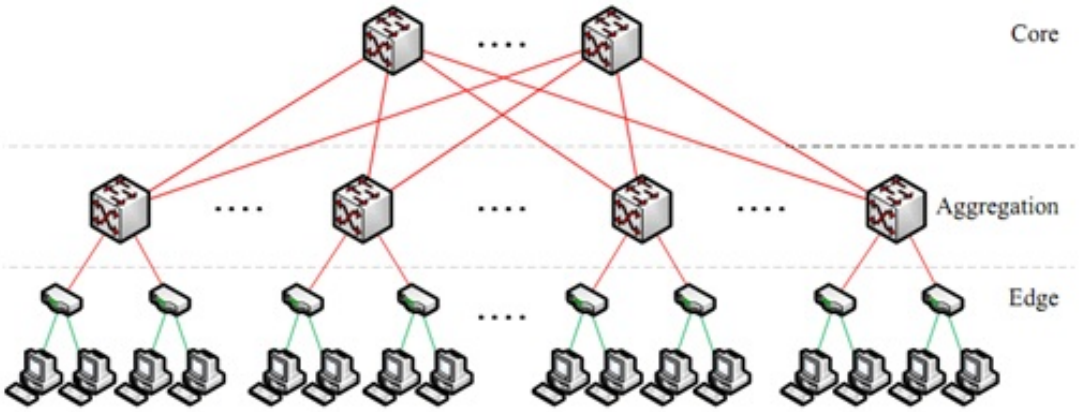


Figure 4.1: Typical network topology

* Copied form [51]

Besides the topology described above, some other topologies such as Hypercube, Mesh, Torus [86] and FAT tree [78] have been proposed for different purposes. However, in modern clusters, hierarchical structure is the most prevalent way to interconnect computer nodes.

4.1 Traditional Approaches to Build Runtimes

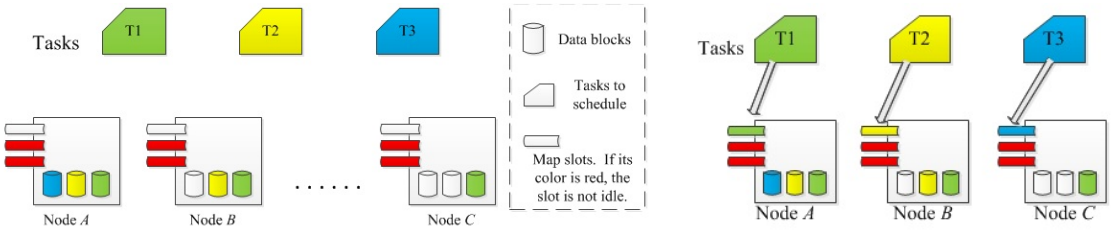
To simplify data accessing, distributed file systems which allow access to files from multiple hosts via network have undergone intensive research. Usually a unified namespace is provided to ease data management. Traditionally the transparency of data accesses is favored and developers should not care about whether data are stored locally or on a remote node. NFS [96], Parallel Virtual File System (PVFS) [30], Lustre [99] and GPFS [97] are good examples. Usually these file systems run on dedicated storage hardware (e.g. Network Attached Storage) and are mounted to compute nodes. Files are striped across disks, which balances load on disks and enables parallel IO to fully exploit the throughput of underlying disk subsystems. They are compatible with POSIX APIs and look like regular local file systems in functionalities to end users. Although they make the lives of developers easier because the details of underlying data placement are hidden, there are various drawbacks for this approach. Firstly, local accessing and remote accessing are different in terms of latency and throughput so that the same API call may exhibit drastically different responsiveness depending

upon where the data are located. The variation may be unexpected to developers and therefore breaks their assumptions about the runtime environments. Performance tends to deteriorate and inconsistent results may even be generated. Secondly, distributed computing runtimes cannot integrate storage affinity into scheduling as the lack of low level data management information. So data staging is inevitable to move data to compute mostly. For clusters with high-profile hardware that run non-data intensive applications, data-staging approach may work well. However, as data size keeps increasing, to move data around is quite inefficient and degrades overall performance significantly because of severe network usage contention.

4.2 Data Locality Aware Approach

Data Parallel Systems, which are natively designed for data intensive applications, adopt a different approach which sacrifices transparency for performance. In data parallel systems, the location of data is exposed by storage systems to upper-level runtime frameworks so that data-locality aware scheduling can be achieved. Data locality is one significant advantage of data parallel systems over traditional HPC systems. It brings more flexibility to scheduling algorithms. For example, the scheduler can bring data to compute, bring data close to compute or bring compute to data. It makes in situ processing possible. For GFS/HDFS, large files are split to equally-sized blocks. MapReduce explores embarrassing parallelism by constructing one task for each block in map phase and running them independently. Map tasks do not communicate with each other. MapReduce scheduler is data-locality aware and co-locates data and compute with best efforts. Figure 4.2 shows an example of how MapReduce schedules tasks. Figure 4.2a shows the state of a system at a specific time instant. There are three map tasks T_1, T_2 and T_3, and three nodes (A, B and C) with idle map slots. Each data block has multiple replicas and each node has three map slots among which those marked as red are not idle. If a data block B is marked with the same color as a task T, it means B is the input data of T. Only the nodes that have idle slots are shown. So the input data of task T_1 are stored on nodes A, B, and C. The input data of task T_2 are stored on nodes A and B. The input data of task T_3 are stored on node A. Figure 4.2b shows an example of scheduling. Node A has one idle map slot and it hosts the input data of task T_1, so T_1 is scheduled to A. Node B has one idle map slot and hosts the input data of task T_2, so T_2 is scheduled to B. The only node that has idle map slots is C where task T_3 has to be scheduled. T_3 needs to fetch input data from node A. As a result, tasks T_1 and T_2 access input data locally and only task T_3 accesses

data remotely. The assignment of map tasks saves 2/3 of network traffic compared with the strategy where all data are staged in. Data parallel systems with good data locality can reduce network usage of tasks and therefore host more applications running concurrently without drastic performance degradation.



(a) Initial state (b) Default scheduling

Figure 4.2: An example of MapReduce scheduling

The fact that data parallel systems enable data-locality aware scheduling does not gaurantee satisfactory data locality. If scheduling algorithms in data parallel systems cannot achieve good data locality, they may degenerate to traditional scheduling where data are always moved around. To further study data locality, one needs to resolve the following issues:

1. What is the relationship between system factors (e.g. the number of nodes, the number of tasks) and data locality?
2. How data locality impacts the execution time of jobs in single-cluster and cross-cluster environments?
3. Does the default Hadoop scheduling strategy yields optimal data locality? If it is not optimal, what is the optimal scheduling algorithm and how does it outperform the default Hadoop scheduling algorithm?

4.3 Analysis of Data Locality In MapReduce

4.3.1 Data Locality in MapReduce

In GFS/HDFS, files are split into equally-sized blocks which are placed across nodes. In Hadoop, each node has a configurable number of map and reduce slots, which limit the maximum number of map and reduce tasks that can concurrently run on the node. When a task starts to execute, it *occupies* one slot; and

when it completes, the slot is *released* so that other tasks can take it. Each slot can only have one task assigned at most at any time. There is a single central master node where *Job Tracker* runs. The job tracker manages all slave/worker nodes and embraces a scheduler that assigns tasks to idle slots.

Data locality is defined as how close compute and input data are, and has different levels – node-level, rack-level, etc. In our work, we only focus on the node-level data locality which means compute and data are co-located on the same node. Data locality is one of the most important factors considered by schedulers in data parallel systems. Please note that here data locality means the data locality of input data. Map tasks may generate intermediate data, but they are stored locally (not uploaded to HDFS) so that data locality is naturally gained. We define *goodness of data locality* as the percent of map tasks that gain node-level data locality.

In this thesis, the default scheduling algorithm in Hadoop is denoted by *dl-sched*. In Hadoop, when a slave node sends a heartbeat message and says it has available map slots, the master node first tries to find a map task whose input data are stored on that slave node. If such a task can be found, it is scheduled to the node and node-level data locality is gained. Otherwise, Hadoop tries to find a task that can achieve rack-level data locality – input data and task execution are on the same rack. If it still fails, a task is randomly picked and dispatched. Apparently *dl-sched* favors data locality.

4.3.2 Goodness of Data Locality

Firstly, we develop a set of mathematical symbols to characterize HDFS/MapReduce which are shown in Table 4.1. Data replicas are randomly placed across all nodes, which approximately matches the block placement strategy used in Hadoop. And idle slots are randomly chosen from all slots. This assumption is reasonable for modestly utilized clusters that run lots of jobs with diverse workload from multiple users. In such a complicated system, it is difficult, if not impossible, to know which slots will be released and when. In addition, we assume that I is constant within a specific time frame and may vary across time frames. This assumption implies that new tasks come into the system at the same rate that running tasks complete. So the system is in a dynamic equilibrium state for small time frames. Time is divided into time frames each of which is associated with a corresponding I, which can be done through the clustering of time series subsequences.

Symbols	Description
N	the number of nodes
S	the number of map slots on each node
I	the ratio of idle slots
T	the number of tasks to be executed
C	replication factor
IS	the number of idle map slots ($N \cdot S \cdot I$)
$p(k, T)$	the possibility that k out of T tasks can gain data locality
goodness of data locality	the percent of map tasks that gain data locality

Table 4.1: Symbol definition

Our goal is to study the relationship between the goodness of data locality and important system factors. Obviously, the relationship depends upon scheduling algorithms. The default Hadoop scheduling algorithm *dl-sched* is the target of our analysis here. To simplify the mathematical deduction of closed-form formulas, we assume replication factor C and the number of slots per node S are both 1. Firstly we need to calculate $p(k, T)$ (the probability that k out of T total tasks can achieve data locality). Each task can be scheduled to any of the N nodes, so the total number of cases is $N \cdot T$. Because both the data placement and idle slot distribution are random, we can fix the distribution of idle slots without affecting the correctness of analysis. We simply assume that the first IS slots among all slots are idle. To guarantee that k tasks have data locality, we first choose k idle slots from total IS idle slots to which unscheduled tasks will be assigned, which gives C_k^{IS} cases (shown as step ① in Fig. 4.3). Then we divide all unscheduled tasks into two groups: g_1 and g_2. For group g_1, the input data of all its tasks are located on the nodes that have idle slots. For group g_2, the input data of all its tasks are located on the nodes that have no idle slots. The input data of the tasks in group g_1 need to be stored on k idle nodes so that exact k tasks can achieve data locality (note if the input data of multiple tasks are stored on the same node with only one idle slot, only one task can be scheduled to the node and other tasks will not achieve data locality). Assume group g_1 has i tasks, the number of ways to choose these tasks from total T tasks is C_i^T, and the number of ways to distribute their input data onto k nodes is $S(i, k)$ (stirling numbers of the second kind) (shown as step ② in Fig. 4.3). The number of tasks in group g_2 is T-i and each of them can choose among N-IS busy nodes to store input data, which yields $(N\text{-}IS)^{T\text{-}i}$

cases (shown as step ③ in Fig. 4.3). Combining all above steps, we deduce (4.2) to calculate $p(k,T)$. Then the expectation E can be calculated using (4.4) and the goodness of data locality R can be calculated using (4.4).

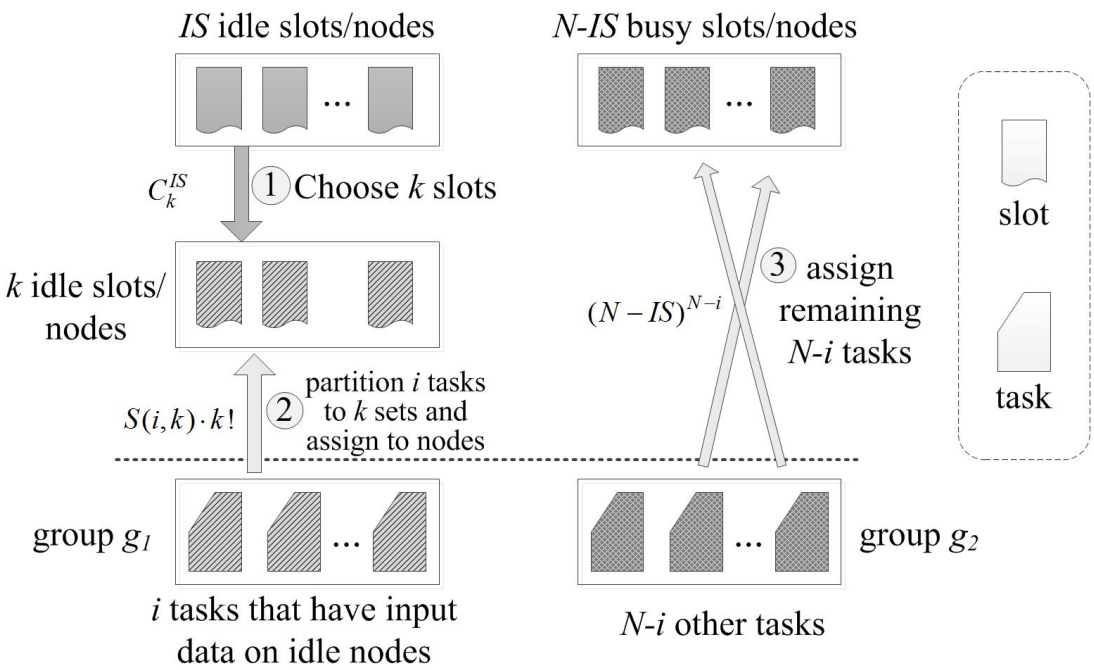


Figure 4.3: The deduction sketch for the relationship between system factors and data locality

$$0 \le k \le 0 \le T \le IS \tag{4.1}$$

$$p(k,T) = C_k^{IS} \cdot \sum_{i=k}^{T} (C_i^T \cdot S(i,k) \cdot k! \cdot (N\text{-}IS)^{T\text{-}i})/N^T \tag{4.2}$$

$$E = \sum_{k=0} T(k \cdot p(k,T)) \tag{4.3}$$

$$R = E/T \tag{4.4}$$

So the goodness of data locality can be accurately calculated. For cases where C and S are larger than one, the mathematical deduction is much more complicated and we are working on it. In our experiments below we take the approach of simulation instead of accurate numerical calculation of (4.4) for two reasons:

a) calculating (4.4) involves factorial and exponential operations requiring enormous computation if operands are large; b) we have not deduced closed-form formulas for the cases where C and S are not 1. Therefore the experiment results below are applicable to the cases where C and S are not 1, although we have not deduced closed-form formulas for them.

4.4 A Scheduling Algorithm with Optimal Data Locality

Here the term optimality means the maximization of the goodness of data locality. Given a set of tasks to schedule and a set of idle slots, if a scheduling algorithm achieves the best data locality, we call it is optimal. We will show that scheduling multiple tasks all at once outperforms the task-by-task approach taken by *dl-sched*.

4.4.1 Non-optimality of *dl-sched*

Fig. 4.4 demonstrates *dl-sched* is not optimal. Fig. 4.4(a) shows an instantaneous state of a system. There are three tasks (T_1, T_2 and T_3) to schedule, and three nodes (A, B and C) that have idle map slots. Each data block has multiple replicas and each node has three map slots among which those marked as black are not idle. If a data block B is marked with the same color and pattern as a task T, B is the input data of T. Only the nodes that have idle slots are shown. From the graph, we can see the input data of task T_1 are stored on nodes A, B, and C; the input data of task T_2 are stored on nodes A and B; and the input data of task T_3 are stored on node A. Fig. 4.4(b) shows an example of *dl-sched* scheduling. Node A has one idle map slot and it hosts the input data of task T_1, so T_1 is scheduled to A. Node B has one idle map slot and hosts the input data of task T_2, so T_2 is scheduled to B. Now the only node that has idle map slots is C and task T_3 must be scheduled there. However, node C does not host the input data of task T_3. To summarize, tasks T_1 and T_2 gain data locality while task T_3 loses data locality. But, we can easily find another way to schedule the three tasks to make all of them achieve data locality, which is shown in Fig. 4.4(c). The reason that *dl-sched* and its variants (e.g. fair scheduling, delay scheduling) are not optimal is that tasks are scheduled one by one and each task is scheduled without considering its impact on other tasks. To achieve a global optimum, all unscheduled tasks and idle slots at hand must be considered at once to make global scheduling decisions.

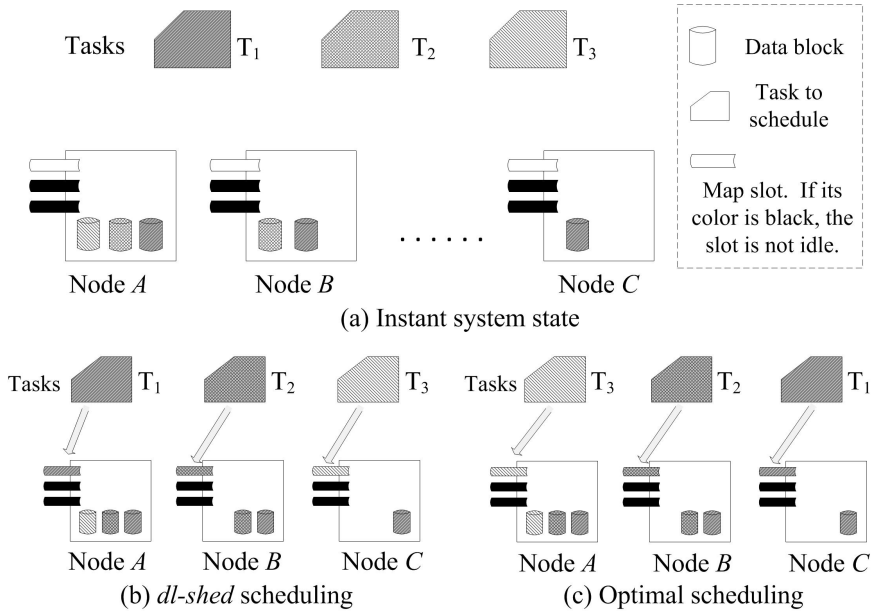


Figure 4.4: An example showing Hadoop scheduling is not optimal

4.4.2 lsap-sched: An Optimal Scheduler for Homogeneous Network

We reformulate the problem into a formal definition using symbols defined in section 4.3.2. The assignment of map tasks to idle slots is defined as function φ. Given task i, $\varphi(i)$ is the slot to which it is assigned. Function φ needs to be injective to guarantee that multiple tasks are not assigned to the same idle slot. We associate an assignment cost to each task-to-slot assignment. Low assignment cost means good data locality and high assignment cost means bad data locality. C_{ij} represents the assignment cost to assign task i to slot j, and is defined in (4.5). If a task is scheduled to a node which stores its input data, its assignment cost is 0. Otherwise, the cost is 1 (in next section, we will calculate the cost for non-local tasks based on the location of compute and input data, rather than use fixed value 1). Basically the cost matrix C measures the data locality of assigned tasks. So it is good for IO intensive jobs and needs to be enhanced for other types of jobs. Given φ, the total assignment cost is the summation of the assignment cost of all scheduled tasks, which is formulated in (4.6). The goal function is shown in (4.7), which tries to find a φ that minimizes the total assignment cost. As we showed, the function φ given by *dl-sched* is not optimal, and thus not a solution

to (4.7).

$$C_{ij} = \begin{cases} 0 & \text{if the input of task } i \text{ and slot } j \text{ are co-located} \\ 1 & \text{otherwise} \end{cases} \tag{4.5}$$

$$C_{sum}(\varphi) = \sum_{i=1}^{T} C_{i\varphi(i)} \tag{4.6}$$

$$g = \arg_{\varphi}\min\{C_{sum}(\varphi)\} \tag{4.7}$$

We found that this problem can be converted to the well-known Linear Sum Assignment Problem (LSAP) [5] shown below. The difference is that LSAP requires that the cost matrix be square. In our case, if T and IS are equal, matrix C is square and we can directly apply LSAP. Otherwise, LSAP cannot be directly applied and we figure out how to convert the problem to LSAP by manipulating matrix C.

Linear Sum Assignment Problem: Given n items and n workers, the assignment of an item to a worker incurs a known cost. Each item is assigned to one worker and each worker only has one item assigned. Find the assignment that minimizes the sum of cost.

If T is less than IS, we make up $IS - T$ extra dummy tasks whose assignment cost is 1 no matter which slots they are scheduled to. Fig. 4.2a shows an example in which t_i and s_j represent tasks and idle slots respectively. The first T rows are from the original cost matrix. The last $IS - T$ rows are for the dummy tasks we make up and filled with constant 1. Now we get a IS x IS square cost matrix and can apply LSAP algorithms to find an optimal solution. LSAP algorithms will give us an optimal assignment for all IS tasks. Among them we just pick those that are not dummy tasks, and we get a specific φ (termed φ-$lsap$) for the original problem. Now let us prove that φ-$lsap$ is a solution to (4.7) by using contradiction.

Proof: The assignment cost given by φ-$lsap$ is $C_{sum}(\varphi\text{-}lsap)$ (see (4.6)). As a result, the total assignment cost given by LSAP algorithms for the expanded square matrix is $C_{sum}(\varphi\text{-}lsap) + (IS - T)$. The key point is that the total assignment cost of dummy tasks is $IS - T$ no matter where they are assigned. Assume that φ-$lsap$ is not a solution to (4.7), and another function φ-*opt* gives smaller assignment cost. It implies $C_{sum}(\varphi\text{-}opt)$ is less than $C_{sum}(\varphi\text{-}lsap)$. We use the same mechanism to create dummy tasks and extend the cost matrix. We extend function φ-*opt* to include those dummy tasks and arbitrarily map them to the

remaining $IS - T$ idle slots. So the total assignment cost for the expanded square matrix is $C_{sum}(\varphi\text{-}opt) + (IS - T)$. Because $C_{sum}(\varphi\text{-}opt)$ is less than $C_{sum}(\varphi\text{-}lsap)$, we can deduce that $C_{sum}(\varphi\text{-}opt) + (IS - T)$ is less than $C_{sum}(\varphi\text{-}lsap) + (IS - T)$. That means the solution given by LSAP algorithm is not optimal. This contradicts with the fact that LSAP algorithms give optimal solutions.

Constant 1 has been used as the assignment cost for dummy tasks. It turns out that we can choose any constant without violating optimality. The reason is the total assignment cost of all dummy tasks is a constant as well so that all task assignments perform equally well for dummy tasks. So what matters is the assignment of the T real tasks. It can be proved formally with the same method as above.

	s_1	$\dots$	$s_{IS\text{-}1}$	s_{IS}
t_1	1	1	0	0
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
t_T	0	1	1	0
t_{T+1}	1	1	1	1
$\dots$	1	1	1	1
t_{IS}	1	1	1	1

(a) $T < IS$

	s_1	$\dots$	s_{IS}	$s_{IS\text{+}1}$	$\dots$	s_T
t_1	1	1	0	1	1	1
$\dots$	$\dots$	$\dots$	$\dots$	1	1	1
t_i	0	1	1	1	1	1
$\dots$	0	0	0	1	1	1
$t_{T\text{-}1}$	0	0	0	1	1	1
t_T	0	0	0	1	1	1

(b) $T > IS$

Table 4.2: Expand cost matrix to make it square

For (a), last $|IS| - |T|$ rows are for dummy tasks we make up and all filled with 0.
For (b), last $|T| - |IS|$ columns are for dummy slots we make up and filled with 0.

For the case where T is larger than IS, we can use a similar technique to add extra T-IS columns representing dummy slots and fill them with 1, and therefore convert the original cost matrix to a square matrix. Fig. 4.2b shows an example. Then we can apply LSAP algorithms. After that, because dummy slots do not exist in reality, we remove those tasks that are assigned to dummy slots from the task assignment given by LSAP algorithms and get the final task assignment. We can prove its optimality ditto. Again, any constant can be used to fill the columns of dummy slots.

We integrate LSAP into our proposed optimal scheduling algorithm *lsap-sched* shown in Fig. 4.5. It naturally follows our prior discussion. Function $co\text{-}locate(T, S)$ checks whether slot S and the input data of task T are co-located on the same node. Function $expandToSquare(C, value)$ expands matrix C to the

closest square matrix by adding extra rows or columns filled with $value$. Function $lsap(C)$ uses an existing LSAP algorithm to calculate the optimal assignment for cost matrix C. Function $filterDummy(R)$ removes assignments for dummy tasks or dummy slots and returns the valid optimal task assignment.

```
Input: instant system state
Output: assignment of tasks to idle map slots
Algorithm:
TS ← the set of unscheduled tasks
ISS ← the set of idle map slots
C ← empty |TS| x |ISS| matrix
for i = 0; i < |TS|; ++i
  for j = 0; j < |ISS|; ++j
    if co-locate(TS[i], ISS[j])
      C[i][j] = 0
    else
      C[i][j] = 1
if C is not square: expandToSquare(C, 1)
R = lsap(C)
R = filterDummy(R)
return R
```

Figure 4.5: Algorithm skeleton of lsap-sched

Now we analyze when *lsap-sched* can be applied. Generally, the more idle slots and tasks there are, the more *lsap-sched* outperforms *dl-sched*. For the extreme case where there is only one idle slot, *dl-sched* and *lsap-sched* perform equally well. For lightly used Hadoop clusters, a large portion of slots are idle. At the start of a new job, the scheduler has multiple tasks to schedule and multiple idle slots at disposal so that *lsap-sched* performs much better. For heavily used clusters, only a small number of slots are idle and the superiority of *lsap-sched* is not fully demonstrated if new tasks are scheduled immediately. Instead, scheduling can be delayed by a short period to accumulate a sufficient number of idle slots before *lsap-sched* is applied. Tradeoffs between data locality and scheduling latency need to be made. The benefit of delayed scheduling, which results in the increase of the ratio of idle slots, can be quantified using (4.4). A simple strategy is to wait until the goodness of data locality exceeds a pre-set threshold. The real cluster traces show

the maximum utilization is seldom reached. CPU utilization was only 10% in Yahoos M45 cluster [73] and below 50% mostly in a Google cluster [25]. So on average a significant portion of slots is available when new tasks are submitted and *lsap-sched* is expected to perform substantially better than *dl-sched*. As our experiments below illustrate, the ratio of idle slots does not need to be high for *lsap-sched* to yield significant performance improvement.

4.4.3 lsap-sched for Heterogeneous Network

In above discussion, the assignment costs of non data local tasks are set to 1 uniformly. It assumes that the data movement of different tasks incurs an identical cost, which is not reasonable for typical hierarchical network topology where switches are increasingly oversubscribed when walking up the hierarchy. The cost of data fetching depends upon where the source node and destination node are located, and should not be assimilated. In our work, the assignment costs of non-data local tasks are computed based on network information. N(IS_j) is the node where slot IS_j resides. The input data of a non data local task may be stored on multiple nodes redundantly. If a task is assigned to IS_j for execution, the storage node with the best connectivity to N(IS_j) is chosen as data source. The calculation of C(i,j) is summarized in (4.8) where DS(T_i) is the size of the input data of task T_i, R_i is the replication factor of the input data of task T_i, ND(T_i,c) is the node where c-th replica of task T$_i$ is stored, and BW(N_1,N_2) is the available bandwidth between nodes N_1 and N_2. We can reuse (4.6) to calculate the sum of assignment costs. With a constructed cost matrix C(i,j), we want to find the task assignment that yields the smallest sum of costs. Mathematically, the goal function is the same as (4.7).

$$C(i,j) = \begin{cases} 0 & \text{if the input of task } i \text{ and slot } j \text{ are co-located} \\ \dfrac{DS(T_i)}{\max\limits_{1 \le c \le R_i} \{BW(ND(T_i,c), N(IS_j))\}} & \text{otherwise} \end{cases} \tag{4.8}$$

From (4.8), we can see the accuracy of pairwise bandwidth information impacts the calculation of assignment costs. Ideally real-time network throughput information should be used. Network Weather Service [112] can be utilized to monitor and predict network usage without injecting an overwhelming number of probing packets. One thing worth mentioning is per-node bandwidth may not equal per-task bandwidth. If

a single node has multiple available idle slots, the available network bandwidth per task depends on how many non data local tasks are assigned to the node and therefore is dynamic. We can set available bandwidth conservatively to total bandwidth divided by the number of idle slots (no matter whether all of them will be occupied). Alternatively, per-task (instead of per-node) network bandwidth can be collected and used to calculate assignment costs.

