Design and Implementation of Scalable and Efficient Programming Models for Fast Computation and Data Processing

by

Xinyu Que

A dissertation proposal submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 3, 2013

Keywords: GAS, ARMCI, MapReduce, Virtual Shuffling, Community Detection, Louvain Algorithm

Approved by

Weikuan Yu, Chair, Associate Professor of Computer Science and Software Engineering
Alvin S. Lim, Associate Professor of Computer Science and Software Engineering
Wei-Shinn Ku, Associate Professor of Computer Science and Software Engineering

Abstract

Despite the tremendous growth of computational power, scientific applications and business data analytics continue to face many challenges such as programming productivity, application scalability, and efficiency. Recently, Global Address Space (GAS) or Partitioned Global Address Space (PGAS) programming models are emerging as scalable alternatives for fast computation because of their ability to alleviate programming burden by supporting data access to both local and remote memory through a simple shared-memory addressing model. Meanwhile, with the exponential growth of the digital universe, the MapReduce programming model becomes popular for data analytics because of its ease of use, low cost on commodity hardware, fault tolerance, and programming flexibility. Furthermore, with social media data gets bigger, relationships inside social media data get complex and have normally been modeled as massive graphs, which require scalable algorithms to analyze the real-world graphs for data processing.

This dissertation investigates the research challenges in those directions and contributes efficient and scalable programming models for fast computation and data processing. It first focuses on addressing the critical challenges faced by the underlying runtime systems of GAS model on petascale systems. In particular, I have proposed and designed a Hierarchical Cooperation (HiCOO) supporting scalable communication for GAS programming models, which is able to realize scalable resource management and achieve resilience