Table 4.3 shows examples of cost matrix. When the numbers of tasks and idle slots are not equal, the same transformation in section 4.4.2 can be reused but 0 is used to fill dummy cells instead of 1.

	s_1	...	s_{IS}-1	s_{IS}
t_1	1.3	1.9	0	0
...	...	...	...	...
t_T	0	2.8	1.5	0
t_{T+1}	0	0	0	0
...	0	0	0	0
t_{IS}	0	0	0	0

(a) $T < IS$

	s_1	...	s_{IS}	s_{IS}+1	...	s_T
t_1	2.7	...	0	0	0	0
...	...	...	...	0	0	0
t_i	0	...	1.9	0	0	0
...	...	...	...	0	0	0
t_T-1	1.3	...	0	0	0	0
t_T	0	...	0	0	0	0

(b) $T > IS$

Table 4.3: Expand cost matrix to make it square

For (a), last $|IS| - |T|$ rows are for dummy tasks we make up and all filled with 0.
For (b), last $|T| - |IS|$ columns are for dummy slots we make up and filled with 0.

Based on above discussion, we propose a network heterogeneity aware version of *lsap-sched* whose algorithm skeleton is shown in Fig. 4.6. The critical difference is network bandwidth information is used to compute assignment costs while constant values were used above.

4.5 Experiments

We have conducted simulation experiments to evaluate the effectiveness of our proposed algorithms. As we do not have access to real MapReduce production clusters, for some of the experiments below we need to mimic multi-user mode by setting some factors artificially (such as slot utilization, workload, and bandwidth usage).

```
Input: instant system state
Output: assignment of tasks to idle map slots
Algorithm:
TS ← the set of unscheduled tasks
ISS ← the set of idle map slots
C ← empty |TS| x |ISS| matrix
for i = 0; i < |TS|; ++i
  for j = 0; j < |ISS|; ++j
    set C[i][j] according to (4.8)
if C is not square: expandToSquare(C, 0)  # expand to a square matrix
R = lsap(C)                               # solve it using LSAP
R = filterDummy(R)                        # filter out dummy tasks
return R
```

Figure 4.6: Algorithm skeleton of heterogeneity aware lsap-sched

4.5.1 Impact of Data Locality in Single-Cluster Environments

In this test, we evaluate how important data locality is in single-cluster Hadoop systems. In other words, we want to know to what extent performance will degrade due to the deterioration in data locality. We wrote a random scheduler *rand-sched* which by default randomly assigns tasks to idle slots so that data locality greatly worsens. In addition, users can specify a parameter called randomness which tells *rand-sched* how random the scheduling should be. If its value is 100%, the scheduling will be thoroughly random; if its value is 0%, *rand-sched* degenerates to *dl-sched*. Other values yield a mixture of random and default scheduling. We compared the cases where *dl-sched* and *rand-sched* with randomness 0.3, 0.5, 0.7 and 1 were applied. An IO intensive application *input-sink*, which mainly reads data from HDFS, was written and used in the tests. We calculated the slowdown of *rand-sched* relative to *dl-sched* and show it in Fig. 4.7. The horizontal line y=0 is the baseline which implies the performance is as good as *dl-sched*. We observe that *rand-sched* with randomness 1 gives the worst performance and the slowdown is positively related to randomness.

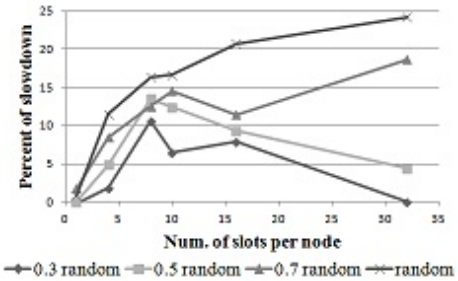


Figure 4.7: Single-cluster performance

4.5.2 Impact of Data Locality in Cross-Cluster Environments

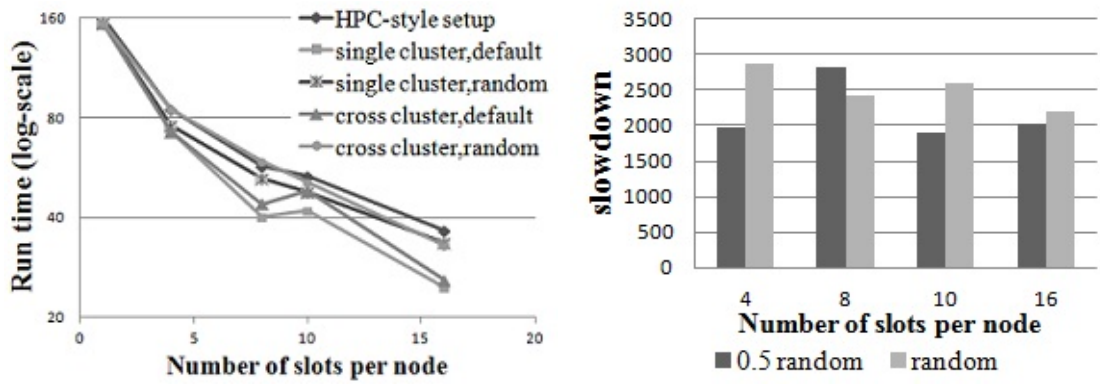
4.5.2.1 With high-speed interconnection among clusters

In this test, we evaluate how Hadoop performs in cross-cluster environments. We categorize deployments into three classes: single-cluster, cross-cluster and HPC-style. For cross-cluster deployments, HDFS and MapReduce share the same set of nodes that are scattered across multiple physical clusters. For HPC-style deployments, HDFS uses nodes in one cluster and MapReduce uses nodes in another cluster so that storage and compute are totally separated. We used clusters in FutureGrid that are equipped with high-speed inter-cluster network. Each node has 8 cores and gigabit Ethernet. Single-cluster deployments used 10 nodes in cluster India; cross-cluster deployments used 5 nodes in India and 5 nodes in Hotel; HPC-style deployments used 10 nodes in India for MapReduce and 10 nodes in Hotel for HDFS. Again, *input-sink* was used as test application, and results are shown in Fig. 4.8a. The plot matches our intuitive expectation. HPC-style deployment thoroughly loses data locality and performs the worst. *dl-sched* in single-cluster deployments performs the best. However, *dl-sched* in cross-cluster deployments performs better than rand-sched in single-cluster deployments. The reason is those clusters only see light use and the interconnection between clusters is fast enough to match the speed of local network to fulfill read/write requests.

4.5.2.2 With drastically heterogeneous network

We set up a unified Hadoop cluster across multiple physical clusters by building a virtual network overlay with ViNe [110]. To know to what extent performance is impacted by the throughput of inter-cluster links, ViNe provided low throughput: only 1-10Mbps. We compared rand-sched with *dl-sched* and show results in

Fig. 4.8b. The loss of data locality slows down the execution by thousands of times. The results also apply to the case where the inter-cluster network is fast but heavily oversubscribed so that on average each application can only get a fairly small share. This demonstrates that Hadoop is not optimized for fairly heterogeneous networks (e.g. Wide-Area Network) so that Hadoop deployments over geographically distributed clusters with oversubscribed interconnection should be carefully investigated.



(a) with high-speed cross-cluster net

(b) with drastically heterogeneous net

Figure 4.8: Cross-cluster performance

4.5.3 Impact of Various Factors on Data Locality

In this set of tests, we evaluate how different factors impact the goodness of data locality for *dl-sched*. As we mentioned in section 4.3.2, simulation is conducted here instead of direct numeric calculation. This makes us able to circumvent the imposed constraints (i.e. C and S are 1) in our prior mathematical deduction. The investigated factors include the number of tasks, the number of map slots per node, replication factor, the number of nodes and the ratio of idle map slots. The configuration is shown in table 4.4. For each test, we varied one factor while fixing all others. All results are shown in Fig. 4.9.

Fig. 4.9a shows how the goodness of data locality changes with the number of tasks. We observe that the goodness of data locality decreases as the number of tasks is increased initially. When the number of tasks becomes 2^7 (128), data locality is the worst. As the number of tasks is increased further beyond 2^7, the goodness of data locality increases sharply. The degree of increment is decreased as there are more and more tasks.

Parameter	Default value	Value range when tested	Env. in Delay Sched. Paper
num. of nodes	1000	[300, 5000]; step 100	1500
slots per node	2	[1, 32]; step 1	2
num. of tasks	300	$(2^0, 2^1, \ldots, 2^{13})$	$(2^4, \ldots, 2^{13})$
ratio of idle slots	0.1	[0.01, 1]; step 0.02	0.01
replication factor	3	[1, 20]; step 1	3

Table 4.4: System Configuration

Increasing the number of slots per node yields more idle slots (the ratio of idle slots is fixed). Its impact on data locality is shown in Fig. 4.9b. The data locality improves drastically as the number of slots per node increases initially. 5, 8 and 10 slots per node yield the goodness of data locality 50%, 80% and 90% respectively. Considering the reality that modern server nodes have multiple cores and multiple processors, to run 5-10 tasks concurrently on each node is reasonable. So to specify more slots per node improves not only the resource utilization but also the data locality. Prior research investigated the impact of concurrently running tasks on resource usage, but has not explored its impact on data locality. Our result quantifies the relationship and serves as guidance for users to tune the system.

The impact of replication factor is shown in Fig. 4.9c. As we expect, replication factor has positive impact: the increase of replication factor yields better data locality. However, the relationship is not linear. The degree of improvement decreases with increasing replication factor. More storage space is required as replication factor is increased and that relationship is linear. Fig. 4.9c can help system administrators choose the best replication factor that balances storage usage and data locality, because it tells how much data locality is lost/gained when replication factor is decreased/increased. As replication factor gets larger and larger, the benefit becomes more and more marginal. Based on how scarce storage space is, the sweet spot of replication factor can be carefully chosen according to Fig. 4.9c to achieve the best possible data locality, compared with the case where replication factor is arbitrarily chosen.

Fig. 4.9d shows the impact of varying the ratio of idle slots. When the ratio of idle slots is around 40% and therefore the utilization ratio is 60%, the goodness of data locality is over 90%. This means the utilization ratio of all slots need not be very low to get reasonably good data locality, which is a little counter-intuitive.

Even if most of the slots are busy, the goodness of data locality can still reach around 30% given that many tasks are to be scheduled, because the scheduler can choose the tasks that can achieve the best data locality among all unscheduled tasks at hand.

A general intuition is that as more nodes are added to a system, the performance usually should be improved. However, the degree of performance improvement is not necessarily linear with the number of nodes. In this test, we increased the number of nodes from 300 to 5000 and the results are shown in Fig. 4.9e. Surprisingly, the goodness of data locality drops as we add more nodes initially. When there are around 1500 nodes, the goodness of data locality becomes the lowest. Beyond 1500, the goodness of data locality is positively related to the number of nodes. To figure out why 1500 is the stationary point, we calculated the ratio of the number of idle slots to the number of tasks, which is used as the x-axis in Fig. 4.9f. Data locality is the worst when there is equal number of idle slots and tasks. We also redraw Fig. 4.9a using the same transformation and present the result in Fig. 4.9f. The two curves in Fig. 4.9f have the similar shapes. From these two plots, we can see that data locality deteriorates sharply when there are less idle slots than tasks and the ratio between them increases. Under these circumstances, tasks need to be scheduled in multiple waves for all of them to run. For each wave, the scheduler can cherry-pick from remaining tasks those that can achieve the best data locality. As the number of idle slots gets close to the number of tasks, the freedom of cherry-picking is decreased because in each wave more tasks need to be scheduled. The freedom is totally lost when the numbers of tasks and idle slots are equal because all tasks need to be scheduled in one wave. When there are more idle slots than tasks, the scheduler can cherry-pick the slots that will yield the best data locality. To summarize, when there are less/more idle slots than tasks, the scheduler can cherry-pick tasks/idle slots among all possible assignments to obtain the best locality. The degree of cherry-picking freedom increases as the difference between the numbers of idle slots and tasks gets larger. Another observation is that the curve is not symmetric with respect to the vertical line ratio=1. The loss of data locality when the ratio grows towards 1 is much faster than the regaining of data locality when the ratio grows beyond 1. When the number of tasks is 40 times that of idle slots, the goodness of data locality is above 90%; while it is only around 50% when the number of idle slots is 200 times that of tasks.

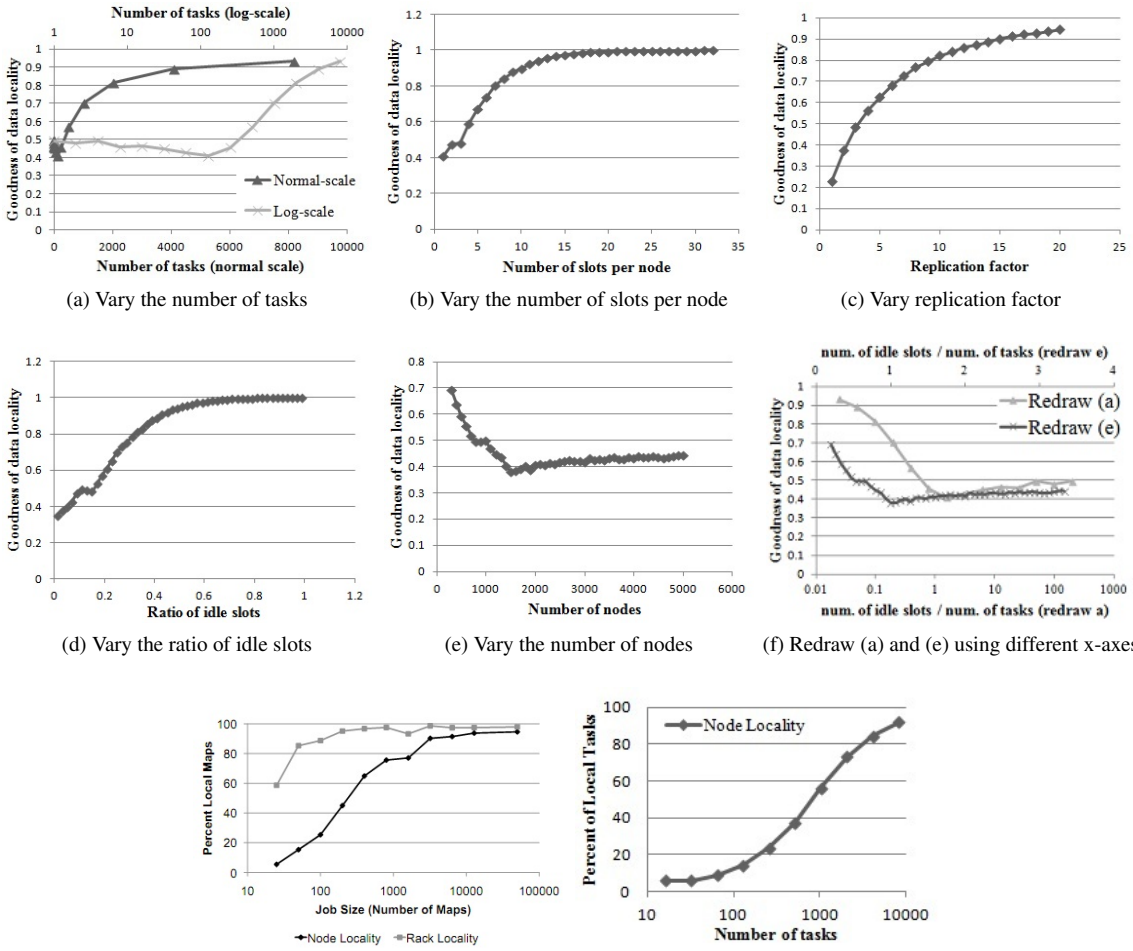
Tests in Fig. 4.9b, 4.9d and 4.9e all result in the change of idle slots with the number of tasks fixed, but they have different curves. The critical difference is that in Fig. 4.9e the number of nodes is changed while in Fig. 4.9b and 4.9d the number of nodes is constant. The difference of those curves originates from the

fact that tasks are scheduled to slots while input data are distributed to nodes. For Fig. 4.9b and 4.9d, the distribution of data is constant and task scheduling varies according to the change of the number of idle slots. Having more nodes means the input data of a set of tasks are more spread out, which has a negative impact on data locality. For Fig. 4.9e, both data distribution and idle slot distribution vary. Increasing the number of slots per node is not equivalent to adding more nodes in terms of data locality.

To verify how close our simulation is to the real system in terms of accuracy, we compared a real trace and our simulation result. In [116], the authors analyzed the trace data collected from Facebook production Hadoop clusters. They found that the system is more than 95% full 21% of the time and 27.1 slots are released per second. The total number of slots is 3100, so the ratio of idle slots is $27.1/3100 \approx 1\%$. We could not find the number of slots per node in the paper. So we assumed the Facebook cluster uses the default Hadoop setting: 2. Then we can deduce that there are $3100/2 \approx 1500$ nodes in the system. Replication factor is not explicitly mentioned in the paper for the trace and we assumed the default Hadoop setting 3 is used. The cluster configuration is summarized in the last column of table II. The authors measured the relationship between the percent of local map tasks and job size (number of map tasks). We duplicate their plot in Fig. 4.9f, in which both node locality and rack locality are shown. We ran simulation tests with the same configuration and show results in Fig. 4.9h. By comparing the two plots, we observe that our simulation gives reasonably good results. Firstly, the curves are similar and both have an S shape. Secondly the concrete y values are also close. So the assumptions we made are reasonable for real clusters. Accurate comparison is impossible unless we have the raw data of Fig. 4.9f. As we said, although the ratio of idle slots in real systems is not constant across time, we can divide the whole time span into shorter periods for each of which the ratio of idle slots is approximately constant and our simulation can be conducted.

4.5.4 Overhead of LSAP Solver

We measured the time taken by *lsap-sched* to compute optimal task assignment in order to understand the overhead of solving LSAP. We varied the number of tasks from 100 to 3000 with step size 400, and the number of idle slots was equal to the number of tasks for each test. They represent small-sized to moderate-sized clusters. The corresponding cost matrices were constructed and fed into LSAP solver. It took 7ms, 130ms, 450ms, and 1s for the LSAP solver to find optimal solutions given the matrices of sizes 100x100,



(a) Vary the number of tasks (b) Vary the number of slots per node (c) Vary replication factor

(d) Vary the ratio of idle slots (e) Vary the number of nodes (f) Redraw (a) and (e) using different x-axes

(g) Real trace copied from (h) Sim. result with similar config

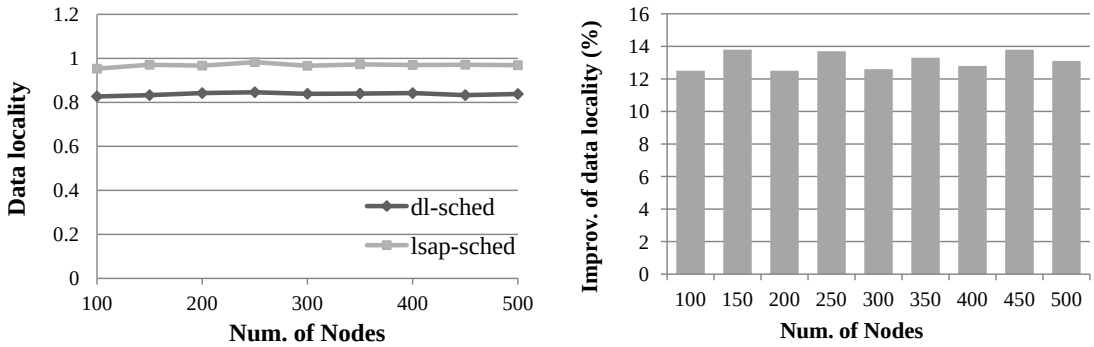
Figure 4.9: Impact of various factors on the goodness of data locality and Comparison of real trace and simulation result

500x500, 1700x1700 and 2900x2900. So the overhead is acceptable in small-sized to medium-sized clusters. Note in our tests, values in the matrices were randomly generated, which eliminates the possibility to mine and explore useful patterns of cost distribution. In reality, the cost of data movement exhibits locality for typical hierarchical network topologies, which can be used to speed up the execution potentially.

4.5.5 Improvement of Data Locality by *lsap-sched*

We have shown that *dl-sched* is not optimal. However, we are not clear yet about how non-optimal it is. In this experiment, we ran simulations to measure how close *dl-sched* is to the optimum. The reasons why we run simulations rather than use closed-form formulas for *dl-sched* have been explained in section 4.3.2.

We consider the case where the number of tasks to schedule is no greater than the number of idle slots. In this test, we evaluate how *lsap-sched* impacts the percent of data local tasks. In the simulated system, the number of nodes varied from 100 to 500 with step size 50, and each node had 4 slots. Replication factor was 3. The ratio of idle slots was fixed to 0.5 and enough tasks were generated to utilize all idle slots. The results are shown in Fig. 4.10. Fig. 4.10a shows the goodness of data locality for *dl-shed* and *lsap-sched*. Obviously, the goodness of data locality is pretty stable for both algorithms: dl-sched achieves 83% while *lsap-sched* achieves 97%. Their differences are shown in Fig. 4(b), which implies *lsap-sched* increases the goodness of data locality by 12% - 14%. This indicates that lsap- sched consistently outperforms *dl-sched* significantly when the system is scaled out. In addition, we observe that the improvement oscillates in Fig. 4.10b. Our conjecture of the cause is that the number of all possible data and slot distributions is gigantic and only a portion of them was covered in our tests.



(a) Data locality of dl-sched and lsap-sched

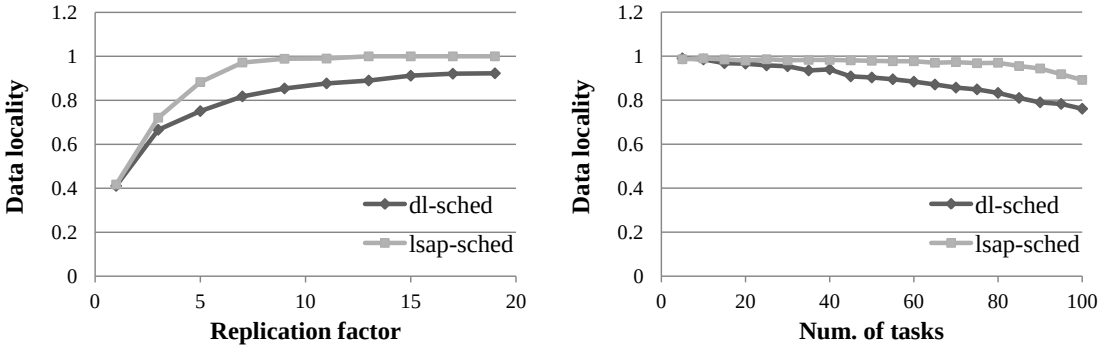
(b) Percent of improvement

Figure 4.10: Comparison of data locality (varied the num. of nodes)

Then we varied replication factor from 1 to 19 and fixed the number of nodes to 100. The goodness of data locality was measured and is shown in Fig. 4.11a. The increase of replication factor yields substantial data locality improvement for both *lsap-sched* and *dl-sched*; and *lsap-sched* can more efficiently explore the

increasing data redundancy and thus achieve better data locality. Common replication factors 3 and 5 yield surprisingly high data locality 72% and 88% respectively for *lsap-sched*.

Finally, we set the total number of idle slots to 100 and increased the number of tasks from 5 to 100 so that they used more and more idle slots. Results are shown in Fig. 4.11b. Data locality degrades slowly as more tasks are injected into the system, and the degradation of *lsap-sched* is much less severe than that of *dl-sched*. When the number of tasks is much smaller than that of idle slots, the scheduler has the great freedom of picking the best slots to assign tasks. As their numbers become close, more tasks need to be scheduled in one "wave" and the cherry-picking freedom is gradually attenuated. This explains why the increase of the number of tasks has negative impact on data locality.



(a) Varied replication factor

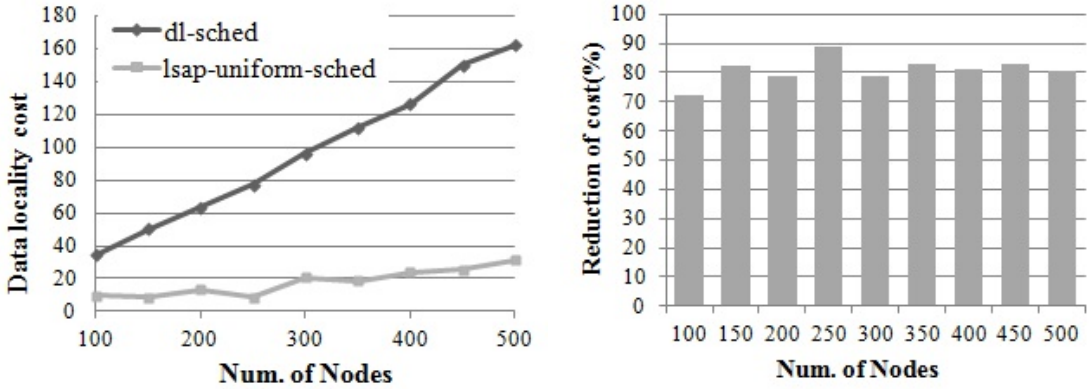
(b) Varied the number of tasks

Figure 4.11: Comparison of data locality (varied rep. factor and the num. of tasks)

4.5.6 Reduction of Data Locality Cost

In above tests, we measured the percentage of data local tasks. In reality, performance depends upon not only the goodness of data locality, but also the incurred data movement penalty of non data local tasks. In this test, we measure the overall data locality cost (DLC) to quantify the performance degradation brought by non data local tasks. The DLC of any non data local task was set to 1 regardless of the proximity between the location of compute and input data, so we call it *lsap-uniform-sched*. The same test environment as above is used. Fig. 4.12a shows the absolute DLC. As the number of nodes is increased, the DLC of *dl-sched* increases much faster than that of *lsap-uniform-sched*. So *lsap-uniform-sched* is more resilient to system scale-out than

dl-sched. We also computed the DLC reduction of *lsap-uniform-sched* against *dl-sched*, and show results in Fig. 4.12b. We observe that *lsap-uniform-sched* eliminates 70% - 90% of the DLC of *dl-sched*.

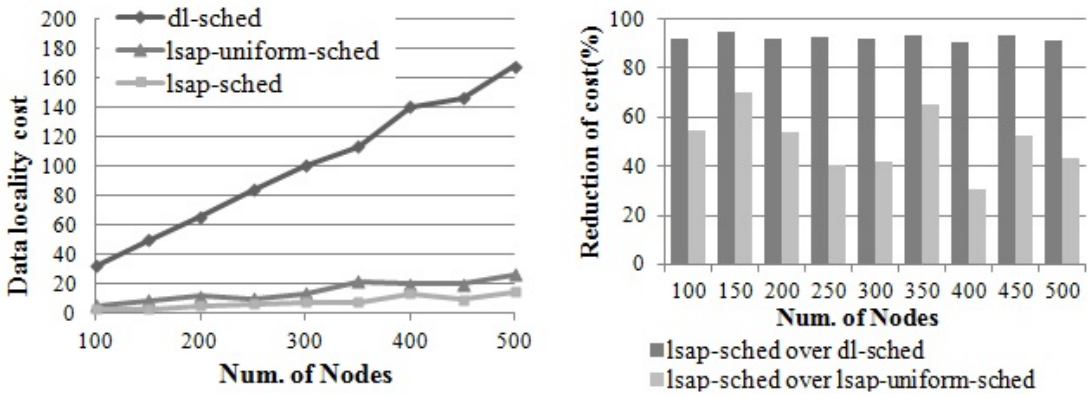


(a) Overall DLC (b) Reduction of DLC(in percent)

Figure 4.12: Comparison of data locality cost (with equal net. bw.)

Previously, constant value 1 was used as the DLC of non data local tasks, which does not reflect the fact that pairwise network bandwidths are not uniform (e.g. intra-rack throughput is usually higher than cross-rack throughput). In this test, we complicate the tests by setting non-constant costs. We simulated a cluster where each rack has 20 nodes. Again the DLC of data local tasks is 0. The DLC of rack local tasks (i.e. computation and input data are co-located on the same rack) follows a Gaussian distribution with mean and standard deviation being 1.0 and 0.5 respectively. The DLC of remote tasks follows another Gaussian distribution with mean and standard deviation being 4.0 and 2.0 respectively. This setting matches the reality that cross-rack data fetching incurs higher cost than intra-rack data fetching. For *lsap-uniform-sched*, the average of intra-rack and cross-rack DLC is used (which is 2.5). We varied the total number of nodes from 100 to 500 and compared *dl-sched*, *lsap-uniform-sched* and *lsap-sched*. Firstly the ratio of idle slots was set to 50% and DLC is shown in Fig. 4.13. *Lsap-sched* outperforms *dl-sched* significantly by up to 95%, and outperforms *lsap-uniform-sched* by up to 65%. Comparing Fig. 4.12 and Fig. 4.13, we observe that rack topology does not result in performance degradation. *Dl-sched* is rack aware in the sense that rack local tasks are preferred over remote tasks if there are no node local tasks. So it avoids assigning tasks to the nodes where they need to fetch input data from other racks with best efforts. *Lsap-sched* is naturally rack aware because the high cross-rack data movement cost prohibits non-optimal task assignments. So both *dl-sched*

and *lsap-sched* can effectively utilize the network topology information. Then we decreased the ratio of idle slots from 50% to 20%, which implies there was a fewer number of idle slots. Results are show in Fig. 4.14. Compared with Fig. 4.13, the DLC of *dl-sched* is decreased and the DLC of *lsap-sched* is increased, so that the performance superiority of *lsap-sched* over *dl-sched* becomes less significant which is between 60% and 70%. *Lsap-sched* outperforms *lsap-uniform-sched* by 40% - 50%. When there are only a small number of idle slots, the room of improvement brought by *lsap-sched* is minor. For the extreme case where there is only one idle slot, *dl-sched* and *lsap-sched* become equivalent approximately. The more available resources and tasks there are, the more *lsap-sched* reduces DLC. The utilization of typical production clusters rarely reaches up to 80% [22, 23]. So we believe *lsap-sched* can offer substantial benefit in moderately-loaded clusters.

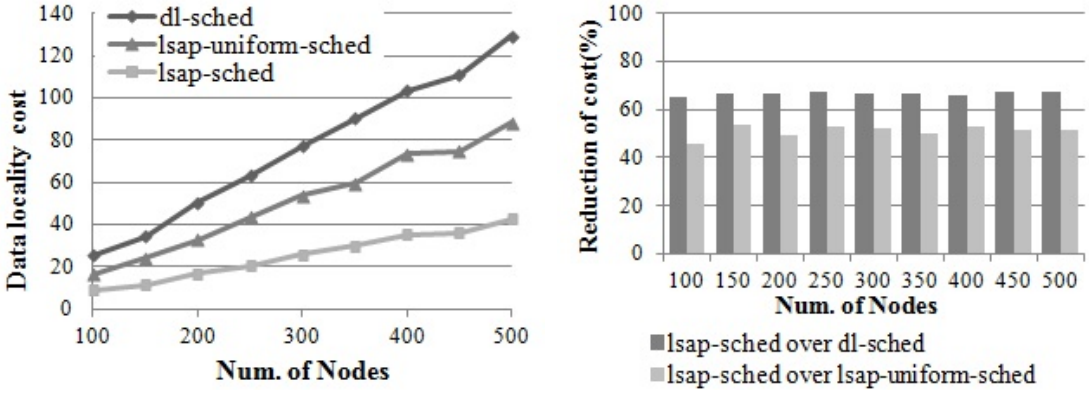


(a) Overall DLC

(b) Reduction of DLC(in percent)

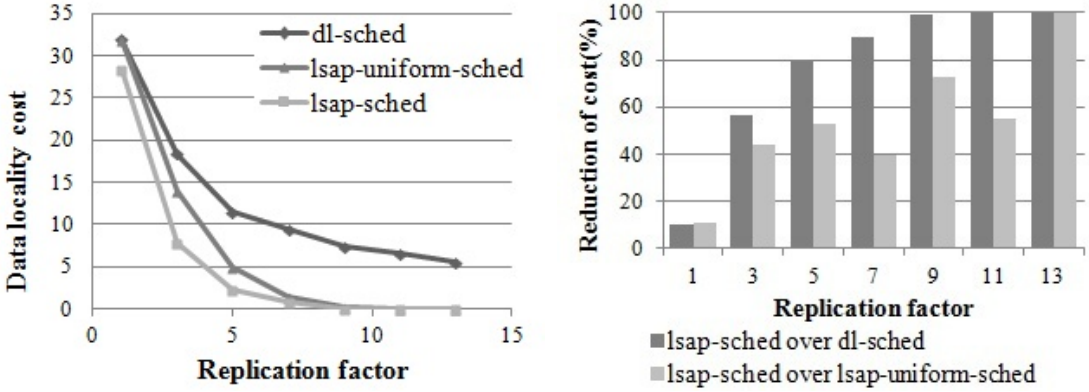
Figure 4.13: Comparison of DLC with 50% idle slots

Then we fixed the number of nodes to 100 and increased replication factor from 1 to 13 with step size 2. The other settings were identical to the test above except each node has 1 idle slot. As replication factor is increased, we expect positive impact on DLC because the possibility of achieving better data locality increases theoretically. Test results are shown in Fig. 4.15. Firstly, DLC drastically decreases as replication factor is increased initially from small values, and the decrement of DLC becomes less significant as replication factor gets larger and larger. Secondly, when replication factor is 1, all algorithms perform comparably. *Lsap-sched* and *lsap-uniform-sched* have much faster DLC decrease than *dl-sched* as replication factor grows, and it almost thoroughly eliminates DLC when replication factor is larger than 7. Fig. 4.15b shows a low replication factor (e.g. 3) is sufficient for *lsap-sched* to outperform *dl-sched* by over 50%. Note the number of slots per node was set to 1 in this test, and increasing it can bring larger improvement.



(a) Overall DLC (b) Reduction of DLC(in percent)

Figure 4.14: Comparison of DLC with 20% idle slots



(a) Overall DLC (b) Reduction of DLC(in percent)

Figure 4.15: Comparison of DLC w/ rep. factor varied

In summary, our algorithms can improve data locality and reduce DLC significantly. Network bandwidth-based cost specification yields much better performance than constant costs.

4.6 Integration of Fairness

4.6.1 Fairness and Data Locality Aware Scheduler: *lsap-fair-sched*

We next investigate the integration of fairness into *lsap-sched*. Both capacity scheduler [3] and fair scheduler [117] take the same approach that jobs are organized into different groups by appropriate criteria (e.g. user-based, domain-based, pool-based). This approach is adopted by us as well.

We do not enforce strict fairness which constrains each group cannot use more than its ration strictly, because it results in the waste of resources. We loosen the constraint. If there are excess idle slots to run all tasks, we just schedule them immediately to make full use of all resources even if some of the groups have used up their rations. If idle slots are insufficient, we need to selectively run tasks aiming to comply with ration specifications.

We enhance *lsap-sched* to support fairness by carefully tuning the cost matrix C. The assignment cost of a task can be positively related to the resource usage of the group the task belongs to. In other words, for groups that use up or overuse the allocated capacity, their tasks have high assignment costs so that the scheduler does not favor them. Oppositely, the assignment costs of tasks from groups which underuse their allocated capacity are low so that they get higher priority.

Let G represent the set of groups that a system administrator configures for a cluster, and i-th group is G_i. Each group contains some number of tasks and each task can only belong to exactly one group. Given a task T, function *group*(T) returns the group which T belongs to. Each group is assigned a weight/ration w which is the portion of map slots allocated to it. The sum of the weights of all groups is 1.0, which is formulated in (4.9). At time t, $rt_i(t)$ is the number of running tasks belonging to group G_i. Formula (4.10) calculates the ratio of map slots used by group G_i among all occupied map slots, which measures the real resource usage ratio of each group. For group G_i,the desired case is that s_i and w_i are equal, which implies real resource usage exactly matches the configured share. If s_i is less than w_i, group G_i can have more tasks scheduled immediately. If group G_i has used its entire ration, to schedule more tasks, it needs to wait until some of its tasks complete or there are sufficient idle slots to run all tasks. A *Group Fairness Cost* GFC is associated with each group to measure its "priority" of scheduling and calculated via (4.11). Groups with low GFC have high priority so that their tasks are considered before the tasks from groups with high GFC.

Fairness-favored		Data Locality-favored		Both-favored	
FC*	DLC*	FC	DLC	FC	DLC
[0, 100]	[0, 20]	[0,100]	[0,200]	[0,100]	[0,100]
(a)		(b)		(c)	

Table 4.5: Examples of How Tradeoffs are Made

* FC: fairness cost; DLC: data locality cost

Data locality sometimes conflicts with fairness. For example, it is possible that the unscheduled tasks that can achieve data locality are mostly from groups that have already used up their rations. And thus we get into the dilemma that tradeoffs between fairness and data locality must be made. To integrate data locality and fairness, we divide assignment cost into two parts: Fairness Cost (FC) and Data Locality Cost (DLC) (shown in (4.12)). From the aspect of fairness constraints, FC implies the order of tasks to be scheduled, and tasks with low FC should be scheduled before tasks with high FC. The range of FC is denoted by [0, FC_{max}]. DLC reflects the overhead of data movement and has the same meaning as the cost definition described above in section 3.2. The weights of FC and DLC can be implicitly adjusted by carefully choosing the value ranges. Table 1 gives examples of how fairness-favored scheduling, data locality-favored scheduling and both-favored scheduling can be achieved. The range of FC is [0, 100] for all the examples, while that of DLC varies. DLC with range [0, 20] makes the scheduler favor fairness because FC has a larger impact on the total assignment cost. DLC with range [0, 200] makes the scheduler favor data locality because the loss of data locality bumps up the total assignment cost significantly. DLC with range [0, 100] makes the scheduler favor both fairness and data locality, because the loss of data locality and fairness impacts overall assignment costs to the same extent.

Above example shows how data locality and fairness can be balanced. We need to quantitatively determine the *FC* and *DLC* of tasks dynamically. Formula (4.13) shows how to calculate *DLC*, in which α is a configuration parameter fed by system administrators and implicitly controls the relative weight of *DLC*. If α is small, *FC* is dominant and the scheduler favors fairness. If α is large, *DLC* stands out and the scheduler favors data locality. If α is medium, *FC* and *DLC* become equally important. The calculation of *FC* is trickier and more subtle. As we mentioned, a GFC is associated with each group. One simple and intuitive strategy is for each group the *FC* of all its unscheduled tasks is set to its GFC. This implies all unscheduled tasks of

a group have identical FC, and thus the scheduler is inclined to schedule all or none of them. Consider the scenario where FC dominates. Initially a group G_i underuses its ration just a little and has many unscheduled tasks. If group G_i has the lowest GFC, all its tasks naturally have the lowest FC and are scheduled to run before other tasks so that group G_i uses significantly more resources. After scheduling, the resource usage of groupG_i changes from slight underuse to heavy overuse. The reason why resource usage oscillates between underuse and overuse is the tasks in each group, no matter how many there are, are assigned the same FC. Instead, for each group we calculate how many of its unscheduled tasks should be scheduled based on the number of idle slots and its current resource consumption, and set only their FC to GFC. In (4.14) *AS* is the total number of all slots and *AS*·w_i is the number of map slots that should be used by group G_i. Group G_i already has rt_i tasks running so we have *AS*·w_i-rt_i slots at disposal (termed *sto – Slots To Occupy*). Because G_i can use only sto_i more slots, accordingly the FC of at most sto_i tasks is set to GFC$_i$ and that of other tasks is set to a larger value (1-w_i)·β (β is the scaling factor of FC). So the tasks of each group do not always have the same FC. The number of unscheduled tasks for group G_i is denoted by ut_i. If ut_i is greater than sto_i, we need to decide how to select sto_i tasks out of ut_i tasks. Then data locality comes into play, and the tasks that can potentially achieve data locality are chosen. Details are given in the proposed algorithm below. Note parameters α and β, which are used to balance data locality and fairness, are specified by system owners based on their performance requirement and desired degrees of fairness.

$$\sum_{i=1}^{|G|} w_i = 1 \ (0 < w_i \leq 1) \tag{4.9}$$

$$s_i(t) = \frac{rt_i(t)}{\sum_{j=1}^{|G|} rt_j(t)} \quad (1 \leq i \leq |G|) \tag{4.10}$$

$$GFC_i = \frac{s_i}{w_i} \cdot 100 \tag{4.11}$$

$$C_{ij} = FC(i,j) + DLC(i,j) \tag{4.12}$$

$$DLC(i,j) == \begin{cases} 0 & \text{if data locality is achieved} \\ \alpha \cdot \dfrac{DS(T_i)}{\max\limits_{1 \leq c \leq R_i} \{BW(ND(T_i,c), N(S_j))\}} & \text{otherwise} \end{cases} \tag{4.13}$$

$$sto_i = \max(0, AS \cdot w_i - rt_i) \tag{4.14}$$

Based on above discussion, scheduling algorithm *lsap-fair-sched* is proposed and shown below. The main difference than *lsap-sched* is how assignment costs are calculated. Lines 7-8 find the set of nodes with idle slots. Lines 10-12 find the set of tasks whose input data are stored on nodes with idle slots. So these tasks have the potential to achieve data locality while all other tasks will lose data locality definitely for current scheduling. Lines 13-16 calculate *sto* of all groups. Lines 18-27 calculate task FC. Lines 29-33 calculate DLC. Line 35 adds together matrices FC and DLC to form the final cost matrix, which is expanded to a square matrix shown in line 36. After that, a LSAP algorithm is used to find the optimal assignment which is subsequently filtered and returned.

Algorithm skeleton of *lsap-fair-sched*

Input:

α : DLC scaling factor for non data local tasks

β : FC scaling factor for tasks that are beyond its group allocation

Output:

assignment of tasks to idle map slots

Functions:

rt(g): return a set of running tasks that belong to group g.

node(s): return the node where slot s resides

reside(T): returns a set of nodes that host the input data of task T

Algorithm:

```
1 TS ← the set of unscheduled tasks
2 ISS ← the set of idle map slots
3 w ← rations/weights of all groups
4 ut ← the number of unshed.  tasks for all groups
5 gfc ← GFC of all groups calculated via (4.11)
6 INS ←  ∅ # the set of nodes with idle slots
7 for slot in ISS:
8 INS ← INS ∪ node(slot)
9 DLT[1:|G|] = ∅ # tasks that can potentially gain data locality
10 for T in TS:
11     if reside(T) ∩ INS ≠  ∅:
12         DLT[group(T)] = DLT[group(T)] ∪ T
13 for i in 1:|G|
14     diff = w[i] · AS - rt[i]
15     if diff > 0:  sto[i] = min(diff, ut[i])
16     else:  sto[i] = 0
17
```

```
fc[1:|TS|][1:|ISS|] = 0 #fill with deft value
for i in 1:|G|
    tasks = G[i] #a list of tasks in group i
    NDLT = tasks - DLT[i] #non-local tasks
    fc[tasks] = β·(1-w[i]) #default value
    if |DLT[i]|  ≥ sto[i]:
       tasks = DLT[i][1:sto[i]] #choose a subset
    else if ut[i] > sto[i]:
       tasks = DLT[i] ∪ NDLT[1:(sto[i]-|DLT[i]|]]
    fc[tasks] = gfc[i] #assign GFC to some tasks

dlc[1:|TS|][1:|ISS|]=1
for T in ∪DLT[i]:
   for j in 1:|ISS|
      if co-locate(T, ISS[j]):  dlc[T][j] = 0
      else:  dlc[T][j]= α·DS(T)/BW(T,ISS[j])

C = fc + dlc
if C is not square:  expandToSquare(C, 0)
R = lsap(C)
R = filterDummy(R)
return R
```

In above strategy, some tasks may get starved although the possibility is remote (e.g. tasks with bad data locality may be queued indefinitely). FC and DLC can be reduced for the tasks that have been waiting in queue for long time, so that they tend to be scheduled at the subsequent scheduling points.

The Dryad scheduler Quincy also integrates fairness and data locality [67]. In that paper, fair and unfair sharing with and without preemption are discussed and evaluated. Our approach is much simpler, and we evaluate continuous tradeoffs below between data locality and fairness instead of two extreme cases - fair and unfair. In addition, we believe Quincy suffers from the performance oscillation issue discussed above (i.e. a job, which has many unscheduled tasks and underuses its allocation slightly, is likely to overuse its allocation significantly in subsequent scheduling when many slots become available).

4.6.2 Evaluation of lsap-fair-shed

We have shown that theoretically fairness and data locality can be integrated together by carefully setting Fairness Cost and Data Locality Cost. In this experiment, we conducted a series of simulations to evaluate our proposed algorithm *lsap-fair-sched*. For each group, formula (4.15) calculates the *fairness distance* between the actual resource allocation s_i and the desired allocation specified via weights w_i. Value 0 indicates that the group uses exactly its ration. If its value is greater than 0, the resource usage of the group either exceeds or is less than its ration. So its value indicates the compliance with administrator-provided allocation policies, and smaller is better usually. Formula (4.16) calculates the mean of fairness distance of all groups and serves as a metric to measure the fairness of resource allocation. At initial time instant 0, the fairness distance is denoted by $d(0)$. Then submitted tasks are scheduled, and at time $t$the fairness distance becomes $d(t)$. If the scheduler is fairness-aware, usually $d(t)$ should be smaller than $d(0)$ which implies fairness is improved. Given an initial state, we use $d(0) - d(t)$ to measure to what extent *lsap-fair-sched* improves fairness, and larger is better.

$$d_i(t) = |s_i(t) - w_i|/w_i \tag{4.15}$$

$$d(t) = \frac{\sum_{i=1}^{|G|} d_i(t)}{|G|} \tag{4.16}$$

In our tests, there were 60 nodes; each node had 1 slot; half of all slots were idle; replication factor was 1; and there were 30 running tasks and 90 tasks to schedule. In addition, there were 5 groups to which tasks belong. Weights for groups were $\{2^0, 2^1, 2^2, 2^3, 2^4\}$ and normalized to $\{2^0/31, 2^1/31, 2^2/31, 2^3/31, 2^4/31\}$ so that they add up to 1.0. The groups to which running tasks belong were randomly assigned. The DLC of non-local tasks was varied and results are shown in Fig. 4.16. Initially, DLC is small compared with FC so that FC dominates the total assignment cost and *lsap-fair-sched* improves fairness most. Gradually, as the DLC of non-local tasks increases, data locality gains larger weight so that data locality improves and fairness deteriorates. After the DLC of non-local tasks gets sufficiently large, data locality becomes the dominant factor so that scheduling favors data locality mainly. Another observation is that improvement/ deterioration of data locality/fairness is not smooth, and the curves are staircase shaped. During the continuous increase of DLC, not every small increment makes DLC become dominant. There are some critical steps that cause "phase transition" and make the assignment costs of some tasks become larger than that of other tasks that

had larger cost before, so that data locality becomes dominant in scheduling. Oppositely, the non-critical increase is not sufficient to influence scheduling decisions.

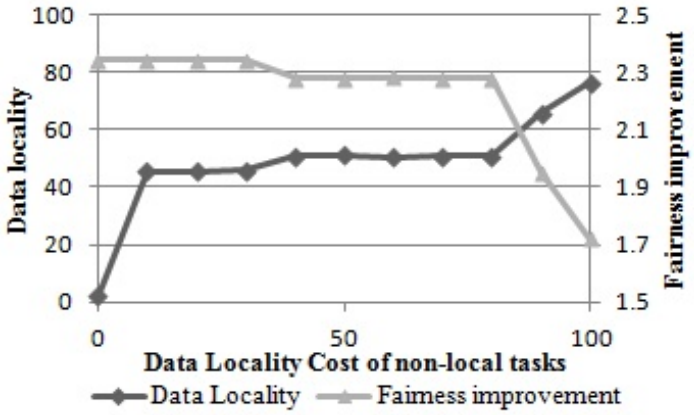


Figure 4.16: Tradeoffs between fairness and data locality

4.7 Summary

The overall goal of the work discussed in this chapter is to investigate data locality in depth for data parallel systems, among which GFS/MapReduce is representative and thus our main research target. We have mathematically modeled the system and deduced the relationship between system factors and data locality. Simulations were conducted to quantify the relationship and some insightful conclusions have been drawn which can help to tune Hadoop effectively. We will complicate the model to integrate more factors that are critical to real clusters (e.g. non-assimilation of system state). In addition, non-optimality of default Hadoop scheduling has been discussed and an optimal scheduling algorithm based on LSAP has been proposed to give the best data locality. We ran intensive experiments to measure how our proposed algorithm outperforms default scheduling and demonstrate its performance superiority. Above research uses data locality as a performance metric and the target of optimization. Besides that, we investigated how data locality impacts the user-perceived metric of system performance: job execution time. Three scenarios single-cluster, cross-cluster and HPC-style setup, have been discussed and real Hadoop experiments were conducted. It shows data locality is important to single-cluster deployments. Also it shows the inability of Hadoop to tackle significant network heterogeneity and the importance of inter-cluster connection to performance.

5

Automatic Adjustment of Task Granularity

In parallel computing, one critical issue is to decide the optimal task granularity for applications. There is no single one-size-fit-all solution for it. The optimal setting depends upon many factors, such as hardware, application workload and scheduling algorithms. Small granularity results in that more tasks are generated for the same amount of work and the run time of each task is shorter compared with large granularity, which inevitably increases the ratio of task management overhead to execution time. Meanwhile, more independent small tasks for an application brings better parallelism compared with few large tasks. Task granularity influences load balancing too. Small granularity gives flexibility for load balancing. These tradeoffs are summarized in Table 5.1.

	Management Overhead	**Concurrency**	**Load Balancing**
Small granularity	High	High	Easy
Large granularity	Low	Low	Hard

Table 5.1: Trade-off of task granularity

5.1 Analysis of Task Granularity in MapReduce

Tasks are schedulable entities and map operations must be organized as tasks for execution. In each map task, the same map operation is applied to all key-value pairs within input data. Each key-value pair is called a *record*. The number of records processed by a map task literally defines task granularity. The MapReduce

model itself does not impose any constraint on how map operations are grouped into tasks. Theoretically, the map operations of a job can be grouped arbitrarily without affecting correctness. However, it affects the efficiency of execution. The key issue is how to partition all input data into blocks, each of which is processed by one task to achieve good load balancing, high concurrency and low management overhead. To maximize performance, load unbalancing should be avoided and the tradeoff between concurrency and overhead must be considered.

In Hadoop, each map task processes one data block stored in HDFS. By default, the block size is 64MB and system administrators can tune this parameter to based on their requirements. For data of the same size, small block size results in more blocks, which means more metadata operations are required to access the data. In other words, small block size imposes heavier overhead on *namenode* and thus the maximum number of applications it can serve concurrently is drastically reduced. So HDFS is not optimized for managing small blocks and the recommended block size setting is 64MB, 128MB or higher. In addition, if the data size of a task is too small, the overhead of preparing for, starting up and cleaning up the task may dominate the overall run time. On the contrary, if blocks are too big, task granularity is coarse and load balancing and concurrency deteroitate. No matter how big each block is, it is fixed once given and therefore task granularity is consequently fixed. The sizes of records may vary so that the numbers of records stored in different blocks may differ even if block sizes are the same. This simple and intuitive implementation strategy has several drawbacks.

Firstly, it limits the degree of parallelism that can be achieved. The number of map tasks is fixed given an input data size, an input format and a block size. This imposes a limitation on the concurrency of a parallel job, because even if the number of available nodes is larger than that of map tasks, not all available nodes can be utilized. Traditional load balancing techniques for distributed systems [100] [91] do not help here because they assume that tasks are given and cannot be modified.

Secondly, MapReduce assumes that all map tasks of a job undertake the same amount of work. This assumption may not hold for several reasons. The nature of a map operation may result in skewed execution time even if map tasks process the same amount of data. In addition, each task may process data of different sizes if a user-defined input format is used. Moreover, map tasks may slow down because of process hanging, software bug, and system fluctuation. In clusters, underlying hardware may be heterogeneous and the time taken to run a task may be drastically different depending on the capability of the node the task is dispatched

to. To sum up, the work required by different tasks cannot be assimilated even if they belong to the same parallel job.

Cluster resource usage varies depending on workload characteristics. Usually severs are neither completely idle nor fully loaded. A study [25] done by Google shows that server utilization is between 10% and 50% most of the time based on profiling results of 5000 servers during a six-month period. As a result, scheduling algorithms should fully exploit parallelism to utilize available resources. Also, skewed task execution time has been observed in real studies. In the study of parallel BLAST, one task took about 18 hours to complete while other tasks took 30 minutes on average [80].

The above two drawbacks prohibit Hadoop from making full use of available resources even if they are idle. We mitigate it by dynamically splitting map tasks according to resource availability. Our goal is to minimize the average job turnaround time which is defined as the time between job submission and job completion. This metric directly reflects how users perceive the performance of a system, compared with throughput that measures performance from the perspective of system owners. Analysis of collected data from real Hadoop clusters shows that most Hadoop jobs are map-only [73]. So in our study, we only consider map-only jobs. We come up with Bag-of-Divisible-Tasks model and propose two new mechanisms - *task consolidation* and *task splitting* which dynamically adjust the granularity of tasks. Detailed task splitting algorithms are proposed for single-job scenarios where prior knowledge is known or unknown. After that, multi-job scheduling is investigated and Shortest-Job-First strategy and task splitting are integrated together in our proposed algorithms. Finally extensive simulation experiments are shown with performance comparison.

5.2 Dynamic Granularity Adjustment

Resource Model In Hadoop, each slave/worker node hosts a fixed number of *map slots*, which determines the maximum number of map tasks a node can run simultaneously. If the number of map slots is too small, resources cannot be fully utilized. If it is too big, severe resource use contention may happen and overhead is increased. For either case, performance is not optimal. We assume the number of map slots per node is perfectly tuned, while how to tune it is out of our scope.

Task Model We propose *Bag-of-Divisible-Tasks*, derived from *Bag-of-Tasks* [16] [111], as our task model.

We use Atomic Processing Unit (APU) to represent a segment of processing that cannot be parallelized. Then we call a nonempty set of APUs a *divisible task* such that it could be divided into sub-divisible-task(s) (or sub-task for short). Each job is modeled as a bag of *independent* divisible tasks. And from now on, we use divisible task and task interchangeably if no confusion under context. APUs may be heterogeneous in that data size and processing time vary. Given a set of independent APUs derived from a problem domain, how to organize them into tasks has significant impact on performance. The optimal solution depends on both the characteristics of APUs and real-time system load. If tasks are too coarse-grained and therefore too large, load unbalancing is likely to happen caused by large variation of task execution time. If tasks are too fine-grained and therefore too small, overhead and actual processing time get comparable and latency becomes significant.

In MapReduce, each map operation is considered as an APU. The limitation of default Hadoop implementation results from the fixed granularity of map tasks driven by data blocks. Job turnaround time is affected by not only data size but also other factors, such as system fluctuation and hardware heterogeneity. We propose *task splitting* and *task consolidation* to mitigate load unbalancing and fully utilize available resources. Task splitting is a process that a task is split to spawn new tasks. Meanwhile input data is also split accordingly so that each newly spawned task processes a portion of it. After a task T is split, m new tasks $\{T_1, T_2, \ldots, T_m\}$ are spawned and T itself becomes task T_0 with smaller input. Equations (5.1) and (5.2) hold where $UI(T)$ is the unprocessed input data of a task T. The processing that has been done by a task is not re-done after it is split.

$$UI(T) = UI(T_0) \cup UI(T_1) \cup UI(T_2) \cup \cdots \cup UI(T_m) \tag{5.1}$$

$$\forall i, j \; 0 \leq i < j \leq m \; UI(T_i) \cap UI(T_j) = \emptyset \tag{5.2}$$

Task consolidation is the inverse process, in which multiple tasks are merged into one task. Formally, if a set of tasks $\{T_1, T_2, \ldots, T_n\}$ are merged into a single task T, equation (5.3) holds.

$$UI(T_1) \cup UI(T_2) \cup \cdots \cup UI(T_n) = UI(T) \tag{5.3}$$

Task consolidation and split can be used to adjust task organization to adapt system environment changes.

They make scheduling more flexible and robust. If tasks are split too aggressively, overhead of splitting and task management may outweigh benefit of higher concurrency. So being splittable does not mean task splitting is beneficial. Based on the fact that tasks usually run much longer than APUs, we make a simplification that APUs are arbitrarily small.

5.2.1 Split Tasks Waiting in Queue

In this section, we give examples about how to split and consolidate tasks that are waiting in queue. Running tasks are considered in the next section. Our task splitting process considers all map tasks in queue, which may belong to different jobs.

If there are no available map slots, no map task in the queue is split or consolidated.

If the number of available map slots is smaller than that of map tasks in queue, one possible strategy is to consolidate map tasks so that all of them can be dispatched immediately. The data to be processed is the same no matter whether map tasks are consolidated or not. Overall overhead of map task start-up and teardown is reduced because there are fewer tasks after consolidation. One potential drawback brought up by consolidation is the loss of data locality. The more map tasks are consolidated, the smaller the possibility becomes that input blocks of all consolidated tasks are located on the same node. As a result, the amount of data transferred from remote nodes increases. So the optimal decision relies on the tradeoff between task overhead and data transfer cost. Plot (b) in Fig. 5.1 shows an example. Three map tasks T_1, T_2, and T_3 are waiting and two nodes are available. So we can schedule two map tasks at most immediately. If we consolidate two map tasks, all map tasks can be scheduled to run immediately. In the plot, map task T_2 and T_3 are consolidated into map task $T_{2\text{-}3}$ which is dispatched to the node where block B_2 is stored. Block B_3 is remotely accessed by task $T_{2\text{-}3}$.

If the number of available map slots is larger than that of the map tasks in queue, the map tasks can be split to spawn new tasks to fill idle map slots. Resultant benefits include higher parallelism and better load balancing. As the number of map tasks increases, the overall task start-up and teardown overhead increases as well. Another disadvantage is that data locality may become worse. If a map task can be dispatched to a node where its input block is stored, one of its spawned map tasks is guaranteed to be able to be dispatched to that node while others may or may not be dispatched to it depending on map slot availability. Otherwise

none of its spawned map tasks after split can be dispatched to the node if they are run immediately. Plot (c) in Fig. 5.1 shows an example of task splitting. Initially there are four available nodes and three map tasks T_1, T_2, and T_3. Only three nodes can be utilized if the default MapReduce scheduling algorithm is applied. Instead, task T_3 is split to tasks $T_{3.1}$ and $T_{3.2}$ and all tasks are scheduled. Tasks $T_{3.1}$ and $T_{3.2}$ share the same input block B_3 but process different portions. Compared with the situation that splitting is not applied, task $T_{3.2}$ needs to access B_3 remotely but all nodes are utilized. One way to mitigate the data locality problem is data replication. When there are multiple copies of a block, the possibility is larger that data-local scheduling is achievable after task spitting. One extreme case is each block is replicated on all nodes so that the data locality becomes less significant.

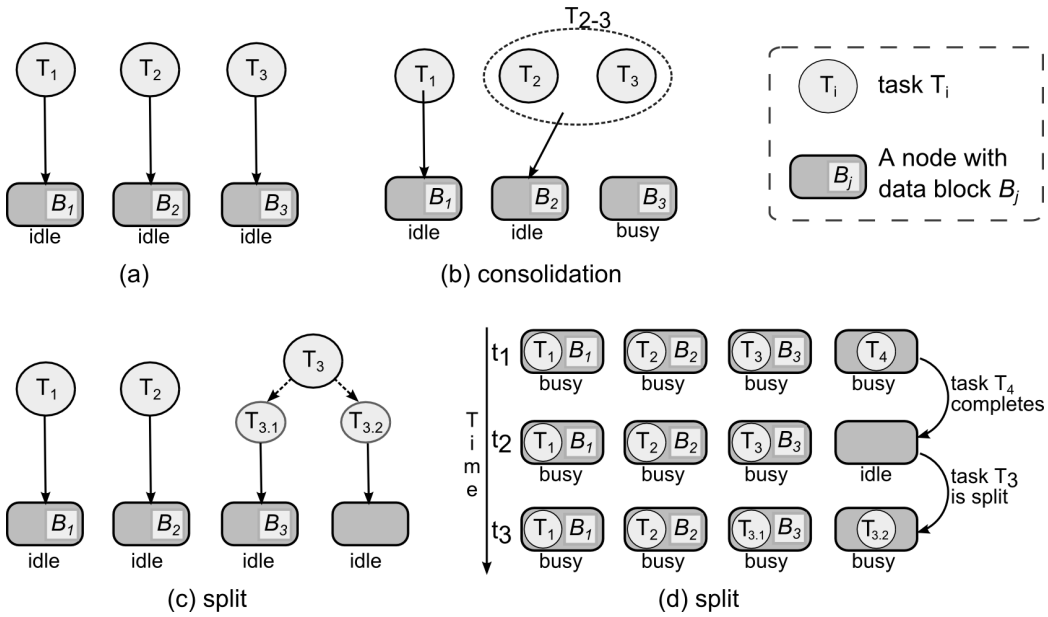


Figure 5.1: Examples of task splitting and task consolidation. Arrows are scheduling decisions. Each node has one map slot and block B_i is the input of task T_i

5.2.2 Split Running Tasks

Besides tasks waiting in queue, running tasks can also be split dynamically to improve performance. When tasks are scheduled and running, computation time skew may slow down the progress of the whole job. Task splitting can be applied dynamically during task execution to offload some processing to other available map slots. Plot (d) in Fig. 5.1 shows an example. At time t_1, four tasks are running. At time t_2, task T_4

completes and the slot originally occupied by task T_4 becomes available while the other three tasks are still running. Task T_3 is chosen to spawn a new task $T_{3.2}$ which is scheduled to the available slot released by completed task T_4. Again, all nodes are utilized but task $T_{3.2}$ accesses its input data (a portion of block B_3) remotely.

5.2.3 Summary

The previous two mechanisms can be combined together to adjust all unfinished tasks (waiting tasks + running tasks), which achieves continuous optimization during whole lifetime of jobs.

Task consolidation reduces the number of tasks to manage and schedule, which is highly beneficial if task management overhead is high and task start-up and teardown overhead is comparable to the actual execution time. We assume task execution time is significantly longer than task start-up and teardown time. If this does not hold, blocks can be enlarged to increase task granularity.

Task splitting is beneficial when the loss of data locality does not impose critical performance degradation. When data are replicated on every node, the data access time is similar no matter where a task is dispatched if data access contention (e.g. multiple tasks access different data stored on the same node) is not severe. If data access contention is severe, the number of map slots on each node can be tuned appropriately to achieve the optimal tradeoff between concurrency and resource use contention, so that data access does not affect scheduling much. This conclusion also holds when jobs are CPU-intensive and the data access cost is negligible. In other words, if the ratio of computation to data access is large, the computation factor is critical and other factors, such as disk I/O and network I/O, can be ignored. We focus on CPU-intensive jobs in the following discussions.

5.3 Single-Job Task Scheduling

First, we consider the task scheduling problem when only one job is running at most at any time. In the next section, multi-job cases are discussed. The following algorithm shows how task splitting is hooked into task scheduling process.

Algorithm skeleton:

```
while isRunning = true:
  split_tasks();
  schedule_tasks();
```

In the beginning of each scheduling iteration, task splitting is applied if needed. This step makes tradeoffs between concurrency and overhead. Then an existing task scheduling strategy (e.g. Hadoop's data locality based scheduling) is used to schedule tasks. So task splitting can be seamlessly integrated with existing schedulers. We focus on the task splitting process and present our proposed solutions for the cases where the prior knowledge about workload is known and unknown. We summarize issues shown below that need to be solved to address the problem.

1. When to trigger task splitting
2. Which tasks should be split and how many new tasks to spawn; and
3. How to split

5.3.1 Task Splitting without Prior Knowledge

When no prior knowledge is known about execution time, a strategy we term Aggressive Splitting (AS) is proposed.

5.3.1.1 When to trigger task splitting:

The goal of task splitting is to shorten the average job turnaround time by utilizing as many nodes as possible. Assume the scheduler is invoked at time t, task splitting decision is made if inequality (5.4) is satisfied where $N_{miq}(t)$, $N_{run}(t)$ and N_{ams} are the number of map tasks in queue at time t, the number of running map tasks at time t and the total number of map slots respectively. Inequality (5.4) means there will be idle map slots even if all tasks in queue are scheduled to run immediately. In this case, the default scheduling strategy cannot use all idle slots. So the task splitting process should be initiated. Otherwise, it does not make sense to split tasks because there are no idle slots where newly spawned tasks can run. This will not make long-running

tasks become "stragglers" because our task splitting process is invoked continuously and long-running tasks will become candidates of splitting targets whenever there are idle slots.

$$N_{miq}(t) + N_{run}(t) < N_{ams} \tag{5.4}$$

5.3.1.2 Which tasks should be split and how many new tasks to spawn:

We evenly allocate available map slots to unfinished tasks. Without prior knowledge, what we do is divide the number of idle map slots by the number of unfinished tasks to calculate how many new tasks to spawn for each task on average. Then tasks are split one by one until no map slots are idle. The algorithm skeleton is shown in Fig. 5.2.

```
UTS:set ← unfinished tasks
IMS:int ← number of idle map slots
MST:int ← ⌈ |UTS| / IMS ⌉
for each task T in UTS:
  if IMS ≤ 0: break
  if IMS < MST:
    NS ← split(T, IMS)
  else
    NS ← split(T, MST)
IMS ← IMS - NS
```

Figure 5.2: Algorithm skeleton of Aggressive Split

Function $split(T, N)$ splits task T to spawn N new tasks at most. Depending on the map slot availability, splitting policy and overhead, the actual number of spawned tasks may be smaller than N. The actual number is returned from the function call so that the following code can update the number of available map slots accordingly. The implementation of function *split* is described in next section.

5.3.1.3 How to split:

Given a task and the maximum number of new tasks it may spawn, this section solves the problem of how to split. Firstly, the number of new tasks is adjusted so that it does not exceed the number of the available map slots. Each data block is logically split to equally sized sub-blocks. We constrain that tasks processing one sub-block are not splittable. In other words it specifies the smallest granularity of spawned tasks. For a task T, the total number of sub-blocks, the number of sub-blocks that have been processed or are being processed, and the number of new tasks to spawn are denoted by $TS(T)$, $PS(T)$, and $NT(T)$ respectively. Since we don't have prior knowledge about the map execution time, we blindly spawn new tasks so that each one processes the same amount of data.

$$\text{sub blocks per task} = \frac{TS(T) - PS(T)}{NT(T) + 1} \tag{5.5}$$

The remaining work is evenly divided among the task being split and newly spawned tasks. The principle is to make them all complete simultaneously if map operation execution time is homogeneous theoretically. To avoid inefficiency caused by spawning small tasks, a threshold is set to prevent small tasks being split. The optimal threshold depends on workload and map operation characteristics. It is our future work to make the threshold automatically tuned.

5.3.1.4 Complexity:

The whole task list is scanned at most once, so time complexity is $O(n)$ with regard to the number of tasks.

5.3.2 Task Splitting with Prior Knowledge

Now we assume that prior knowledge about task execution time is known. By prior knowledge, we mean that *Estimate Remaining Execution Time* (ERET) is known or predictable. ERET indicates how long a task will run before completion approximately. In [120], a simple algorithm is proposed to estimate finish time of MapReduce tasks. How to calculate ERET is out of our scope. We propose *Aggressive Split with Prior Knowledge* (ASPK) to optimize job turnaround time.

5.3.2.1 When to trigger task splitting:

The same algorithm from last section can be reused here.

5.3.2.2 Which tasks should be split and how many new tasks to spawn:

The ways to split tasks are not unique. The number of task splitting done during the whole lifetime of a job should be as small as possible without degrading performance. Fig. 5.3 demonstrates different ways to split tasks to achieve the same turnaround time. Graph (a) shows a scenario where there are two running tasks - T_1 and T_2, one idle slot and no waiting tasks. ERET of T_1 and T_2 is t and $2t$ respectively. If overhead and data locality are negligible, we definitely should split tasks to fill the idle slot. We can split task T_1 to spawn a new task and both will run for period t before completion, which is demonstrated in (b). At time t all tasks complete. Another way shown in (c) is to split task T_2 to spawn a new task and both will run for period $t/2$. At time $t/2$, two slots become idle and task T_1 is split to spawn two new tasks each of which runs for $(2t - t/2)/3 = t/2$. In both cases, the final job turnaround time is t. However the number of spawned tasks is different. In (b), one task is spawned while in (c) three tasks are spawned. More task splittings incur higher probability to degrade performance and destabilize the system. In the example, (b) is preferred to (c).

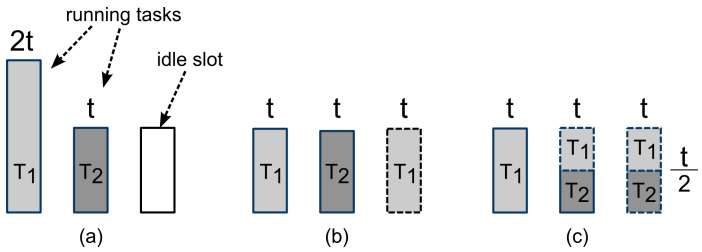


Figure 5.3: Different ways to split tasks (Processing time is the same). Dashed boxes represent newly spawned tasks.

Tasks that complete last determine when a job finishes. For jobs with tasks that have highly varied execution time, the scenario should be avoided that few long tasks last much long after other short jobs complete. When long running tasks exist, to split tasks with small ERET generates smaller tasks, which doesn't affect job turnaround time. So our heuristic is that tasks with large ERET should be split first so that they do not become "stragglers".

Firstly, the tasks with small ERET are filtered because to split a task that will complete very soon does not yield much benefit. In addition, task filtering is an optimization step that reduces the number of map tasks considered by following steps for faster processing. Secondly remaining tasks are sorted by ERET in descending order. After that, tasks are clustered into $\{C_1, C_2, \ldots, C_m\}$ according to ERET so that tasks with similar ERET belong to the same cluster. Each cluster C has several pieces of information including task list ($C.TS$), the number of tasks ($C.Count$), the sum of ERET ($C.ERET$) and the average of ERET ($C.AE$). We go through task clusters one by one to evaluate whether task splitting is beneficial. Initially, we only consider the tasks in cluster C_1. The tasks in C_1 are split to fill all idle slots, and the estimated task execution time T_1 after splitting is calculated. If T_1 is larger than $C_2.AE$, it doesn't benefit to split the tasks contained in following clusters and the estimated execution time of newly spawned tasks is set to $C_2.AE$. If T_1 is significantly smaller than $C_2.AE$, the spawned tasks are too small compared with the tasks in C_2 . So we consider the tasks from both C_1 and C_2 for splitting. Time T_2 is calculated and compared with $C_3.AE$. If T_2 is much smaller, we consider C_1, C_2, and C_3. This process is repeated until T_i is larger than or comparable to $C_{i+1}.AE$ or all clusters have been included. The algorithm skeleton is shown in Fig. 5.4.

```
1  IMS ←  number of idle map slots
2  UTS ←  unfinished tasks
3  FTS ←  filterTasks (UTS)
4  STS ←  sortByERET (FTS)
5  {C1,C2,...,Cm} ←  clusterTasks (STS)
6  sumERET ←  0, count ←  IMS
7  for cluster Ci 1≤ i≤ m:
8    sumERET += Ci.ERET
9    count += Ci.Count
10   avgERET = sumERET / count
11   if i = m: break
12   if avgERET << Ci+1.AE:
13     continue
14   else
15     break
```

Figure 5.4: Algorithm skeleton of Aggressive Split with Prior Knowledge

Filtering Ideally, how tasks are filtered should depend on the ERET of unfinished tasks. A pre-set threshold is not flexible enough to capture task characteristics. Instead, we calculate the *optimal remaining job execution time* (ORJET) by assuming that all unfinished tasks are split to use all available slots. Total ERET is gained by adding up ERET of all unfinished tasks. It is divided by the total number of map slots (including both occupied and idle slots) to get ORJET. ORJET measures how long a job will run before completion optimally. Then ERET of each task is compared with ORJET. If task ERET is significant smaller than ORJET, the task is filtered out. Towards the end of job execution, ORJET becomes increasingly small because running tasks are close to completion and more slots are released. In this situation, task splitting is not beneficial because the overhead of task splitting outweighs the potential gain of higher concurrency. So we filter out tasks that are close to completion without affecting overall performance. Thus the filtering process is adaptive to workloads of different types.

Clustering Task clustering algorithm is designed to group tasks with similar ERET and separate tasks with significantly different ERET. Existing clustering algorithms, such as K-means, Expectation- Maximization and agglomerative hierarchical clustering, from the machine learning community can be used without modification. Considering that scheduling routine is called frequently and its performance is critical to the whole system, we favor simple linear algorithms. The tasks being clustered have been ordered by ERET, which guarantees that tasks belonging to the same cluster are consecutive in the task list. Our current algorithm requires that the task list is scanned once by moving a "cursor" from beginning to end. A running list is maintained to contain tasks that are before the "cursor" and belong to the current cluster. If ERET of the task pointed by the cursor is much smaller than the average ERET of the current cluster, the current cluster is added to the global cluster set and a new cluster is created which initially only contains the task pointed by cursor. This guarantees the maximal ERET of tasks within a cluster is significantly smaller than the average ERET of tasks within previous cluster.

5.3.2.3 How to split:

The way to split tasks can be optimized if we also have prior knowledge about mean task execution time, network throughput, disk I/O throughput, etc. For task T, disk I/O cost, network I/O cost, and computation cost are denoted by $DIO(T)$, $NIO(T)$, and $Com(T)$ respectively. So the total time is $DIO(T) + NIO(T) +$

$Com(T)$, if these operations don't overlap. The task being split is denoted by T_{cur}, and the newly spawned tasks are T^1_{cur},T^2_{cur},...,T^N_{cur}. Ideally, equation (5.6) should be satisfied to make tasks complete simultaneously after splitting.

$$\begin{aligned} & DIO(T^1_{cur}) + NIO(T^1_{cur}) + Com(T^1_{cur}) \\ & = \cdots\cdots \\ & = DIO(T^N_{cur}) + NIO(T^N_{cur}) + Com(T^N_{cur}) \\ & = DIO(T_{cur}) + NIO(T_{cur}) + Com(T_{cur}) \end{aligned} \tag{5.6}$$

Because we assume $DIO(T)$ and $NIO(T)$ are negligible compared to $Com(T)$, the above equation is converted to (5.7).

$$Com(T^1_{cur}) = Com(T^2_{cur}) = \cdots = Com(T^N_{cur}) = Com(T_{cur}) \tag{5.7}$$

So the unfinished work of task T is evenly distributed to itself and the newly spawned tasks after splitting.

5.3.2.4 Complexity:

In ASPK, the complexity of sorting is $O(n \cdot \log n)$ and that of other operations is not greater than $O(n)$. So the overall complexity is $O(n \cdot \log n)$. However, sorting can be further optimized considering that in each iteration, except the first one, tasks are mostly ordered.

5.3.3 Fault Tolerance

Our proposed algorithms do not handle fault tolerance directly. Task splitting is not enough to cope with situations where some tasks stall or fail due to hardware failure, severe system fluctuation or hanging process. We integrate speculative execution to solve the problem. Speculative execution component monitors the system using heartbeat messages and starts duplicate tasks automatically to replace failed tasks. Now we have a complete solution which can speed single-job execution by splitting relatively long tasks and speculatively re-execute failed tasks.

5.4 Multi-Job Task Scheduling

We put multi-job scheduling into the context of classic queuing theory. We adopted $M/G/s$ model [74]. Jobs arrive according to a homogeneous Poisson process. Job execution time is independent and may follow generic distributions. Also there is more than one server in the system. One difference from the classic model is that a job may use multiple servers during its execution and the execution time depends on the number of used nodes. We propose *Greedy Task Splitting* (GTS) which minimizes the run time of each job by splitting tasks to occupy all map slots and making tasks of last round complete simultaneously. Because each job uses all available nodes, following jobs cannot execute until the current running job completes. In other words, the queue time of some jobs is increased compared with non-GTS scheduling. As a result, the change of job turnaround time depends on both decrease of job execution time and possible increase of job queuing time. We will show that GTS gives optimal job turnaround time.

5.4.1 Optimality of Greedy Task Splitting

Fig. 5.5 shows two examples of execution arrangement of a job J. In (a), job J starts at $S(J)$ and completes at $F(J)$. It uses all resources during the execution. In (b), the processing is grouped to four segments - 1, 2, 3 and 4. Now we formulate the scheduling model. C denotes the capacity of a certain type of resource in the system. n denotes the number of jobs to run. S_i ($1 \leq i \leq n$) denotes the total resource requirement of job i. The resource usage function $r(t, i)$ represents the amount of resource consumed by job i at time t. Constraints are shown in (5.8), (5.9), and (5.10): and objective function is shown in (5.11).

$$t \geq 0 \tag{5.8}$$

$$\forall t, \sum_{i=1}^{n} r(t, i) \leq C \tag{5.9}$$

$$\forall 1 \leq i \leq n, \sum_{t=0}^{+\infty} r(t, i) \geq S_i (\text{or} \int_{t=0}^{+\infty} r(t, i) \mathrm{d}t \geq S_i) \tag{5.10}$$

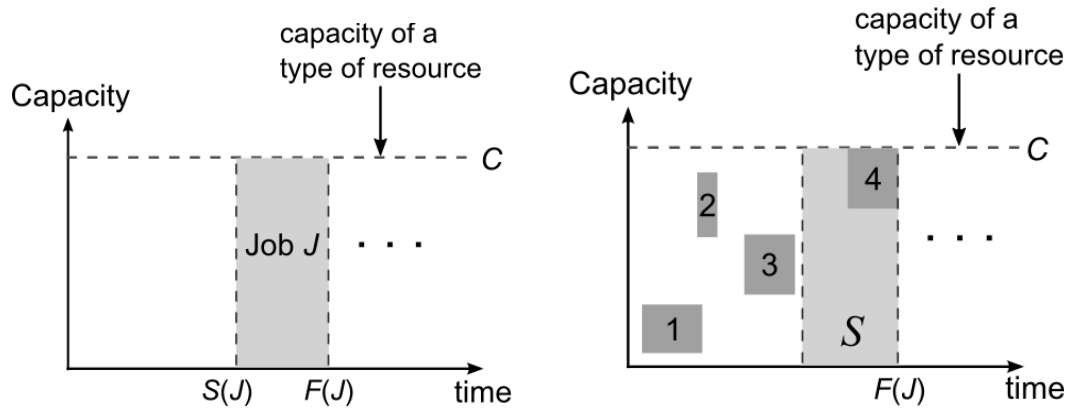
$$min(\sum_{i=1}^{n} max_t \, r(t, i) \neq 0) \tag{5.11}$$

Inequality (5.9) means that at any moment, the resource consumed by all jobs must not be more than the capacity. Inequality (5.10) means that the sum of resource consumption by any job across time is not less than the requirement of the job. The ideal case is that the actual resource consumption is equal to the resource requirement, which means no overhead is incurred. In the objective function, $max_t\, r(t,i) \neq 0$ is the turnaround time for job i. So our goal is to minimize overall job turnaround time.

Firstly we will show that once a job starts running, it should complete as soon as possible by using all available resources. Secondly we will convert this problem to $n/1$ (n jobs/1 machine) scheduling problem solved in [45].

Given a job J, its start time $S(J)$ and its completion time $F(J)$, Fig. 5.5 shows possible strategies of execution arrangements. Execution arrangement of J affects completion time of other jobs. One fact is the start time of job J does not matter when $F(J)$ is fixed. Intuitively, all parts of execution of Job J should be placed as close to $F(J)$ as possible. In plot 5.5b the execution of job J is interspersed along time axis. The execution arrangement demonstrated in plot 5.5b can be converted to that demonstrated in plot 5.5a by interchanging interspersed execution segments of job J (e.g. marked by 1, 2 and 3 in the plot) and the execution segments of other jobs falling into the continuous area S. After the interchange, the completion time of those affected jobs either does not change or becomes earlier because their changed execution segments start earlier. This interchange process can be iterated until each job utilizes all resources during its execution (see Fig. 5.5 for an example). In each iteration, only one job is considered. The whole process makes overall turnaround time monotonically decrease regardless of the order of jobs picked during iterations.

However, different job execution orders may result in different overall turnaround time. Fig. 5.6 shows one example. The next question is how to determine job execution order so that the objective function is minimized. Because at any moment only one job consumes all resources, we can view the whole system as a single big virtual node. This problem becomes the $n/1$ problem (n jobs / 1 machine) solved in [45]. Shortest-job-first strategy gives overall optimal turnaround time. So jobs should be executed in the ascending order of execution time.



(a) Job J uses all available resources during execution

(b) Execution of job J is segmented and interspersed

Figure 5.5: Different ways to arrange the execution of a job

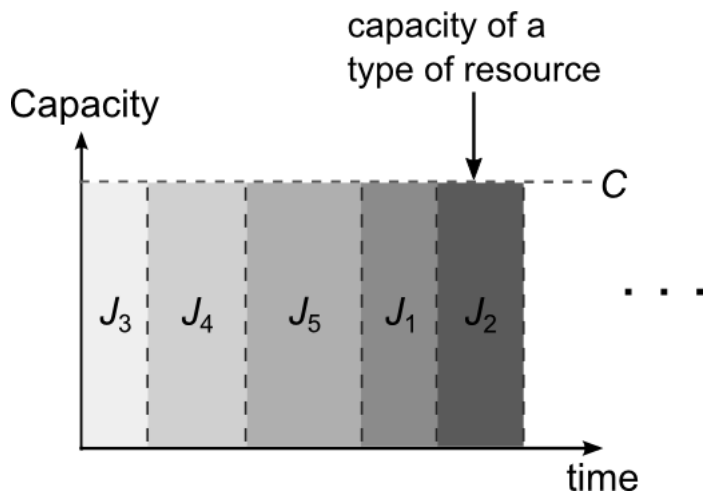


Figure 5.6: Multiple scheduled jobs (Each uses all resources for execution)

5.4.2 Multi-Job Scheduling

Given a number of jobs to run, the algorithm skeleton of *Shortest Job First Scheduling* (SJFS) is shown below. *Serial Execution Time* (SET) denotes how long a job runs serially.

Algorithm skeleton of SJFS

```
order jobs by SET in ascending order
schedule jobs in turn
```

If we know the SET of all jobs to be run, we can apply SJFS directly. However, in real systems, it is hard, if not impossible, to know all jobs to run ahead. To cope with the uncertainty, we use *Non Overlapped Periodic Shortest Job First Scheduling* (NOPSJFS) in which SJFS is run periodically. Let I be interval that SJFS is called. Scheduling decisions are made at time 0, I, $2I$, ... Let J_t be a set of jobs that are submitted at or earlier than time t. At time $n \cdot I$, SJFS is applied to the job set $J_{n \cdot I} - J_{(n-1) \cdot I}$. So jobs that are scheduled at time $n \cdot I$ only include those submitted between time $(n-1) \cdot I$ and $n \cdot I$. The jobs submitted prior than time $(n-1) \cdot I$ are not considered at all even if some of them are still waiting in the queue. This strategy makes each job be scheduled just once and jobs scheduled during different period do not overlap. But unexpected system fluctuation exists and prior knowledge of SET may be inaccurate. So the assumptions made when a job is scheduled may be rendered useless by the time it is dispatched to run. *Overlapped Shortest Job First Scheduling* (OSJFS) is proposed in which all jobs are considered that have been submitted but not completed yet. To avoid starvation of long jobs, an aging factor is associated with each job which measures how long a job has been waiting in the queue. Priority is positively correlated to aging factor. So the longer a job has waited, the higher its priority becomes.

5.5 Experiments

We conduct experiments using the MapReduce simulator mrsim [60] which is built on top of an event-driven framework. Table 5.2 shows the configuration of our simulated system. Data are placed randomly on nodes. Each node hosts only 1 map slot. We will assess *effectiveness* of our approaches. Hardware configuration affects absolute job turnaround time, but it does not affect comparison between our strategies and the default strategy.

Number of nodes	64	**Disk I/O - read**	40MB/s
Processor frequency	500MHz	**Disk I/O - write**	20MB/s
Map slots per node	1	**Network**	1Gbps

Table 5.2: Configuration of test environment

Two distributions are used to model the execution time of map operations - Gaussian distribution and step

functions abstracted from real workload trace. Firstly, we set up tests to show that our approach improves performance in single job environments.

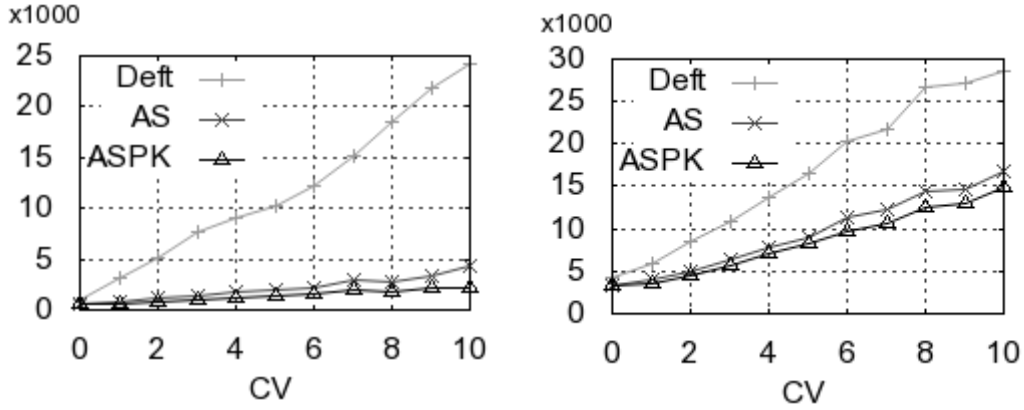
5.5.1 Single-job tests

In this set of tests, we investigate the effect of variation of map task execution time on performance. We design a micro-benchmark to measure the performance improvement of task splitting. Based on the total number of map slots and that of map tasks, two cases are considered.

When the number of map tasks is smaller than that of available map slots, the default strategy cannot utilize all resources. In the first test, we compose a job whose input data has 32 blocks each of which is 64MB. The cluster has 64 nodes. We assume that task execution time follows Gaussian distribution with negative values cut off. Mean is fixed and variance is varied which is an indicator of variation of execution time of map tasks. Baseline distribution is uniform distribution with mean and coefficient of variance (CV) is zero by definition. We let Gaussian distributions have the same mean and change variance to $(k \cdot \mu)^2 (1 \leq k \leq 10)$. So CV is between 1 and 10. Job turnaround time is shown in 5.7b. One observation is that job turnaround time increases as CV increases. That results from cut-off of negative values sampled from tested distributions. So the mean of sampled values is no longer μ and it increases slightly with CV. Both AS and ASPK improve performance significantly and performance gain increases with CV. AS incurs larger variation compared with ASPK. When CV is small, the difference between AS and ASPK is not significant. As CV becomes large, ASPK performs significantly better than AS. When CV is 10, ASPK performs better than AS by 50%.

Now, we increase the number of map tasks of a job to 200 to make it significantly larger than the number of map slots. The test environment is the same as the previous test. Fig. 5.7b shows results. The distributions of task execution time are the same as in previous test. Default scheduling has embedded support for load balancing. Whenever a map slot becomes available, it dispatches a waiting task to it. Because the execution time of map tasks is sampled from the same distribution, the sum of task execution time for different nodes follows the same distribution as well. In other words, mixture of long and short tasks dispatched to nodes naturally makes the load balanced during the early lifetime of the job. In the early phase of job execution, all map slots are occupied so that task splitting does not benefit. Towards the end of the execution, all tasks are either running or completed. Any released map slot cannot be utilized because there is no waiting task.

Then task splitting improves performance by rebalancing load. Considering task splitting is mostly applied near job completion, it may not benefit much. Test result shows that even in that situation, AS and ASPK improves performance by 50% at most. The larger CV is, the more efficient ASPK is compared with AS.



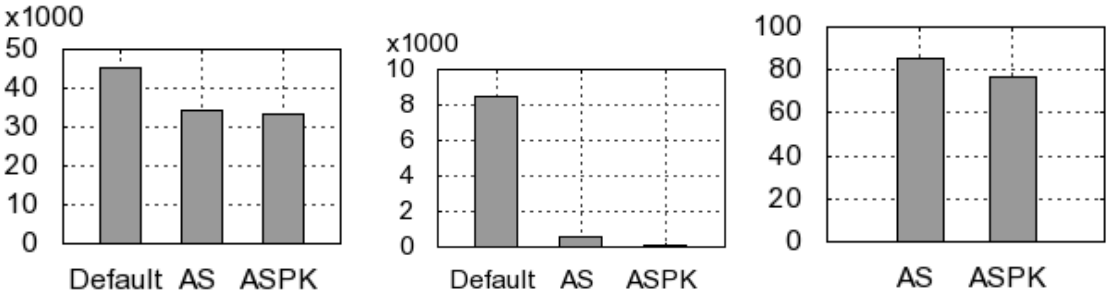
(a) job turnaround time (32 tasks)

(b) job turnaround time (200 tasks)

Figure 5.7: Single-Job test results (Gaussian distribution is used)

Besides synthesized workload, workload data collected in real clusters is also used. Concretely, we use cluster data published by Google [15]. It is analyzed in [35] to extract characteristics of jobs and tasks. One observation made is that task execution time for three types of jobs is bimodal. Around 75% of map tasks are short, running for approximately 5 minutes. Around 20% of map tasks are long, running for approximately 360 minutes. Execution time of the remaining 5% the map tasks is between 5 minutes and 360 minutes. This distribution is used to model task execution time in this test. *Slot completion time* is termed to describe when the last task run in a map slot completes. We measured both job turnaround time and the variation of slot completion time for all slots. Fig. 5.8 shows the results. AS and ASPK shorten job turnaround time by 20% - 30%. ASPK performs slightly better than AS by reducing job turnaround time by 5% - 10%. The standard deviation of slot completion time is shown in plot 5.8b. For the default scheduling, the value is 8521 seconds which indicates that the last round of map task execution results in severe load unbalancing. ASPK achieves the smallest standard deviation around 9 seconds, so that its histogram is almost invisible in the plot. This result is surprisingly good considering that the job runs for tens of thousands of seconds. For AS, the standard deviation is around 570 seconds. To figure out whether the best performance of ASPK is achieved by splitting much more tasks than AS, the number of spawned tasks is measured. Plot 5.8c shows that ASPK even has

smaller number of spawn tasks than AS. So ASPK achieves the shortest job turnaround time and the smallest variation of slot completion time by spawning fewer tasks. This means when prior knowledge is known the additional optimization done in ASPK is effective.



(a) Overall DLC (b) Reduction of DLC(in percent) (c) Reduction of DLC(in percent)

Figure 5.8: Single-Job test results (Real profiled distribution is used)

Above tests demonstrate that task splitting strategy improves performance significantly and the degree of improvement is related to characteristics of map tasks.

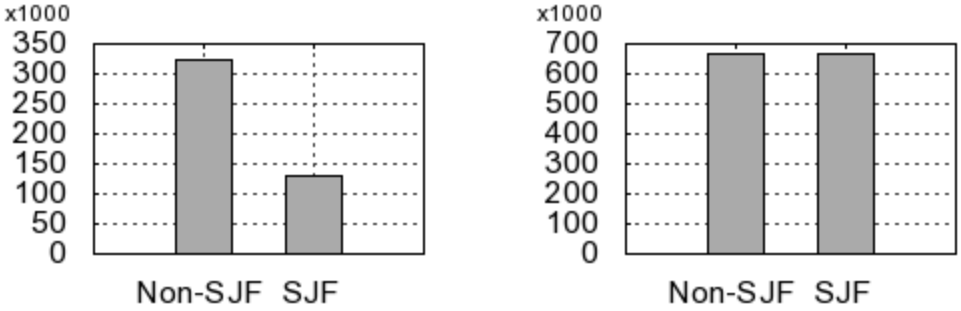
5.5.2 Multiple-job tests

As $M/G/s$ model is adopted for multi-job scenario, inter-arrival time of jobs follows exponential distribution. We generate a workload to have 100 jobs each of which is synthesized according to Google cluster data. We measure average job turnaround time with and without SJF policy applied. If interarrival time is longer than job execution time, on average one job is running at most at any time. Single-Job scheduling can be used directly. So we set mean of interarrival time to be much shorter than average job execution time.

In this test, all jobs have the same number of map tasks, which is equal to total number of map slots, so that each job can occupy all map slots. The execution time of tasks belonging to a job is the same. 75% of jobs are short, 20% of jobs are long and 5% of jobs are medium. 100 jobs are generated. Task splitting in this test does not benefit much because all map tasks of a job complete almost simultaneously and load unbalancing occurs rarely. Results are shown in Fig. 5.9. Non-SJF scheduling and SJF scheduling have comparable makespan. SJF decreases the average job turnaround time by 63%.

Then we tested the case where different jobs have the same serial execution time. Obviously SJF strategy

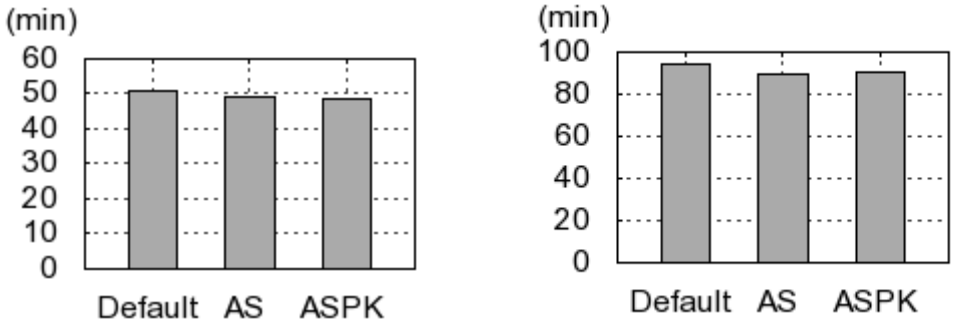
does not make sense because all jobs are equally long. So we ignore SJF and evaluate task splitting strategies. Task execution time of each job follow the same distribution extracted from Google cluster data. 100 jobs are generated and all slots are used at any time except near completion. Fig. 5.10 shows that both job turnaround time and makespan are shortened by 5% - 10%. One well-known fact is that if a system is fully loaded, it is harder to make optimization compared with the situation where a system is partially loaded. Our test results show that even if the system is fully loaded and SJF is useless, task splitting still benefits. Considering that study in Google shows CPU utilization ratio is between 20% and 50% for their production clusters, task splitting will give more improvement in real clusters than in this test.



(a) Average job turnaround time

(b) Makespan

Figure 5.9: Multi-Job test results (task execution time is the same for a job)



(a) Average job turnaround time

(b) Makespan

Figure 5.10: Multi-Job test results (job execution time is the same)

5.6 Summary

In this chapter, we examined strategies for optimizing job turnaround time in MapReduce. Firstly, we analyzed the MapReduce model and its Hadoop implementation, and found that the way map operations are organized into tasks in Hadoop has several drawbacks. Then we proposed task splitting, which is a process to split unfinished tasks to fill idle map slots, to tackle those problems. For single-job scheduling, Aggressive Scheduling (AS) and Aggressive Scheduling with Prior Knowledge (ASPK) were proposed for cases where prior knowledge is known and unknown respectively. For multi-job scheduling, we proved that combination of Shortest-Job-First strategy and task splitting mechanism gives optimal average job turnaround time if tasks are arbitrarily splittable. Overlapped Shortest-Job-First Scheduling (OSJFS) was proposed which invokes basic short-job-first scheduling algorithm periodically and schedules all waiting jobs in each cycle. We also conducted extensive experiments to show that our proposed algorithms improve performance significantly compared with the default strategy. One thing we may explore in the future is how task splitting and consolidation can benefit IO intensive applications. Tradeoffs between data access concurrency and data locality need to be investigated to achieve optimal performance. In addition, it will be helpful to implement our algorithms in Hadoop and run some real applications to show the usefulness of our algorithms.

6

Resource Utilization and Speculative Execution

Traditional batch queuing systems adopt reservation-based resource allocation mechanisms. Although a cluster is shared by different users, the use of individual nodes is mostly exclusive among jobs.

MapReduce takes a different approach in that all nodes are shared among jobs submitted by users and multiple tasks from different jobs can run concurrently on individual nodes to explore the processing capability of modern multi-core processors. Users cannot reserve nodes for a specific period of time explicitly, and it is the runtimes' responsibility to decide where each task will be scheduled based on run-time resource availability. In this sense, MapReduce adopts dynamic scheduling while batch queuing systems adopt static scheduling. As we discussed above, each node has a number of map and reduce slots where tasks can run. One can think of it this way: each slot is reserved for the prospective tasks that will be assigned to it. A slot gets *occupied* when a task is assigned to it, and gets *released* when the task completes. Usually, the number of slots is tuned to balance parallelism and resource usage contention. When all slots are occupied by tasks, severe resource contention should be avoided and an optimal resource utilization ratio should be achieved. Each slot corresponds to a portion of resources that are consumed by the task running in the slot. Roughly the number of occupied slots is proportional to the resource utilization ratio for each node. A consequence is that a significant portion of resources are wasted when the whole system is underloaded (i.e. there are no enough tasks to fill all slots). We propose *Resource Stealing* in which running tasks steal resources corresponding to idle slots in order to shorten execution time. Whenever an idle slot is assigned with a task, the stolen resources are returned so that the normal execution of the newly assigned task is not influenced. In addition, We propose and evaluate several strategies to reasonably allocate unused resources to running tasks.

Speculative execution is adopted by MapReduce to support fault tolerance. The master node keeps track of the progresses of all scheduled tasks. When it finds a task that runs unusually slow compared with other tasks of the same job, a speculative task is launched to process the same input data with the hope that it will complete earlier than the original task. Whenever a task completes, all other redundant tasks that process the same data are immediately terminated. The slowness of task execution may be caused by faulty nodes, process hang, etc. The speculative tasks, which complete later than original tasks and thus do not benefit the overall job execution, are termed *non-beneficial speculative tasks*, the number of which should be minimized to maximize efficiency. To make speculative execution effective and efficient, two issues need to be addressed: i) How to identify slow tasks; ii) among slow tasks, which should be speculated. In Hadoop, the default mechanism incurs the execution of many non-beneficial speculative tasks and is inefficient.

Hadoop 0.21 integrates the algorithm Longest Approximate Time to End (LATE) proposed by Zaharia et al. [120]. Progress rate is calculated by dividing task progress by execution time. The tasks whose progress rates are one standard deviation slower than the mean of all tasks of the same job are identified as slow tasks and subject to being speculated. Among slow tasks, the one whose completion time is estimated to be farthest is speculated. The heuristic is that tasks that hurt the overall job response time should be speculated. A major drawback is that LATE does not evaluate whether it is beneficial to launch a speculative task. The goal of speculative execution is to improve performance and therefore shorten job execution time. After a speculative task is launched, resources are possibly wasted if the corresponding speculated task actually completes earlier than it. If a task is progressing slow but close to completion, to launch a speculative task may not be beneficial because it needs to re-execute the processing that has been done by the slow task and may complete later than the slow task.

We propose BASE algorithm for Benefit Aware Speculative Execution which evaluates the benefit brought by speculative tasks before they are launched.

6.1 Resource Stealing

How to tune Hadoop parameters automatically has been studied in [72, 61]. In this paper, we assume the number of task slots are set optimally so that optimal resource utilization is achieved when all slots are

occupied. Resource utilization is proportional to the number of occupied slots approximately. According to the trace of production clusters [25], resource utilization is way below 100% and varies across time periods, so mostly there exist idle slots in large systems. It implies that the capability of resources can be further exploited to minimize execution time. The portion of the resources that sit idle on a slave node is termed *residual resources* which can be utilized without incurring severe usage contention or degrading overall performance. We can consider that residual resources are reserved for prospective tasks that will be assigned to currently idle slots. One advantage of resource reservation is that whenever a new task is assigned resource availability is guaranteed. However, an obvious drawback is that residual resources are left unused until new tasks are assigned.

We propose *resource stealing* to improve resource utilization. The resource usage of running tasks (if any) on each node is dynamically expanded or shrunk according to the availability of task slots. When there are idle slots, running tasks temporarily steal resources reserved for prospective tasks so that residual resources are fully utilized. If a node is perfectly loaded by using resource stealing, to assign a new task obviously will overload it and degrade the performance of currently running tasks. Our solution is to adjust the resource usage of running tasks by making them relinquish stolen resources proportionally. In this way, resource stealing does not violate the assumption made by Hadoop that resources are guaranteed for new tasks, which is critical to efficient Hadoop scheduling. To summarize, the overall philosophy is to steal residual resources if corresponding map/reduce slots are idle, and hand them back whenever new tasks are launched to fill the idle slots. From the perspective of the central task scheduler, idle slots are still idle and new tasks can be assigned to them, so resource stealing is transparent to the task scheduler and can be used in combination with any Hadoop scheduler directly such as fair scheduler and capability scheduler. Resource stealing is applied periodically with the up-to-date information of task execution and system status. So it is adaptive in the sense that it reacts to real-time changes of the system state.

6.1.1 Allocation Policies of Residual Resources

Given residual resources and the number of running tasks on a node, the next issue is how to distribute residual resources among running tasks, e.g. which tasks should get how much. The policies can range from simple to complex in their use of system state information. Complex policies have the potential to take full

advantage of the processing capability of each node. The disadvantages include high overhead cost and the risk that a well tuned policy may behave unpredictably when inaccurate state information is collected. We come up with several policies summarized in Table 6.1.

Policy	**Description**
Even	Evenly allocate residual resource to tasks.
First-Come-Most	The task that starts earliest is given residual resource.
Shortest-Time-Left-Most	The task that will complete soonest is given residual resource.
Longest-Time-Left-Most	The task that will complete latest is given residual resources.
Speculative-Tasks-Most	Speculative tasks are given residual resource.
Laggard-Tasks-Most	Straggler tasks are given residual resources.

Table 6.1: Allocation strategies of residual resources

Even: This policy equally divides residual resources among running tasks. It is inherently stable because of not relying on the collection or prediction of system state (and thus not impacted by the information inaccuracy). The fact that this policy is extremely simple does not necessarily indicate it will perform significantly worse than other more sophisticated policies.

First-Come-Most (FCM): This policy orders running tasks by start time. The task with the earliest start time is given residual resources. The heuristic is to make tasks complete in the order of job submission with best efforts.

Shortest-Time-Left-Most (STLM): Firstly, the remaining execution time of tasks is estimated, where different mechanisms can be plugged in. Here we adopt the same mechanism used in [120] which assumes each task progresses at a constant rate across time and predicts the time left based on progress rate and current progress (shown in (6.1) and (6.2)). The task with the shortest time left is given residual resources. The heuristic is to make close-to-completion tasks complete as soon as possible to release resources that can be allocated to long-running tasks. Therefore it increases the processing throughput.

$$progress\_rate = \frac{progress}{exec\_time} \tag{6.1}$$

$$time\_left = \frac{1 - progress}{progress\_rate} \tag{6.2}$$

Longest-Time-Left-Most (LTLM): This policy is the same as STLM except that the task with the longest time left is given residual resources.

Speculative-Task-Most (STM): Speculative execution in MapReduce aims to mitigate the impact of slow tasks by duplicate their processing on multiple nodes. The basic idea of STM policy is that speculative tasks are given more resources than regular tasks with the hope that their execution will be accelerated and they will not hurt the job execution time. Because speculative tasks are given more resources, they can run faster and will not be stragglers any longer hopefully. If there are no speculative tasks on a node, it falls back to the regular case and other policies can be applied. If there are multiple speculative tasks running on a node, residual resources are allocated to them evenly. Algorithm skeleton is shown in Fig. 6.1.

```
1  Input: Statuses of running tasks on a node
2  Output: resource assignment to tasks
3  Algorithm:
4  TS ← The set of running tasks on a node
5  ST ← ∅ # the set of speculative tasks
6  for T in TS:
7    if T is a speculative task:
8      ST = ST ∪ T
9  if ST is not empty:
10   allocate residual resources to tasks in ST
11 else
12   fall back to other allocation policies
```

Figure 6.1: Algorithm skeleton of STM

Laggard-Task-Most(LTM): In this approach, we do not distinguish between regular tasks and speculative tasks. Instead, for each job we use the estimated remaining execution time of all its scheduled tasks (both regular and speculative tasks) to calculate the *fastness* of a running task T using (6.3). Fastness reflects the expected order of task completion for each job; and a task with small *fastness* will complete later than a task with large *fastness*.

The *fastness* of a task cannot be computed locally by a slave node because it requires the information of all other tasks belonging to the same job. The master node maintains the statues of all tasks so that it is the ideal component to compute *fastness*. Each slave node reports the statuses (e.g. progress, failure) of its

running tasks to the master node with heartbeat messages. After collecting the information of all tasks, the master node calculates the *fastness* of each task and returns it to the corresponding slave node. Upon receiving *fastness* information, slave nodes order tasks by *fastness*. The tasks whose *fastness* is smaller than threshold *SlowTaskThreshold* (a user configurable parameter) are called *laggards* and given residual resources. If there are multiple laggards on a node, residual resources are evenly allocated to them.

$$fastness(T) = \frac{\#\ of\ tasks\ that\ will\ complete\ later\ than\ T}{\#\ of\ running\ tasks} \tag{6.3}$$

As we discussed, the motivation of speculative execution is to improve performance by running duplicate processing. There are several drawbacks. Firstly, if speculative execution is triggered, the completion of any task renders the work done by other duplicate tasks to be wasted. So the triggering of speculative execution should be avoided in the first place. Secondly, if the slowness of tasks is caused by intermittent and temporary resource contention, it is highly likely that they do not lag much behind and still complete earlier than their speculative tasks, which subverts the motivation of speculative execution. Thirdly, sometimes speculative execution deteriorates performance rather than improve it [120]. LTM reduces the invocations of speculative execution by proactively allocating more resources to laggards whenever possible and thus accelerating their execution. Fourthly, the tasks of a job may be heterogeneous intrinsically in that their execution time varies greatly depending upon both data size and the content of the data. For example, easy and difficult Sudoku puzzles have similar input sizes (9 x 9 grids) but require dramatically different amounts of computation. Speculative execution is not helpful because the efficiency variation is not mainly caused by extrinsic factors (e.g. faulty nodes) and the execution time will not be reduced significantly no matter how many speculative tasks are run. In that case, the tasks demanding the most computation progress slower than other tasks and thus are the laggards with small fastness. LTM speeds up their execution by assigning more resources. By balancing the workload within each job, LTM reduces both job execution time and the number of speculative tasks. Assignments of new tasks decrease the number of residual resources while the completion of running tasks increases the number of residual resources. They both trigger the re-allocation of residual resources.

```
Input: Statuses of running tasks on a node
Output: resource assignment to tasks
Algorithm:
TS ←The set of running tasks on a node
TS ←sort TS by fastness in ascending order
LT ←∅   # the set of laggards
for T in TS:
  if fastness(T) < SlowTaskThreshold:
    LT = LT ∪ T
if LT is empty
  fall back to other allocation policies
else
  allocate residual resource to tasks in LT
```

Figure 6.2: Algorithm skeleton of LTM

6.2 The BASE Scheduler

Speculative execution is not a simple matter of running redundant tasks for sufficiently slow tasks. To make it effective, two issues need to be addressed: i) detect slow tasks; ii) choose the tasks to speculate. Hadoop identifies the tasks whose progress rates are one standard deviation lower than the mean of all tasks as slow tasks. Then it chooses the task with the longest remaining execution time to speculate. It does not take into consideration whether speculative tasks will complete before the original tasks. Assume a job has two tasks A and B; task A is 90% done but progresses slowly with rate 1; and task B progresses fast with rate 5. Because task A progresses slow, the master node decides to start a speculative task A' for A which progresses with rate 5. By doing a little math, we can easily figure out that task A will complete earlier than A' although A progresses slowly. The reason is that task A is close to completion when A' is launched. This inefficiency was observed in our tests, where a large portion of speculative tasks were killed before their completion because the original tasks completed earlier. Those speculative tasks were not beneficial at all and their execution resulted in the waste of resources. To overcome the issue, we propose BASE in which speculative tasks are launched only when they are expected to complete earlier than the original tasks. The estimation of the remaining execution time of a running task has been discussed above. We propose a mechanism to estimate the execution time of prospective speculative tasks. It depends upon two factors: 1)

the progress rates of other tasks of the same job; 2) the node where the speculative task will run. The key is to estimate the progress rate which can be directly used to calculate run time. Slow tasks can be identified using the mechanism described in [120]. Given a slow task T of job J and a slave node N_i, following algorithm solves the problem whether a speculative task T' should be launched on N_i for T.

1. If some tasks belonging to J are running or have run on node N_i, the mean of their progress rates is calculated and used as the progress rate of T'.

2. Otherwise, the progress rates of all scheduled tasks of job J are gathered and normalized against the reference baseline. The normalization of progress rates, computed based on hardware processing power (e.g. weighted sum of the capabilities of processors, disks and network interface cards), is needed when nodes are heterogeneous. Then the mean of normalized progress rates is calculated. Because the mean is against the reference baseline, we de-normalize it against the specification of node N_i to compute the expected progress rate of T' on N_i. We assume the scheduling order of tasks is stochastic approximately and thus the mean of scheduled tasks reflects the expectation of real progress rate, which is reasonable given Hadoop scheduling strategy.

3. No matter which of 1) and 2) is applied, the estimated progress rate of T' has been calculated so far. The execution time is $1/progress\ rate$. If it is shorter than the remaining execution time of T, T' is launched on N_i. Otherwise, do not run T' on N_i.

To predict the run time of T' via the mean of progress rates actually is equivalent to the harmonic mean of the run time of scheduled tasks.

6.3 Implementation

Our implementation in Hadoop is optimized for compute-intensive applications and thus processors and cores are the critical resources. Multithreading technique is adopted to explore the parallel processing capability of modern servers. In Hadoop, each task is run in a separate process to isolate its execution environment. Within each task process, one thread is started to process data. Fig. 6.3 shows an example. There are two slave nodes each of which has 5 cores. Each node has 4 slots among which 2 slots are idle. For node A, Slots

A_1 and A_2 are busy; and slots A_3 and A_4 are idle. In Hadoop, each task process only runs one thread even if there are lightly-utilized cores (shown in Fig. 6.3(a)). Resource stealing starts multiple threads within a task process that concurrently process input data (shown in Fig. 6.3(b)). One extra thread is created for both A_1 and A_2, and each of the four threads can be scheduled to an individual core. For each task, *task manager* periodically adjusts the number of threads dynamically based on the latest system status.

Resource stealing and BASE are transparent to end users. Regular MapReduce applications can be run directly without any modification. Additional configuration parameters are added to allow administrators to tune various aspects of our improvements. For example, administrators can enable/disable resource stealing and/or BASE, and change the allocation policy of residual resources.

Although our implementation is based on Hadoop, our algorithms are not specific to Hadoop and can be applied to other systems as well such as Twister and HaLoop that adopt resource "partition"/reservation and speculative execution.

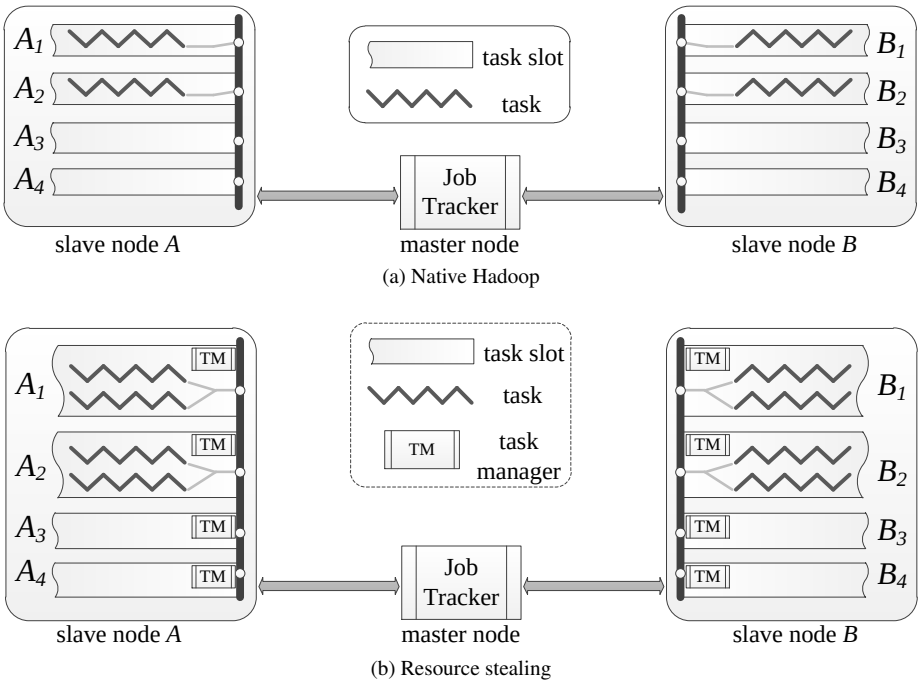


Figure 6.3: Scheduling with native Hadoop and resource stealing

6.4 Experiments

We conducted extensive experiments to evaluate our proposed algorithms. Instead of directly measuring resource utilization (e.g. CPU usage), we measure user-perceivable job execution time which indirectly reflects the improvement or deterioration of resource utilization. Many MapReduce applications have been developed for different purposes. Instead of experimenting with an arbitrary number of common applications one by one, sufficiently representative compute-intensive, IO-intensive and network-intensive data parallel applications (e.g. rep-grep, wordcount, web crawler) are picked and used in our tests below, whose results are applied to not only those tested applications per se but also other applications of the same types. So we believe our findings are applicable to applications of the types under our consideration.

6.4.1 Scheduling of Map-only Jobs

On FutureGrid Hotel cluster, we deployed Hadoop which comprised one master node and twenty slave nodes that were homogeneous in both hardware and software. Each node had 20GB memory and 8 cores one of which was reserved for HDFS and MapReduce daemons. According to the best practice that the number of slots should be between 1x and 2x the number of cores, each node was configured to host 7 map slots and 7 reduce slots. So there were 140 map slots and 140 reduce slots total. Block size of HDFS was set to 128MB.

We ran *grep* without reduce phase to eliminate the impact of shuffling and merging and exactly measure the effectiveness of resource stealing for map-only jobs. A large portion of MapReduce jobs (over 70%) are map-only jobs[73]. In our tests, each map task processed 128MB text data and was tuned to run approximately for 5 minutes by repeating regular expression matching in map operations to simulate the interactive job types in MapReduce [47]. To avoid confusion, we name it *rep-grep*. Please note that grep is IO intensive while rep-grep is compute intensive. Multiple rep-grep jobs were run with the number of map tasks varied. We set the number of map tasks to 35, 70, 105 and 126 which yield system workload 25%, 50%, 75% and 90%.

Rep-grep without BASE We ran rep-grep without BASE and show job execution time in Fig. 6.4(a). Execution time is not significantly influenced by workload for native Hadoop, which means that processing 4.375GB, 8.75GB, 13.125GB, and 15.75GB data takes similar amounts of time. The reason is resource usage

is proportional to the number of tasks and residual resources are not utilized at all. Resource stealing shortens run time by 64%, 32%, 13%, and 6% respectively for policy Even. The lower the workload is, the more resource stealing outperforms native Hadoop. So the performance benefit of resource stealing is negatively related to system workload, which matches our expectation well. We also calculated the average processing time per GB data by dividing job execution time by data size. Increasing workload can drastically improve the efficiency for native Hadoop, while it approximately keeps invariant for resource stealing. Different allocation policies exhibit different performance. Overall, STLM and LTLM perform the worst and LTM performs well for all tests. It implies that it is inefficient to blindly allocate residual resources evenly or simply enforce FIFO order. When the workload gets relative high (e.g. 75%, 90%), the performance difference becomes smaller.

In our setup, all nodes were on the same rack and blocks were randomly placed on nodes by HDFS with its default block placement strategy. Data locality aware scheduling in Hadoop co-locates compute and data with best efforts. As a result, map tasks were evenly distributed among all slave nodes approximately so that each node ran a similar number of tasks. This is beneficial to resource stealing because its gain is not substantial if the resources of a node are fully loaded already.

Rep-grep with BASE We ran the same tests as above except BASE was enabled and present results in Fig. 6.4(b). The plot has similar characteristics to Fig. 6.4(a) in that native Hadoop performs the worst and the performance superiority of resource stealing decreases with the increased sytem workload. By comparing 6.4(b) and 6.4(a), we observe that BASE slightly shortens execution time and the improvement is increased as system workload is also increased.

For the cases where BASE is disabled and enabled, we counted the number of non-beneficial speculative tasks and computed the difference shown in Fig. 6.4(c). BASE drastically eliminates the launches of non-beneficial speculative tasks. For workload 75% and 90%, almost all unneeded speculative tasks are removed.

We conclude that BASE reduces the number of non-beneficial speculative tasks significantly without sacrificing run time. Because a fewer number of speculative tasks are launched, the saved resources can be allocated to regular tasks to speed up their execution. It also indicates that the estimation of execution time is approximately accurate so that BASE rarely removes the runs of beneficial speculative tasks.

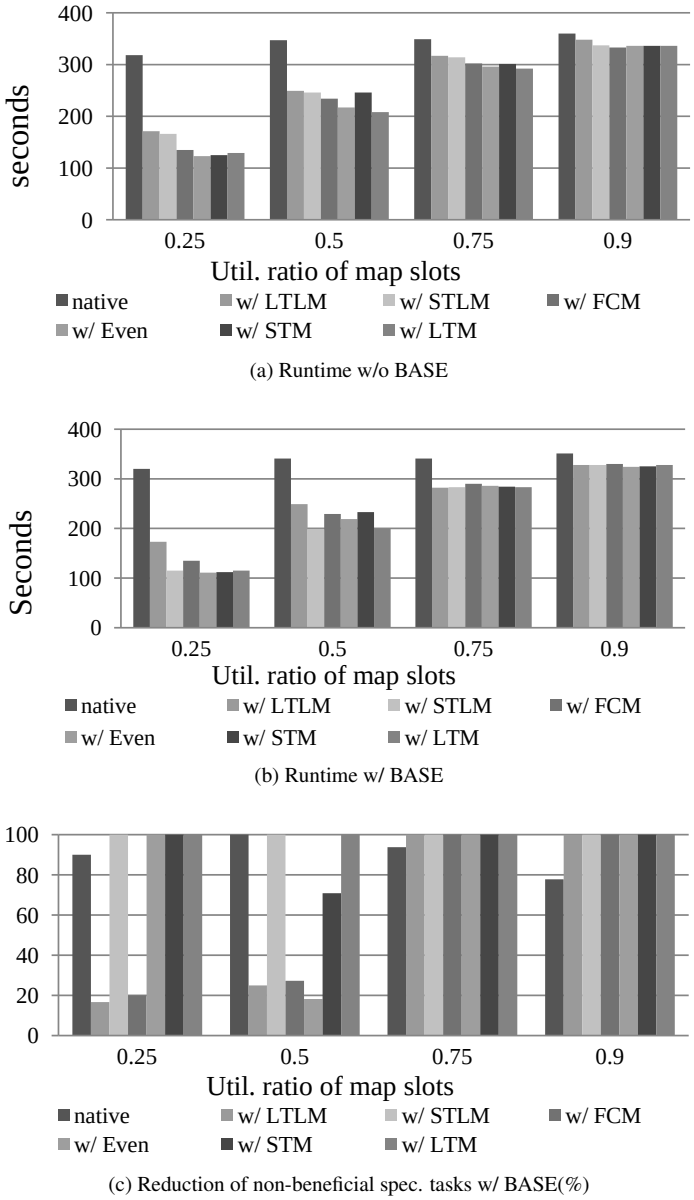


Figure 6.4: Run map-only rep-grep in a homogeneous environment

6.4.2 Scheduling of Map-only Jobs with Straggler Nodes

In this experiment, background load is generated to slow down some nodes and simulate stragglers. We wrote a load generator that can generate user-specified load of computation, network and disk IO. We ran two CPU-hogging threads per core which resulted in nearly 100% core utilization, and one IO-intensive thread

reading/writing data continuously from/to disks. The background load significantly slowed down the nodes without rendering them thoroughly unresponsive. We ran rep-grep jobs that utilize 75% of all map slots and thus 8.75GB data was processed total in each run.

Firstly, two slave nodes were slowed down. Job execution time is shown in Fig. 6.5(a). Again resource stealing improves performance over native Hadoop significantly no matter which resource allocation policy is used. LTM performs well stably for the cases with and without BASE. Fig. 6.5(b) shows BASE can save runs of nearly all unnecessary speculative tasks, which implies the estimation of execution time is accurate when only a small number of nodes are stragglers.

Secondly, four slave nodes were slowed down. Fig. 6.5(c) shows execution time. The jobs ran longer compared with the previous test because more map tasks were slowed down. Resource stealing is still effective to speed up job execution. The performance disparity of different resource allocation policies becomes marginal and they perform equally well approximately. Fig. 6.5(d) shows BASE can eliminate 20% - 50% of non-beneficial speculative tasks. Compared with the previous case, BASE becomes less effective. It indicates our estimation of execution time gets inaccurate as more straggler nodes incur larger variation of task execution. In addition, resource stealing aggravates the situation because of the dynamic nature of the (re-)allocation of residual resources.

6.4.3 Scheduling of Reduce-mostly Jobs

In this test, we ran reduce-mostly jobs and used modified *grep* (not *rep-grep*) as the test application. Operations in the reduce phase of grep were run repeatedly to make reduce phase dominate overall execution. Each job comprised 10 reduce tasks and 70 map tasks each of which processed 128MB data, and ran for 5 minutes approximately. This setup matches the fact that most MapReduce jobs tend to have significantly lesser reduce tasks than map tasks. For resource stealing, only policy Even is compared below because it is simple and performs among the best based on the results above.

Job execution time and the number of non-beneficial speculative tasks are shown in Fig. 6.6(a) and Fig. 6.6(b) respectively. BASE reduces job execution time marginally, but reduces the number of non-beneficial

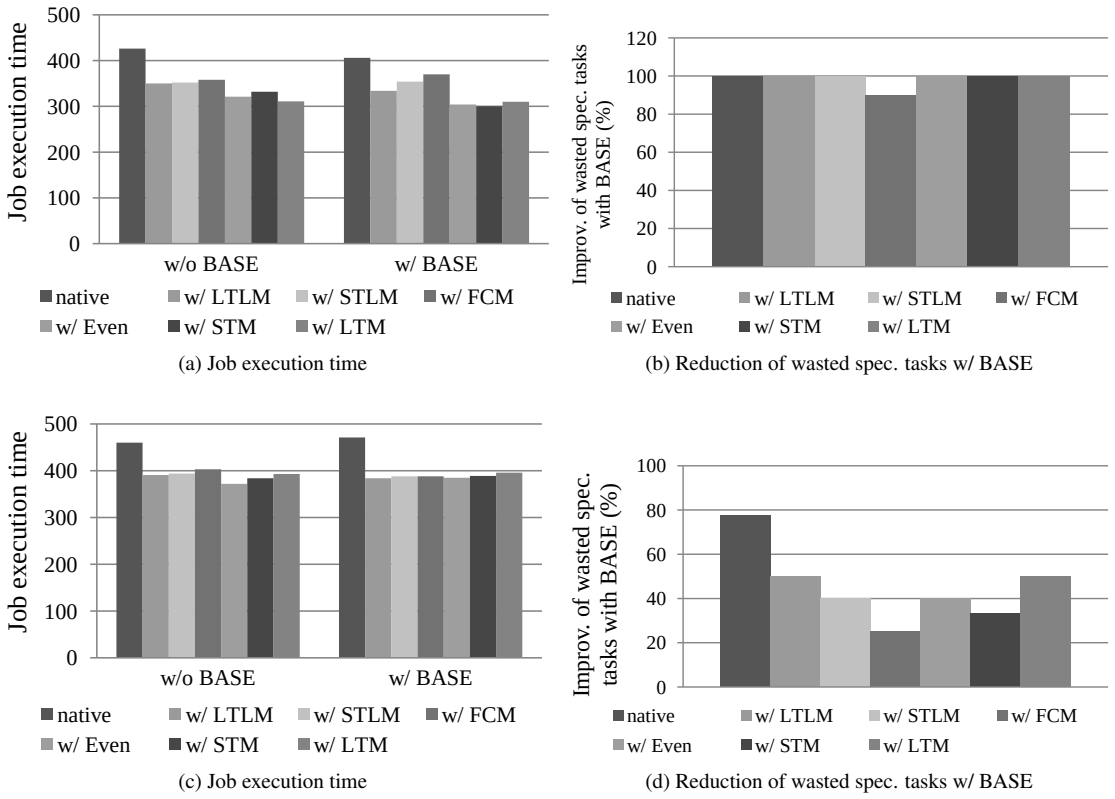
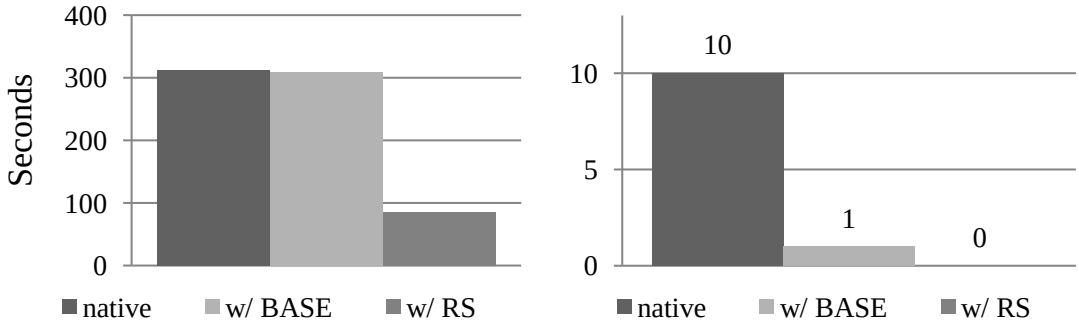


Figure 6.5: Run map-only rep-grep with straggler nodes. There are two straggler nodes for (a) and (b), and four straggler nodes for (c) and (d)

speculative tasks by 90% (from 10 to 1) compard with native Hadoop. Because of the drastic reduction of resource waste, more useful tasks can be run concurrently and thus the efficiency of resource usage is improved. This demonstrates the effectiveness of BASE. Fig. 6.6(b) shows resource stealing thoroughly eliminated non-beneficial speculative tasks, and Fig. 6.6(a) shows resource stealing substantially shortens job execution time by 70% - 80%. There are many more nodes than reduce tasks which are well spread out so that each node runs one reduce task at most on average. For each reduce task, resource stealing creates 6 new reduce tasks (remember the number of reduce slots is 7 on each node) to run in parallel, which should yield 7x speedup optimally. In reality, we only got 4x–5x speedup because of additional overhead. Reduce threads compete for the same input stream and only one thread can read from the stream at any time. To alleviate the contention, in our implementation each thread locks the input stream, copies next (key, values) tuple to its local buffer, unlocks the input stream and processes the data in local buffer without interfering with other threads. But this approach incurs extra memory copies. In addition, reduce threads belonging to the same

task contend for the same output stream as well. To investigate advanced mechanisms to mitigate contention further is among future work.



(a) Job execution time

(b) Num. of non-beneficial spec. tasks

Figure 6.6: Reduce-mostly modified grep

6.4.4 Experiments with Other Workload

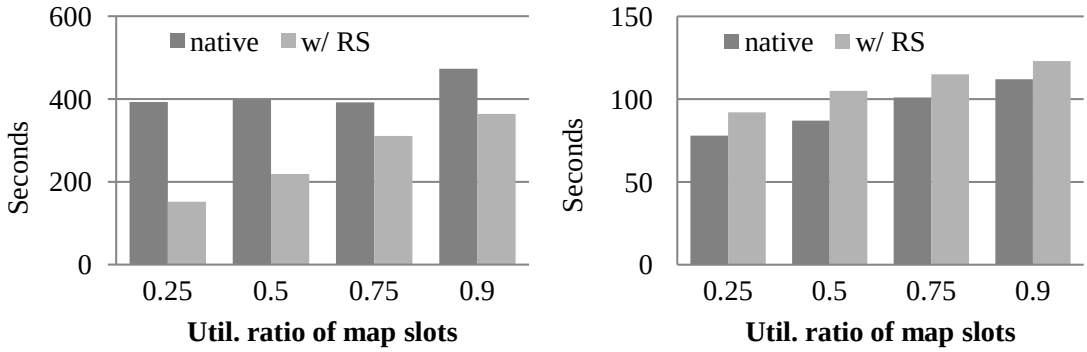
Besides compute-intensive applications. We also ran jobs of other types to comprehensively evaluate our approaches.

Network-Intensive Workload: We wrote a distributed web crawler *mr-wc*. Its input is a set of URLs of the webpages to download. Mr-wc does not have reduce phase; and its map tasks download web pages and save them into HDFS. Network is the most critical resource for mr-wc. Lemur project published a data set of unique URLs [2]. We use a small portion of it as the input of mr-wc. The same testbed as above was used. In our tests, each map task downloaded 400 web pages and the number of map tasks was set to 35, 70, 105 and 126 for different runs, so the system workload was 25%, 50%, 75% and 90% respectively. Fig. 6.7(a) shows the execution time. For native Hadoop, the execution time of mr-wc is not significantly impacted by the workload, which implies spare resources cannot be utilized. In contrast, resource stealing expands the usable resources of running tasks by creating more threads to concurrently download webpages. Execution time is shortened drastically by 61%, 45%, 21% and 23% respectively.

For above tests, speculative execution was disabled because our additional tests showed it deteriorates

performance mostly. The efficiency of webpage crawling depends heavily on the response time of the servers where webpages are hosted, which ranges from milliseconds to seconds. Under this circumstance, running speculative tasks is not helpful because the efficiency variation of tasks is not caused by the system itself.

IO-Intensive Workload: *Wordcount*, which counts the number of word occurrences, is a typical IO-intensive application and used in this set of tests where each map task processed 128MB text and the number of map tasks was varied. Fig. 6.7(b) shows the result. As the number of map tasks increases, job execution time increases as well and the processing throughput (the amount of processed data per unit of time) is improved. Resource stealing slightly degrades rather than improves performance, which is caused by Hadoop implementation. In map tasks, each map operation processes one line of text and is invoked repeatedly. Although resource stealing enables Hadoop to start multiple threads to run map operations in parallel, these threads share the same underlying input reader and output writer (to comply with Hadoop design). This incurs substantial contention among threads because the computation time of each map operation is short and synchronization is the performance barrier. As a result, the overhead outweighs the benefit of higher concurrency brought by resource stealing. This inefficiency is not intrinsic and pertains to Hadoop design more than our algorithm. Theoretically moderate increase of I/O parallelism can exploit the interleaving of computation and data I/O and thus improve I/O throughput.



(a) mr-wc (b) wordcount

Figure 6.7: Experiment with other workload

6.5 Summary

The goal of our work is to improve resource utilization in MapReduce. We present resource stealing to dynamically re-allocate idle resources to running tasks with the promise that they will be handed back whenever they are required by newly assigned tasks. It can be applied in conjunction with existing job schedulers smoothly because of its transparency to central task scheduling. In addition, we have analyzed the mechanism adopted by Hadoop to trigger speculative execution, discussed its inefficiency and proposed Benefit Aware Speculative Execution (BASE) which starts speculative tasks based on the estimated benefit. Our conducted experiments demonstrate their effectiveness. Resource stealing yields dramatic performance improvement for compute-intensive and network-intensive applications and BASE effectively eliminates a large portion of unnecessary runs of speculative tasks. For IO-intensive applications, we observed slight performance degradation caused by intensive contention in Hadoop framework. In future, we will investigate lock-free data structures to make resource stealing benefit IO-intensive applications as well in Hadoop.

7

Hierarchical MapReduce and Hybrid MapReduce

Since its proposal, MapReduce has drawn the attention of many communities. It has been widely adopted by run data parallel applications. However, MapReduce is not a panacea to all parallel applications. In this chapter, we propose two enhancements to MapReduce model: Hierarchical MapReduce (HMR) and Hybrid MapReduce (HyMR). HMR is an extended MapReduce programming model and a tiered framework enabling MapReduce jobs to run in cross-domain environments. HyMR is a workflow management system specifically designed to integrate regular MapReduce and iterative MapReduce in a way that efficiency and fault tolerance are well balanced.

7.1 Hierarchical MapReduce (HMR)

In academic institutes, traditional HPC clusters have been deployed. It is more cost effective to reuse those resources for Hadoop deployment than to rebuild from scratch. Sometimes a user has access to multiple clusters under different administrative domains. For example, we can access FutureGrid clusters, TeraGrid clusters and some Indiana University (IU) campus clusters. Ideally we want to utilize all the accessible resources to run complicated jobs. Each cluster has a couple of login nodes (or head nodes) that are publicly accessible. After a user logins, he/she can do various job-related tasks, including job submission, job status query and job cancellation. The internal compute nodes however, usually cannot be directly accessed from

outside for security purpose. All requests must be initiated from login nodes. This constraint prevents us from building a single MapReduce cluster across these physical clusters because the master node needs to be able to communicate with all slave nodes directly.

To resolve the issue, we propose Hierarchical MapReduce (HMR) which adopts a multi-tier approach. For a specific tier, the results collected from the tiers immediately under it are processed and the results are sent to the tier above it. Although theoretically the number of tiers can be arbitrary, in this thesis we only consider two-tier architecture which is sufficient to most scenarios. Having more tiers will increase the implementation complexity and incurs higher overhead. In this sense, we prefer "fat" trees with smaller depth to "tall" trees with a fewer number of clusters at each level. Our current design is optimized for CPU-intensive applications where computation time is dominant and data movement cost is negligible.

7.1.1 Programming Model

Firstly, we extended the vanilla MapReduce model to Map-Reduce-Global Reduce mode where each job is expressed with three operations: map, reduce and global reduce. The term "global reduce" is used intentionally to distinguish it from the regular reduce. However, the implementations of local reduce and global reduce can be identical. Like MapReduce, each map operation takes a key/value pair as input and produces a list of key/value pairs; and each reduce operation takes as input a list of intermediate key/value pairs having the same key and produces key/value pairs. Global reduce operation collects the partial results from local reduce operations, processes them and generates the final results. Existing MapReduce applications such as wordcount and sorting can be trivially converted to our model if their reduce operations are associative.

7.1.2 System Design

Fig. 7.1 shows the overall architecture of our proposed HMR framework. The top tier is the global controller which has four main components: Task Scheduler, Data Transferer, Workload Collector, and Global Reducer. The bottom tier comprises a set of local MapReduce clusters each of which runs a Workload Reporter and a Job Manager.

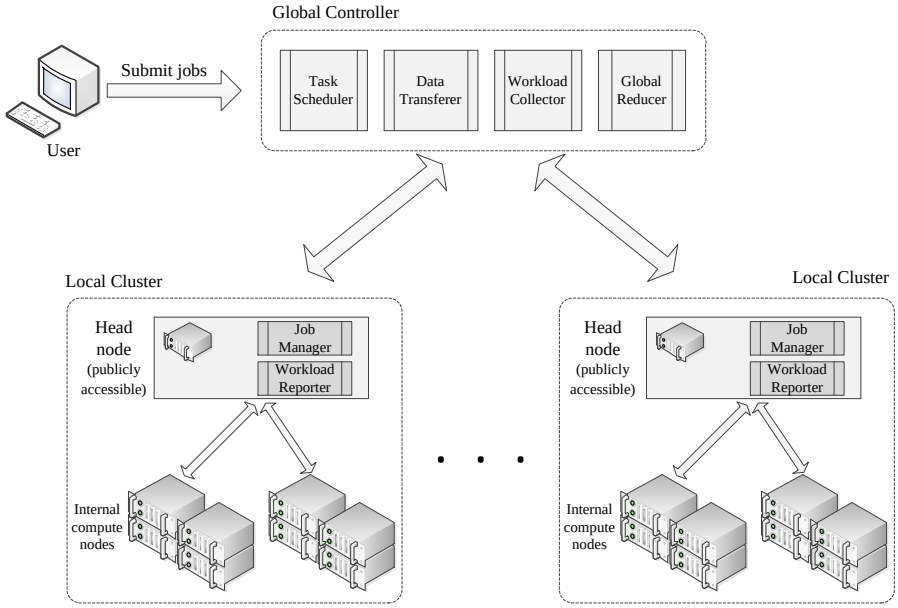


Figure 7.1: Hierarchical MapReduce (HMR) architecture

The functionalities of the main components of the global controller are elaborated below:

- **Task Scheduler** accepts jobs submitted by users, splits and dispatches them to local clusters. It can utilize the information maintained by other components such as Workload Collector to optimize scheduling.
- **Data Transferer** handles data transfers to and from local clusters on demand.
- **Workload Collector** collects the workload information of all bottom-tier clusters.
- **Global reducer** applies the user-provided global reduce implementation to the aggregated data from local clusters and generates the final output.

The functionalities of the main components of the local clusters are elaborated below:

- **Workload Reporter** reports the workload information of a cluster to the global controller periodically. Currently, the collected workload information includes the number of running tasks, the ratio of idle slots, etc.

- **Job Manager** accepts the jobs assigned by the global controller and runs them in local MapReduce cluster.

After a new job is submitted to the global controller by a user, HMR automatically parallelizes the execution. The data flow is:

1. After a new job is received, the task scheduler splits the input into a number of partitions based on the number of local clusters. If the data has been uploaded beforehead, this step can be skipped.
2. Each data partition along with the user provided job jars and configuration files is sent to the corresponding local cluster through data transferer.
3. When the data transfer completes, the task scheduler notifies the local job manager.
4. The local job manager starts up a new MapReduce job to process the data received in step 2.
5. After a local MapReduce job completes on a cluster, its output is sent back to the global controller if required by the application.
6. The global controller invokes the global reducer to apply the final reduce operation, after all partial results have been received from local clusters.

Since there is only a single global controller, it may become performance bottleneck. Data transfer is expensive and thus we recommend that users use the global controller to stage data only when the amount is small. We want to avoid the scenarios where data transfer cost is significant and dominates the overall execution. If the size of input data is large, we recommend that users upload it into the clusters beforehand manually or by using a script. One potential solution we have not fully explored is to use multiple global controllers, which is demonstrated in Fig. 7.2. A gateway node is added upfront which runs a load balancer dispatching received user requests to global controllers according to their load. As a result, each global controller only handles a portion of all the jobs. One drawback is the state of all local clusters needs to be collected and maintained by all global controllers.

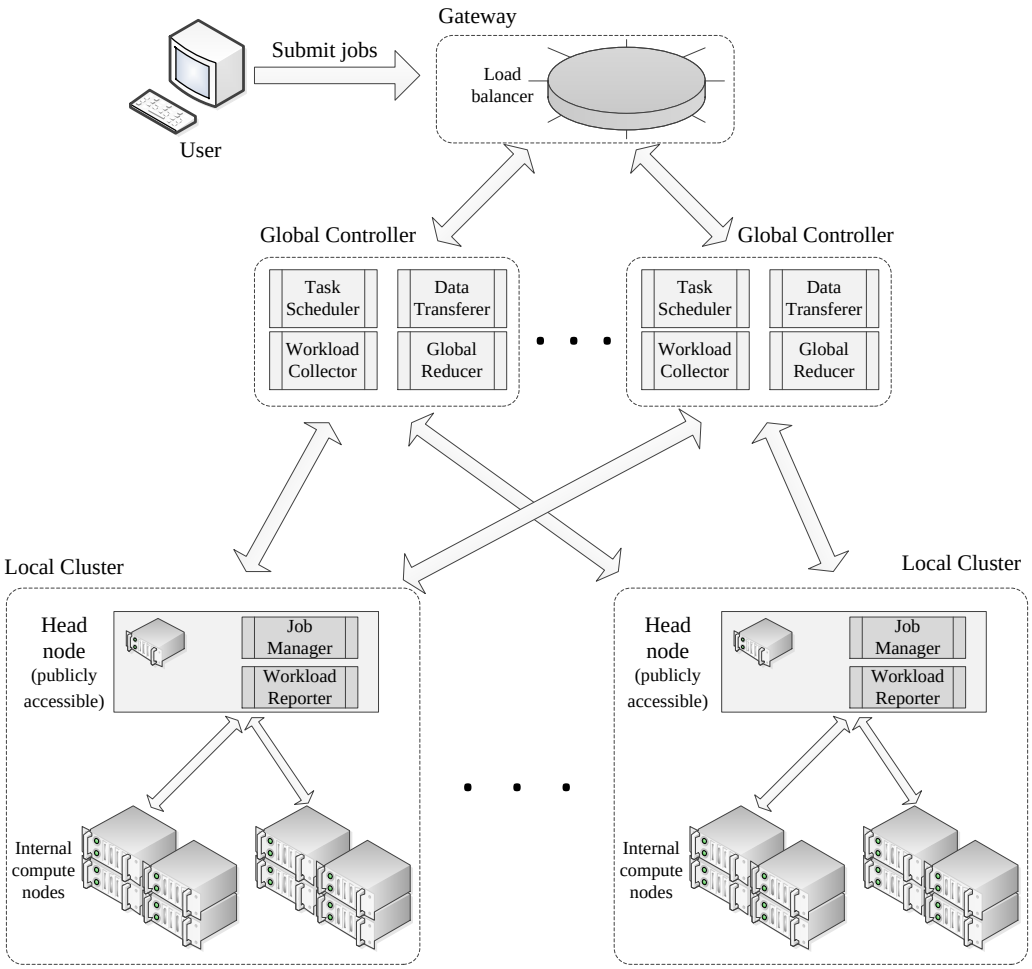


Figure 7.2: Hierarchical MapReduce (HMR) architecture with multiple global controllers

7.1.3 Data Partition and Task Scheduling

One challenge of HMR is how to partition the workload among local clusters so that their load is well balanced. No matter input data are staged through the global controller or pre-uploaded beforehand, the task scheduler takes data locality into consideration when scheduling jobs. Here we mainly discuss the case where data are distributed by the global controller because that is easier and less error-prone. After data are received, the global controller counts the number of records using the user-implemented $InputFormat$ and $RecordReader$, and splits them into different partitions that are staged to local clusters subsequently.

As we discussed, we are mainly focused on compute intensive applications and assume all map tasks of a parallel job undertake similar amounts of work (i.e. their workload is approximately identical). The task homogeneity assumption is also made by Hadoop. Our use of AutoDock below to evaluate HMR exhibits this behavior. Table 7.1 summarizes the symbols we will use to express data partition policies. For a job, the portion of data processed by cluster i is denoted by W_i, which should match the idle computation power of cluster C_i. W_i can be calculated with (7.1), from which we can see the absolute magnitude of θ_i does not matter and the relative magnitude impacts W_i. Given the number of records to process, the portions assigned to clusters are calculated with (7.2).

Symbol	**Description**
N	the number of clusters
C_i	cluster i
NC_i	the total number of processor cores in cluster C_i
M^i_{max}	the total number of map slots in cluster C_i
M^i_{avail}	the number of available map slots in cluster C_i
M^i_{run}	the number of occupied map slots in cluster C_i (i.e. $M^i_{max} - M^i_{avail}$)
θ_i	the available compute power of cluster C_i (proportional to M_{avail})
J_i	a job submitted by a user
JM_i	the total number of map operations to run (i.e. the number of records in input data)
JM^j_i	the work assigned to cluster C_j for job J_i

Table 7.1: Symbols used in data partition formulation

$$W_i = \frac{\theta_i}{\sum_{i=1}^{N} \theta_i} \tag{7.1}$$

$$JM^j_i = W_i \cdot JM_i \tag{7.2}$$

7.1.4 AutoDock

We apply the MapReduce paradigm to running multiple AutoDock instances using the HMR framework to prove the feasibility of our approach. We take the outputs of AutoGrid (one tool in the AutoDock suite) as input to the AutoDock. The key/value pairs of the input of the Map tasks are ligand names and the location

of ligand files respectively. We designed a simple input file format for AutoDock MapReduce jobs. Each input record, which contains 7 fields shown in Table 7.2, corresponds to a map task. For our AutoDock MapReduce, the Map, Reduce, and Global Reduce functions are implemented as follows:

Map Each map task takes a ligand to run the AutoDock binary executable against a shared receptor, and then runs a Python script *summarize_result4.py* to generate the lowest energy result. All intermedate output shares a constant key.

Reduce The reduce task takes all the values generated by map tasks, sorts them by energy in ascending order, and writes the sorted results to a file using a local reducer intermediate key.

Global Reduce The Global Reduce finally takes all the values of the local reducer intermediate key, sorts and combines them into a single file by energy from low to high.

Field	Description
ligand_name	Name of the ligand
autodock_exe	Path to AutoDock executable
input_files	Input files of AutoDock
output_dir	Output directory of AutoDock
autodock_parameters	AutoDock parameters
summarize_exe	Path to summarize script
summarize_parameters	Summarize script parameters

Table 7.2: AutoDock MapReduce input fields and descriptions

7.1.5 Experiments

We build a prototype system for our proposed HMR on top of Hadoop. The system is written in Java and Shell scripts. We use *ssh* and *scp* scripts for data stage-in and stage-out. On the local clusters side, the workload reporter is a component that exposes the load information of local clusters. The load information can be used by global scheduler to make task scheduling more efficient. Our original design was to make it a separate program without touching Hadoop source code. Unfortunately, Hadoop does not expose the load

information we need to external applications, and thus we had to modify Hadoop code to add an additional daemon that collects load data by calling Hadoop internal Java APIs.

In our evaluation, we use several clusters including the IU Quarry cluster and two clusters in FutureGrid. IU Quarry is a classic HPC cluster which has several login nodes that are publicly accessible from outside. Several distributed file systems (e.g. Lustre, GPFS) are mounted to each compute node for data sharing. FutureGrid partitions the physical resources into several parts, each of which is dedicated to a specific testbed such as Eucalyptus, Nimbus, and HPC.

To deploy Hadoop to traditional HPC clusters, we first use the built-in batch scheduler to allocate nodes. To balance maintainability and performance, we install the Hadoop program in shared directory while store data in local directory, because the Hadoop program (Java jar files, etc.) is loaded only once by Hadoop daemons whereas the HDFS data are accessed multiple times.

We use three clusters for evaluations: IU Quarry, FutureGrid Hotel and FutureGrid Alamo. We applied for 21 nodes in each cluster among which one is a dedicated master node and others are slave nodes. They all run Linux 2.6.18 SMP. Each node in these clusters has an 8-core CPU. The specifications of these cluster nodes are listed in Table 7.3.

Cluster	**CPU**	**L2 cache size**	**Amount of memory**
Hotel	Intel Xeon 2.93GHz	8192KB	24GB
Alamo	Intel Xeon 2.67GHz	8192KB	12GB
Quarry	Intel Xeon 2.33GHz	6144KB	16GB

Table 7.3: Cluster node specifications

In our first experiment, we ran AutoDock to process 2000 ligands and 1 receptor in one cluster. One of the most important configuration parameters is *ga_num_evals* - the number of docking evaluations. The larger its value is, the higher the probability becomes that better results are obtained. Based on prior experiences, the *ga_num_evals* is typically set from 2500000 to 5000000. We configured it to 2500000 in our experiments. Fig. 7.3 plots the number of running map tasks during the job execution. The cluster had 20 slave nodes, so the maximum number of running map tasks at any moment is 20 * 8 = 160. From the plot, we can see that the number of running map tasks quickly grew to 160 in the beginning and stayed approximately constant for a long time. Towards the end of job execution, it dropped to a small value quickly (roughly 0 - 5). Notice

there is a tail near the end, indicating that node usage ratio was low and all tasks were not perfectly balanced. At this moment, if new MapReduce tasks were submitted, the available mappers would be utilized by those new tasks. Fig. 7.4 shows the number of map tasks executed on each slave node. We observe that the load balancing across nodes was good and each node ran 100 map tasks or so.

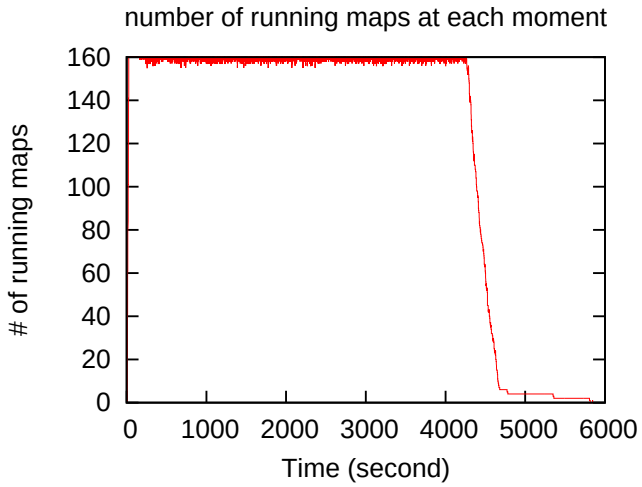


Figure 7.3: Number of running map tasks for an Autodock MapReduce instance

7.1.5.1 Micro-benchmark each cluster

In this test, global controller is not involved. We intend to figure out how each local Hadoop cluster performs with different sizes of input data. We ran AutoDock in the Hadoop to process 100, 500, 1000, 1500 and 2000 ligand/receptor pairs in each clusters. See Table 7.4 and Fig. 7.5 for results. Firstly the execution time is approximately linear with the number of processed ligand/receptor pairs. Secondly, the total execution time of the jobs running on the Quarry cluster is approximately 30% - 50% slower than running on Alamo and Hotel. The main reason is that the nodes in Quarry have less powerful CPUs than that in Alamo and Hotel.

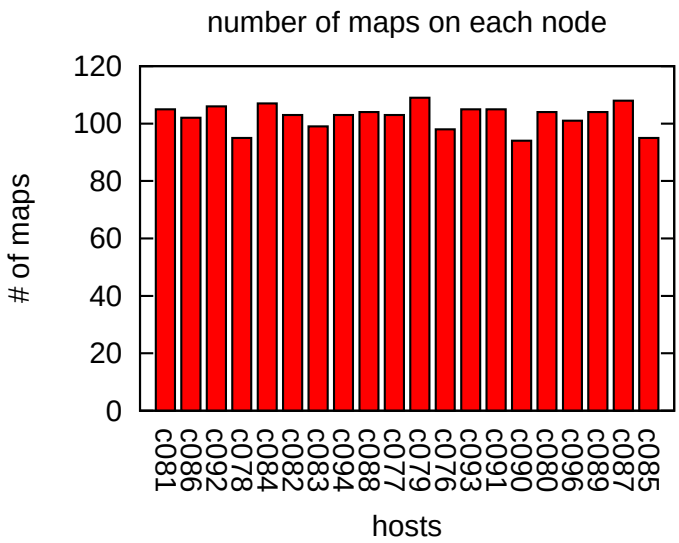


Figure 7.4: Number of map tasks executed on each node

Number of Map Tasks Per Cluster	Execution Time on Three Clusters		
	Hotel (seconds)	Alamo (seconds)	Quarry (seconds)
100	1004	821	1179
500	1763	1771	2529
1000	2986	2962	4370
1500	4304	4251	6344
2000	5942	5849	8778

Table 7.4: MapReduce execution time on different clusters with varied input size

7.1.5.2 Even data distribution:

In this test, we use all three clusters to run AutoDock to process 6000 ligand/receptor pairs. The input data were evenly distributed across the clusters. Thus, the weight of map tasks distribution on each cluster is W_i=1/3. We equally partition the dataset (apart from shared dataset) into 3 parts, send the data together with the jar executable and job configuration file to local clusters for execution in parallel. After the local MapReduce execution, the output files will be staged back to the global controller for the final global reduce. All resources are used by a single HMR job.

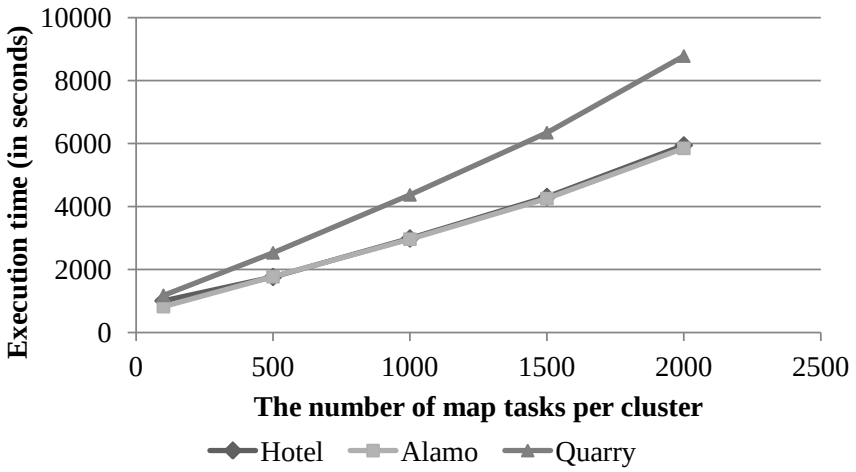


Figure 7.5: Execution time of tasks in different clusters

A receptor is shared by all ligands and described as a set of approximately 20 gridmap files totaling 35MB in size, and the 6000 ligands are stored in 6000 separate directories, each of which is approximately 5-6 KB large. In addition, the executable jar and job configuration file together has a total of 300KB in size. For each cluster, the global controller creates a 14MB tarball containing 1 receptor file set, 2000 ligands directories, the executable jar, and job configuration files, all compressed, and transfers it to the destination cluster, where the tarball is decompressed. We call this global-to-local procedure "data stage-in". Similarly, when the local MapReduce jobs finish, the output files together with control files (typically 300-500KB in size) are compressed into a tarball and transferred back to the global controller. We call this local-to-global procedure "data stage-out". The data stage-in procedure takes 13.88 to 17.3 seconds to finish, while the data stage-out procedure takes 2.28 to 2.52 seconds to finish. The Alamo cluster takes a little longer to transfer the data but the difference is insignificant compared to the relatively long duration of local MapReduce executions.

The time it takes to run 2000 map tasks on each of the local MapReduce clusters varies due to the different specification of the clusters. The local MapReduce execution makespan, including data movement costs (both data stage-in and stage-out) is shown in Figure 7.6. Similar amounts of time was taken to run jobs in Hotel and Alamo, while the job execution in the Quarry cluster took approximately 3000 more seconds (about 50% more than that of Hotel and Alamo). The Global Reduce task, invoked only after all the local results are transferred to the global controller, took only 16 seconds to finish. Thus, the relatively poor performance on

Quarry becomes the bottleneck on the current job scheduling.

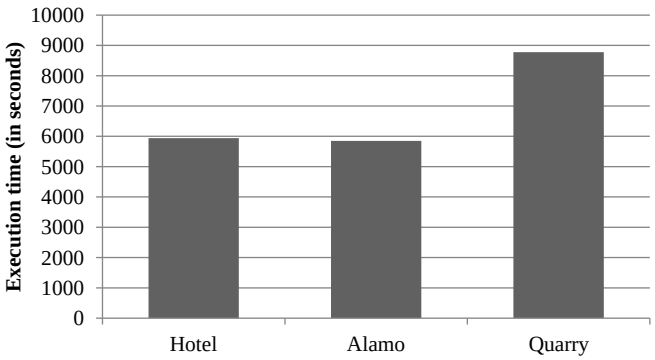


Figure 7.6: Execution time of tasks in different clusters (even data distribution)

7.1.5.3 Resource capability aware data distribution

In even data distribution, we observed substantial execution time skew among jobs in different clusters although all clusters have the same number of compute nodes and process the same amount of data. In this test, we partition the data in a resource capability aware manner to achieve better load balancing. Among the three clusters, Quarry is less powerful than Alamo and Hotel. The specifications of the cores on Quarry, Alamo and Hotel are shown in Table 7.3. The processing time is approximately inversely proportional to CPU frequency. So we hypothesize that the difference in processing time is mainly due to the different core frequencies. Therefore it is not enough to merely factor in the number of cores for load balancing, and the computation capability of each core is also important. We refine our scheduling policy to add CPU frequency as a factor to set θ_i. Here we set θ_1=2.93 for Hotel, θ_2=2.67 for Alamo, and θ_3=2 for Quarry. Thus, the weights are W_1=0.3860, W_2=0.3505, and W_3=0.2635 for Hotel, Alamo, and Quarry respectively. The dataset is also partitioned according to the new weight.

With this resource capability aware data partition, we expect the load is better balanced among clusters. The local MapReduce execution time including data movement costs (both data stage-in and stage-out) and the number of map tasks are shown in Fig. 7.7. The execution time of local jobs was similar for all three clusters, while they ran different numbers of map tasks. We observe that our refined data partitioning improves

performance by balancing workload among clusters. In the final stage, the global reducer combines partial results from local clusters and sorts them. The average time taken to merge local results is 16 seconds.

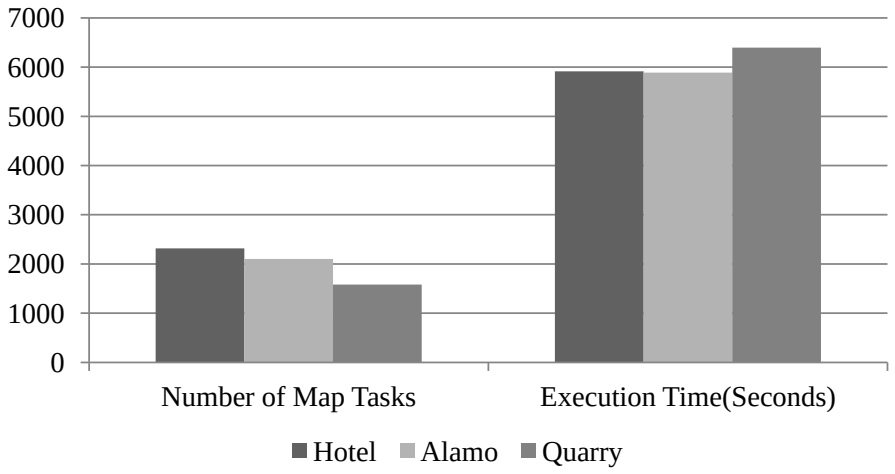


Figure 7.7: The number of tasks and execution time in different clusters

7.1.6 Summary

HMR enhances MapReduce model by adding a new global reduce operation, which allows the system to be built in a hierarchical manner. HMR circumvents the constraint of grid resource accessing and unifies multiple physical clusters into a single system. Users only need to submit a HMR job and the runtime automatically parallelizes the processing and makes full use of all cross-domain resources to maximize performance. Our evaluation with AutoDock, a compute intensive biology applications, demonstrates that our prototype HMR implementation is effective and efficient to shorten job execution time. Currently, computation capability is the only factor we considered for workload partition, which may not be inefficient for other types of applications.

7.2 Hybrid MapReduce (HyMR)

Hadoop is a widely used implementation of MapReduce. Hadoop supports fault tolerance in both MapReduce execution engine and HDFS. For each data block, HDFS maintains multiple replicas to improve data availability. Hadoop MapReduce uses speculative execution to minimize the impact of faulty, failed and over-loaded nodes.

Although many domain-specific problems can be artificially converted and expressed in MapReduce paradigm, the performance may deteriorate. In section 2.3.2, we discussed why Hadoop does not perform well for iterative applications. Twister [50] is one of the earliest framework designed specifically for iterative applications. Twister adopts publish/subscribe messaging for communication and data transfer. Intermediate data across iterations are cached in memory and processed by long-running map tasks. Despite its performance advantage, Twister has some drawbacks. Firstly, Twister lacks the support of distributed file systems. Usually input data are stored in a shared file system (e.g. NFS). The user needs to manually partition the file and distribute data to the local storage of compute nodes. Twister has limited support of fault tolerance. In addition, Twister adopts static scheduling and the number of map tasks (equal to the number of partitions) must be the same as that of processor cores.

For complicated applications, to express all processing in a single job is not feasible and usually they are logically split into a set of nonindependent jobs. It is likely that an application comprises both regular MapReduce jobs and iterative MapReduce jobs. We propose Hybrid MapReduce (HyMR) which is a workflow management system combining the best of both regular MapReduce and iterative MapReduce.

Fig. 7.8 shows the architecture of our proposed HyMR built upon Hadoop and Twister. Workflows are expressed in Extensible Markup Language (XML) or Java properties format. The user can also provide runtime configuration files which indicate how Hadoop and Twister should be configured. This benefits those workflows which require a different configuration than default to maximize performance. The progress of workflow execution is recorded in a XML file and accessible to users. HyMR has three main components: workflow instance controller, job controller and runtime controller.

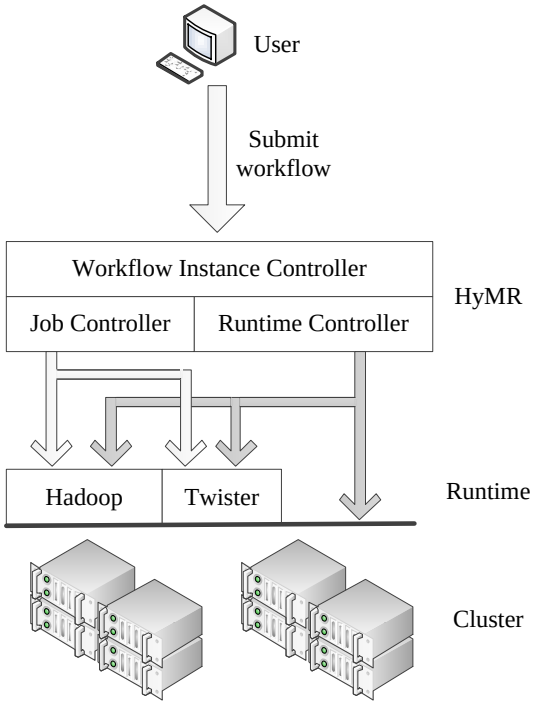


Figure 7.8: Hybrid MapReduce (HyMR) architecture

Workflow instance controller manages workflow instances which are running copies of workflow definitions. Based on the user-provided workflow description file, a DAG is generated which encodes jobs and their dependency relationship. Workflow instance controller invokes the *runtime controller* to starts runtimes, and controls the execution of jobs. If a job fails, the workflow instance controller will re-run it. If the maximal number of retries is reached, the workflow instance controller gives up and reports the failure. If a runtime fails (e.g. Hadoop daemon crashes), the workflow instance controller restarts the runtime and re-runs all failed jobs.

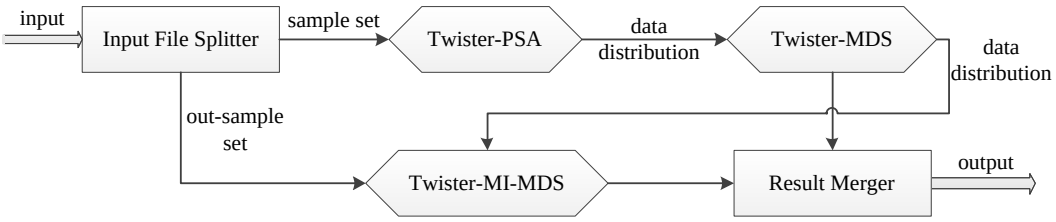
Job controller manages the execution of a single job. A job is automatically started when all its prerequisite jobs have completed. A job can be sequential or parallel. The job controller monitors job execution and can recover failed tasks. For Hadoop, fault tolerance is natively built in. For Twister, a fault detector keeps track of the state of Twister daemons. Current job is killed if any daemon failure is detected. Once the execution of a job completes or fails, the job controller notifies the workflow instance controller.

Runtime controller manages Hadoop and Twister. After physical nodes are allocated through TORQUE

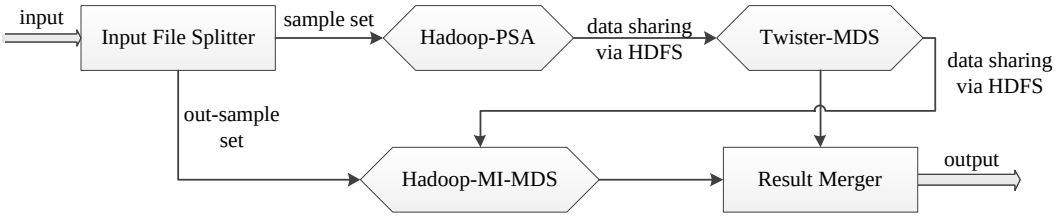
[14], the runtime controller deploys and starts Hadoop and Twister on them. They are shared by all jobs of a workflow, and termed *persistent runtimes*. After a workflow instance completes, the runtime controller stops Hadoop and Twister. Compared with the approach that runtimes are started and stopped for each parallel job, persistent runtimes incur lower overhead.

7.2.1 Workflows

We have implemented two HyMR workflows for bioinformatics data visualization, termed Twister-pipeline and Hybrid-pipeline shown in Fig. 7.9. After input data are received, they are split into two sub-sets: sample set and out-sample set. The sample set is firstly processed by a Pairwise Sequence Alignment (PSA) algorithm whose output is the input of a Multidimensional Scaling (MDS) algorithm. MDS interpolation computes the mapping result of out-sample set by interpolating the mapping of sample set produced by MDS. Some parallel algorithms have been implemented by us for both Hadoop and Twister. For Twister-pipeline, all parallel jobs run on Twister and data partitioning and staging need to be explicitly handled. For Hybrid-pipeline, iterative algorithms such as MDS is implemented in Twister while other parallel jobs are implemented in Hadoop; and data sharing is implicitly handled by HDFS. The detailed discussion of each application is presented below.



(a) Twister-pipeline



(b) Hybrid-pipeline

Figure 7.9: HyMR workflows

7.2.1.1 Pairwise sequence alignment

Sequence alignment identifies similar regions among the sequences of DNA, RNA and protein that may be a consequence of functional, structural or evolutionary relationships. Pairwise sequence alignment does all-pair alignment over a set of sequences. Its result is naturally expressed as a all-pair dissimilarity matrix. Among proposed PSA algorithms, Smith Waterman Gotoh (SWG) algorithm is used in our workflows and it has been implemented by us on both Hadoop and Twister.

7.2.1.2 Multidimensional Scaling (MDS)

MDS is often used in information visualization for exploring similarities in data. Given a pairwise dissimilarity matrix, an MDS algorithm assigns a location to each item in N-dimensional space in a way that the corresponding goal function is minimized. SMACOF is an optimization strategy used in MDS where m-dimensional data items are "mapped" to n-dimensional ($n \ll m$) space with stress function minimized via iterative EM technique. Parallel SMACOF is used in our workflows. The output of PSA is the input of MDS. Because SMACOF is an iterative algorithm, we only implemented it on Twister.

7.2.1.3 MDS interpolation

The memory usage of SMACOF is $O(N^2)$, which tightly limits the practically achievable scale of parallel MDS. Majorizing Interpolation Multidimensional Scaling (MI-MDS) has been proposed which significantly reduces the complexity [24]. The whole data set is split into two parts: sample set and out-sample set. The sample set is fed into PSA and MDS, and sample mapping result is generated. Based on the sample mapping result, MI-MDS can calculate the mapping result of out-sample set. MI-MDS can be easily parallelized because the interpolation of data points is independent. MI-MDS has been implemented on both Hadoop and Twister, called Hadoop-MI-MDS and Twister-MI-MDS. Hadoop-MI-MDS uses HDFS for data store and sharing, while Twister-MI-MDS partitions the data and stages it to compute nodes.

7.2.2 Summary

8

Related Work

There has been substantial research on various issues of distributed computing. We survey below the related work on parallel and distributed file systems, data staging, and task scheduling.

8.1 File Systems and Data Staging

In early days, disk space was quite limited and large data were stored on tapes. The Distributed Parallel Storage Server (DPSS) [71] is a dynamically configurable distributed disk-based cache built on top of a collection of disk servers over Wide Area Network to isolate applications and tertiary archive storage systems. Data copy from tertiary storage system to DPSS takes place on demand when applications request data that are not present in DPSS cache. TCP parameters are tuned and parallelism at different levels such as disk level, controller level, processor level, server level and network level are exploited to provide high performance. Storage Resource Broker (SRB) [26] provides applications uniformed APIs to access heterogeneous distributed storage resources. In addition, SRB employs a metadata catalog service MCAT and provides collection view of a set of data objects. In Data Grid [38], different storage sites are federated to work together to store large data sets and maintain multiple replicas when needed. In grid computing, different sites are federated to provide more accessible resources. Data transferring tools are needed to move data around among participant sites and allow users to upload data to grids. GridFTP [21] [20] extends File Transfer Protocol (FTP) to add features that are critical to high-performance data movement in grids. The newly added features include third-party data transfer, security, striped data transfer, parallel data transfer,

automatic negotiation of TCP buffer sizes, partial file transfer, and support for reliable and restartable data transfer. GridFTP improves interoperability over DPSS because it standardizes message-level protocol while DPSS provides only APIs. Reliable File Transfer (RFT) is built on top of GridFTP to provide Web-Service interface and persist data transfer state to database so that users can inquiry the progress and failed transfer can be resumed from the last checkpoint [64]. Globus Replication Location Service (RLS) manages data replicas and provides a mechanism for registering new replicas and discovering them [37] [39]. Globus Data Replication Service (DRS), which is built on top of RFT and RLS, automatically creates new replicas and registers them to RLS when needed [36]. Stork [75] treats data placement activities as jobs and interacts with high level scheduler such as DAGMan which manages the dependencies among computation and storage jobs. Pre-defined types of data placement jobs (e.g. reserve, transfer, release, locate) are provided to facilitate the specification of common jobs. In addition, Stork natively supports commonly used storage systems (e.g. SRB, UniTree, NeST), data transport protocols (e.g. FTP, GridFTP, HTTP) and middleware (e.g. SRM).

High performance parallel file systems such as Lustre [99], GPFS [97], and PVFS [30] have been developed and used in large clusters to manage data. They are mounted to compute clusters and look like regular local file system from the perspective of users. So it eliminates the need to explicitly stage in data although data are still fetched from remote sites. Besides file systems, other types of abstraction have been proposed and used. Amazon Dynamo [48] uses eventual consistency to balance performance and consistency. Many key-value stores have been developed including Redis, Riak, and Tokyo Cabinet. Couch DB and Mongo DB are document-oriented databases which are designed for storing, retrieving and managing document-oriented data. Amazon S3 and OpenStack Swift implement object stores. BigTable/HBase [32], Cassandra and Hypertable take column family and column store approach to organize data.

8.2 Scheduling

Traditional task scheduling algorithms [102] utilize task graphs which capture data flow and dependency among tasks to make scheduling decisions. Task graphs themselves are not adjusted to improve performance. In [65], some heuristics including MinMin, MaxMin and Sufferage are proposed for scheduling independent computational tasks to compute resources, and analyzed to understand their characteristics. Bag-of-Tasks

[16][111] simplifies task graphs by assuming that tasks of each application are independent, which is motivated by prior efforts such as SETI@home and parameter sweep applications [31]. Infrastructures (e.g. Condor [79] and BOINC) haven been developed and used widely. The traditional task scheduling research takes the strategy that once tasks start running, they are not modified dynamically.

Batch queuing systems [52] such as PBS [104], LoadLeveler [46], Sun Grid Engine [82] and Simple Linux Utility for Resource Management (SLURM) [69] maintain job queues and allocate resources to jobs based on their priorities, resource requirement and resource availability. When a job is scheduled, the requested number of nodes are reserved for a specific period of time even though the resource usage may vary across the phases of the job. Usually the input data are stored in separate storage systems (e.g. data grids, archival) and must be staged in before jobs start to execute. Most of the batch schedulers only map jobs to processing elements (e.g. processors, cores) without taking data affinity into account when they make scheduling decisions. More sophisticated and intelligent data staging mechanisms can improve performance by caching and reusing the data across job runs and interleaving data staging and computation more efficiently. Therefore, they are mainly designed for compute-intensive applications for which data staging is not a critical issue. Backfilling [84] moves small jobs ahead to leapfrog big jobs in front to alleviate fragmentation and improve resource utilization. Backfilling does not delay the first job or any job waiting in the queue depending on its aggressiveness. Resources are shared among jobs in MapReduce while grid systems adopt reservation-based resource allocation. In resource stealing, jobs are not re-ordered or moved in the queue and stealing is done at task level without impacting job scheduling at all, so it is a finer-grained and lower-level optimization of resource usage. Gang scheduling [53] [68] synchronizes all threads/processes of a parallel job by coordinating context switching across nodes so that they are scheduled and de-scheduled simultaneously. Better performance is achieved by using gang scheduling if the processes of a job need to communicate with each other. However, the global synchronization of processes incurs significant overhead. Communication-driven coscheduling algorithms such as Dynamic Coscheduling [103], Spin Block [85], Periodic Boost [85], and Coordinated Coscheduling [17] are proposed to overcome the drawback of gang scheduling. In communication-driven coscheduling, each node runs a scheduler separately and they work in a coordinated manner by passing messages. HYBRID coscheduling [41] combines the merits of gang scheduling and communication-driven coscheduling to improve performance. It split job execution into computation and collective communication phases. When a process enters communication phase, its priority

is boosted with the hope that it will not block the execution of other processes within the same job. Above research does not explicitly consider heterogeneity and is mainly used in homogeneous clusters.

In Condor matchmaking and gangmatching [93], each agent expresses their capabilities and requirements by specifying classified advertisements, which can address the issues of resource heterogeneity and policy heterogeneity. Matchmaker is the component that finds the bilateral or multilateral matching among class-ads.

There has been substantial research on load balancing which tries to balance resource usage in clusters [121]. Pre-emptive process migration supports dynamic migration of processes from overloaded nodes to lightly-loaded nodes. It's possible that the whole system is well balanced while some nodes are idle (e.g. when the number of task processes is less than that of nodes). In that case, traditional algorithms cannot utilize idle nodes while our solution can split running tasks and dispatch spawned tasks to idle nodes. Work stealing [28] enables idle processors to steal computational tasks from other processors and is more communication efficient than its work-sharing counterparts. Our proposed resource stealing shares similar motivations. But the execution model of MapReduce is logically independent of underlying hardware while work stealing is closely coupled with processors. Cycle stealing [27] enables busy nodes to take control of idle nodes, supply them with work, and receive results. The motivation is to harness the otherwise wasted resources of idle nodes. Task splitting yields better load balancing across nodes by dynamically adjusting task granularities [58]. Our proposed resource stealing is applied at a lower level to the resources located on a single node. Iterative MapReduce [50] optimizes the performance of iterative applications by aggressively caching and reusing data across iterations.

The importance of data replication and locality has drawn some attention in grid computing communities. To support fast data access in data grids, Hierarchical Cluster Scheduling and Hierarchical Replication Strategy are proposed that generate redundant copies of existing data across multiple sites and reduce the amount of transferred data [33]. Different dynamic replication strategies, which increase the possibility of local data accessing, are proposed and evaluated to show that the best strategies can significantly reduce network consumption and access latency if access patterns exhibit a small degree of geographical locality [95]. Computation scheduling and data replication in data grids are investigated in [94], which shows it is beneficial to incorporate data location into job scheduling and automatically create new replicas for popular data sets across sites. Their proposed mechanisms outperform traditional HPC approaches for data-intensive computing. The minimization of the loss of data locality is studied in [43] with the assumption that the

number of splits of an item is inversely proportional to the data locality. They found it is NP-hard to find optimal solutions and a polynomial-time approach is proposed to give near-optimal solutions. But their run-time model is different from MapReduce in that the whole data set needs to be staged in before a job can run. Close-to-Files strategy for processor and data co-allocation is proposed and evaluated for multi-cluster grid environments in [83] with the assumption that a single file has to be transferred to all job components prior to execution. A reservation-like scheduling mechanism is adopted. These are not valid in the system I investigate. In [105] execution sites and storage sites are organized into IO communities to reflect physical reality or administration domains so that schedulers can make informed decisions based on data locality, etc. For instance, a job can be scheduled to a community where its input data are stored. The experiment results show that localized I/O yields better performance, which demonstrates the importance of data locality.

For MapReduce, several enhancements have been proposed to improve data locality. Delay scheduling has been proposed to improve data locality in MapReduce [118] [116]. For a system in which most of jobs are short, if a task cannot be scheduled to a node where its input data reside, to delay its scheduling by a small amount of time can greatly improve data locality. As mentioned in the paper, if long tasks are common, delay scheduling is not effective because it will not improve data locality substantially. Our approach improves data locality without incurring additional delay for all workload. Actually, our algorithms can be used in combination with delay scheduling. In Purlieus, MapReduce clusters in clouds are provisioned in a locality-aware manner so that data transfer overhead among tasks is minimized [89]. MapReduce jobs are categorized into three classes: *map-input heavy*, *map-and-reduce-input heavy* and *reduce-input heavy*, for each of which different data and VM placement techniques are proposed accordingly to minimize the cost of data shuffling between map tasks and reduce tasks. In [81], scattered grid clusters controlled by different domains are unified to form a MapReduce cluster by using a Hierarchical MapReduce framework. But they assumed data are fed in dynamically and staged to local MapReduce clusters on demand. To improve speculative execution in Hadoop in heterogeneous environments, LATE is proposed that uses the remaining execution time of tasks as the guideline to prioritize the tasks to speculate and choose fast nodes to run speculative tasks [120]. It has been incorporated into Hadoop 0.21.0 [12] which is used in our tests. Our work shows that LATE is not sufficient to cope with the drastic heterogeneity of network. Shared scans of large popular files among multiple jobs have been demonstrated to be able to improve the performance of Hadoop significantly [18]. It relies on the accurate prediction of future job arrival rates.

The term speculative execution has been used in different contexts. For example, at instruction level, branch predictors [59] guess which branch a conditional jump will go to and speculatively execute the corresponding instructions. For distributed systems where communication overhead is substantial, task duplication [19] redundantly executes some tasks on which other tasks critically depend. So task duplication mitigates the penalty of data communication by running the same task on multiple nodes. Speculative execution in MapReduce employs a similar strategy but is mainly used for fault tolerance. It is implemented in Hadoop to cope with the situations where some tasks in a job become laggard compared with others. The assumption is that the execution time of map tasks does not differ much, which makes it possible for Hadoop to predict task execution time without any prior knowledge. When Hadoop detects that a task runs longer than expected, it starts a duplicate task to process the same data. Whenever any task completes, its other duplicate tasks are killed. This can improve fault tolerance and mitigate performance degradation. However the performance gain is obtained at the cost of duplicate processing and more resource usage. In addition, the speculative execution caused by the nature of map operations does not benefit at all, because duplicate tasks cannot shorten the run time either. Our work is complementary to task speculation in that task splitting and task duplication can be combined together to deal with long running tasks resulting from either the nature of map operations or system failure. Moreover, there has been some research on heterogeneity in MapReduce. A MapReduce implementation for .NET platform was presented in [70].

Some enhancements to vanilla MapReduce model have been proposed. HaLoop [29], Spark [119] and Twister [50] add special support for iterative applications by caching inter-iteration data and reusing them. Spark [119] also supports interactive query. Map-Reduce-Merge [114] enables processing multiple heterogeneous datasets, which facilitates the express of relational operations. MapReduce Online [44] allows intermediate data to be pipelined between operators and supports online aggregation and continuous queries while preserving fault tolerance and programming model of MapReduce. In our Hierarchical MapReduce [81], Map-LocaleReduce-GlobalReduce model is proposed to build a unified MapReduce cluster on top of multiple grid clusters under the control of different domains. Each raw job submitted by users is partitioned to sub-jobs which are dispatched to local MapReduce clusters. The global controller gathers results from local MapReduce clusters and applies global reduce operation. In addition to MapReduce, some other runtimes have been proposed such as Sector/Sphere [57] and Dryad [66]. Dryad allows users to express their jobs as DAGs and automatically manages task execution, data staging and task dependency.

Some schedulers have been developed for MapReduce that support fair resource sharing. Facebook's fairness scheduler aims to provide fast response time for small jobs and guaranteed service levels for production jobs by maintaining job "pools" each of which is assigned a guaranteed minimum share and dividing excess capacity among all jobs or pools [117]. Yahoo's capacity scheduler supports multi-tenancy by assigning capacity to job queues [3]. However, the tradeoff between data locality and fairness is not considered. Dominant Resource Fairness addresses the fairness issue of multiple resources by determining a user's allocation based on his/her dominant share [56]. Quincy is a Dryad scheduler that tackles the conflict between data locality and fairness by converting the scheduling problem to a graph that encodes both network structure and waiting tasks and solving it using a min-cost flow solver [67]. In our work a different approach is taken which has lower complexity. In [98], load unbalancing policy is proposed to balance fairness and performance and minimize mean response time and mean slowdown when scheduling parallel jobs.

9

Conclusions and Future Work

9.1 Summary of Work

9.2 Conclusions and Contributions

9.3 Future Work

Bibliography

[1] Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/.

[2] Clueweb09. http://lemurproject.org/clueweb09.php/.

[3] Hadoop's Capacity Scheduler. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html.

[4] Hands-On Introduction to OpenMP. http://openmp.org/mp-documents/omp-hands-on-SC08.pdf.

[5] LSAP Introduction. http://www.assignmentproblems.com/doc/LSAPIntroduction.pdf.

[6] Microsoft drops Dryad; puts its big-data bets on Hadoop. http://www.zdnet.com/blog/microsoft/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop/11226.

[7] MPICH. http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/.

[8] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[9] Oozie: Yahoo!'s workflow engine for Hadoop. http://rvs.github.com/oozie/index.html.

[10] Open MPI. http://www.open-mpi.org/.

[11] OpenStack Swift. http://swift.openstack.org/.

[12] Speculative execution start up condition based on completion time. https://issues.apache.org/jira/browse/HADOOP-2141.

[13] Top500. http://www.top500.org/.

[14] TORQUE. http://www.adaptivecomputing.com/products/open-source/torque/.

[15] Traces of Google workloads. http://code.google.com/p/googleclusterdata/.

[16] Micah Adler, Ying Gong, and Arnold L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '03, pages 1–10, New York, NY, USA, 2003. ACM.

[17] Saurabh Agarwal, Gyu S. Choi, Chita R. Das, Andy B. Yoo, and Shailabh Nagar. Co-Ordinated Coscheduling in Time-Sharing Clusters through a Generic Framework. *Cluster Computing, IEEE International Conference on*, 0:84, 2003.

[18] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, August 2008.

[19] Ishfaq Ahmad and Yu K. Kwok. A New Approach to Scheduling Parallel Programs Using Task Duplication. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 47–51, Washington, DC, USA, 1994. IEEE Computer Society.

[20] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28(5):749–771, May 2002.

[21] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, volume 0 of *SC '05*, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Ilkay Altintas, Adam Birnbaum, Kim K. Baldridge, Wibke Sudholt, Mark Miller, Celine Amoreira, Yohann Potier, and Bertram Ludaescher. A Framework for the Design and Reuse of Grid WorkFlows. In *International Workshop on Scientific Aspects of Grid Computing*, pages 120–133. Springer-Verlag, 2005.

[23] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Seung H. Bae, Jong Y. Choi, Judy Qiu, and Geoffrey C. Fox. Dimension reduction and visualization of large high-dimensional data via interpolation. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 203–214, New York, NY, USA, 2010. ACM.

[25] Luiz A. Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40:33–37, December 2007.

[26] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '98. IBM Press, 1998.

[27] Sandeep N. Bhatt, Fan R. K. Chung, F. Thomson Leighton, and Arnold L. Rosenberg. On Optimal Strategies for Cycle-Stealing in Networks of Workstations. *IEEE Trans. Comput.*, 46:545–557, May 1997.

[28] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[29] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[30] Philip H. Carns, Walter, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[31] Henri Casanova and Fran Berman. Parameter Sweeps on the Grid with APST. pages 773–787, March 2003.

[32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[33] Ruay S. Chang, Jih S. Chang, and Shin Y. Lin. Job scheduling and data replication on data grids. *Future Gener. Comput. Syst.*, 23(7):846–860, August 2007.

[34] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[35] Yanpei Chen, Archana S. Ganapathi, Rean Griffith, and Randy H. Katz. Analysis and Lessons from a Publicly Available Google Cluster Trace. Technical Report UCB/EECS-2010-95, EECS Department, University of California, Berkeley, June 2010.

[36] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, November 2005.

[37] Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[38] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[39] Ann L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, and Robert Schwartzkopf. Performance and Scalability of a Replica Location Service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Avery Ching, Alok Choudhary, Kenin Coloma, Wei K. Liao, Robert Ross, and William Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Gyu S. Choi, Jin H. Kim, Deniz Ersoz, Andy B. Yoo, and Chita R. Das. Coscheduling in Clusters: Is It a Viable Alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Cheng T. Chu, Sang K. Kim, Yi A. Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[43] Fan Chung, Ronald Graham, Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Maximizing data locality in distributed systems. *J. Comput. Syst. Sci.*, 72:1309–1316, December 2006.

[44] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, page 21, Berkeley, CA, USA, 2010. USENIX Association.

[45] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of scheduling*. Addison-Wesley Pub. Co., 1967.

[46] International Business Machines Corporation. IBM Load Leveler: Users Guide. Technical report, International Business Machines Corporation, Kingston, NY, USA, 1993.

[47] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[48] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[49] Ewa Deelman, Gurmeet Singh, Mei H. Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[50] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung H. Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International*

Symposium on High Performance Distributed Computing, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[51] Mohammad A. Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[52] Dror G. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657), IBM, October 1994.

[53] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[54] Geoffrey C. Fox and Dennis Gannon. Special Issue: Workflow in Grid Systems: Editorials. *Concurr. Comput. : Pract. Exper.*, 18(10):1009–1019, August 2006.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[56] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, page 24, Berkeley, CA, USA, 2011. USENIX Association.

[57] Yunhong Gu and Robert L. Grossman. Sector and Sphere: the design and implementation of a high-performance data cloud. *hilosophical Transactions of the Royal Society - Series A: Mathematical, Physical and Engineering Sciences*, 367(1897):2429–2445, 2009.

[58] Zhenhua Guo, Marlon Pierce, Geoffrey Fox, and Mo Zhou. Automatic Task Re-organization in MapReduce. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 335–343, Washington, DC, USA, 2011. IEEE Computer Society.

[59] L. Gwennap. New algorithm improves branch prediction. *Microprocessor Report*, 9(4):17–21, 1995.

[60] S. Hammoud, Maozhen Li, Yang Liu, N. K. Alham, and Zelong Liu. MRSim: A discrete event based MapReduce simulator. 6:2993–2997, August 2010.

[61] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of CIDR'11, Asilomar, California, USA*, 2011.

[62] Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in Open MPI. In *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, pages 49–58, New York, NY, USA, 2009. ACM.

[63] Joshua Hursey, Thomas Naughton, Geoffroy Vallee, and Richard L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[64] William A. Ian, Ian Foster, and Ravi Madduri. Reliable Data Transport: A Critical Service for the Grid. In *In Building Service Based Grids Workshop, Global Grid Forum 11*, 2004.

[65] Oscar H. Ibarra and Chul E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Non-identical Processors. *J. ACM*, 24(2):280–289, April 1977.

[66] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[67] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[68] Morris A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–12, New York, NY, USA, 1997. ACM.

[69] Morris A. Jette, Andy B. Yoo, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[70] Chao Jin and Rajkumar Buyya. MapReduce Programming Model for .NET-Based Cloud Computing. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume 5704 of *Euro-Par '09*, pages 417–428, Berlin, Heidelberg, 2009. Springer-Verlag.

[71] William E. Johnston, William Greiman, Gary Hoo, Jason Lee, Brian Tierney, Craig Tull, and Douglas Olson. High-speed distributed data handling for on-line instrumentation systems. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–19, New York, NY, USA, 1997. ACM.

[72] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[73] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, May 2010. IEEE Computer Society.

[74] David G. Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

[75] Tevfik Kosar and Miron Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 342–349, Washington, DC, USA, 2004. IEEE Computer Society.

[76] Rob Latham, Rob Ross, and Rajeev Thakur. The Impact of File Systems on MPI-IO Scalability. pages 87–96. 2004.

[77] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[78] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[79] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. pages 104–111, 1988.

[80] Wei Lu, Jared Jackson, Jaliya Ekanayake, Roger S. Barga, and Nelson Araujo. Performing Large Science Experiments on Azure: Pitfalls and Solutions. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 209–217, Washington, DC, USA, 2010. IEEE Computer Society.

[81] Yuan Luo, Zhenhua Guo, Yiming Sun, Beth Plale, Judy Qiu, and Wilfred W. Li. A hierarchical framework for cross-domain MapReduce execution. In *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, ECMLS '11, pages 15–22, New York, NY, USA, 2011. ACM.

[82] Gentzsch Sun Microsystems. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, Washington, DC, USA, 2001. IEEE Computer Society.

[83] H. H. Mohamed and D. H. J. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. pages 287–298, 2004.

[84] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, June 2001.

[85] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. Alternatives to coscheduling a network of workstations. *J. Parallel Distrib. Comput.*, 59:302–327, November 1999.

[86] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 26(2):62–76, February 1993.

[87] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[88] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international*

conference on Management of data, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[89] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. ACM.

[90] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, October 2005.

[91] Xiao Qin, Hong Jiang, Adam Manzanares, Xiaojun Ruan, and Shu Yin. Dynamic load balancing for I/O-intensive applications on clusters. *Trans. Storage*, 5(3), November 2009.

[92] Xiaohong Qiu, Jaliya Ekanayake, Scott Beason, Thilina Gunarathne, Geoffrey Fox, Roger Barga, and Dennis Gannon. Cloud technologies for bioinformatics applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, New York, NY, USA, 2009. ACM.

[93] Rajesh Raman, Miron Livny, and Marvin Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, HPDC '03, Washington, DC, USA, 2003. IEEE Computer Society.

[94] Kavitha Ranganathan and Ian Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, Washington, DC, USA, 2002. IEEE Computer Society.

[95] Kavitha Ranganathan and Ian T. Foster. Identifying Dynamic Replication Strategies for a High-Performance Data Grid. In *Proceedings of the Second International Workshop on Grid Computing*, GRID '01, pages 75–86, London, UK, 2001. Springer-Verlag.

[96] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[97] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[98] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: the case for load unbalancing and fairness. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 211–220, 2000.

[99] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[100] Ruchir Shah, Bhardwaj Veeravalli, and Manoj Misra. On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments. *IEEE Trans. Parallel Distrib. Syst.*, 18:1675–1686, December 2007.

[101] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[102] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[103] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 231–256, London, UK, 1998. Springer-Verlag.

[104] Veridian Systems. OpenPBS v2.3: The portable batch system software. Technical report, International Business Machines Corporation, Mountain View, CA, USA.

[105] Douglas Thain, John Bent, Andrea A. Dusseau, Remzi A. Dusseau, and Miron Livny. Gathering at the well: Creating communities for grid I/O. In *Proceedings of Supercomputing 2001*, Denver, Colorado, November 2001. IEEE CS Press, Los Alamitos, CA, USA.

[106] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[107] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[108] A. Thusoo, J. S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005, March 2010.

[109] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep S. Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[110] Maur'ıcio Tsugawa and José A. B. Fortes. A virtual network (ViNe) architecture for grid computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, page 148, Washington, DC, USA, 2006. IEEE Computer Society.

[111] Chuliang Weng and Xinda Lu. Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid. *Future Gener. Comput. Syst.*, 21:271–280, February 2005.

[112] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comput. Syst.*, 15(5-6):757–768, October 1999.

[113] Joachim Worringen, Jesper L. Traff, and Hubert Ritzdorf. Fast Parallel Non-Contiguous File Access. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[114] Hung C. Yang, Ali Dasdan, Ruey L. Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[115] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. Technical report, Journal of Grid Computing, 2005.

[116] 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 *Proc. of EuroSys*, pages 265–278. ACM, 2010.

[117] Matei Zaharia. The Hadoop Fair Scheduler. http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt.

[118] Matei Zaharia, Dhruba Borthakur, Joydeep S. Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical report, EECS Department.

[119] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, page 10, Berkeley, CA, USA, 2010. USENIX Association.

[120] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[121] Xiaodong Zhang, Li Xiao, and Yanxia Qu. Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS '00, Washington, DC, USA, 2000. IEEE Computer Society.

Curriculum Vitae

Name of Author:	Zhenhua Guo
Date of Birth:	Sepetember 4, 1983
Place of Birth:	Shuozhou, Shanxi, China
Education:	
March, 2009	Master of Science, Computer Science
	Indiana University, Bloomington, IN, USA
June, 2006	Bachelor of Science, Computer Science and Technology
	Peking University, Beijing, China
Experience:	
Sepetember, 2006 - present	Research Assistant
	Community Grids Laboratory, Indiana University
	Bloomington, Indiana, USA
June, 2010 - August, 2010	Summer Intern, Amazon
	Amazon.com LLC, Seattle, Washington, USA
June, 2011 - August, 2011	Summer Intern, Amazon
	Amazon.com LLC, Seattle, Washington, USA
Honors/Affiliations:	
	Graduate Student Scholarship - Indiana University, Bloomington (2006 - 2012)
	Lenovo Student Scholarship - Peking University, China (2004)
	Committer for Apache Rave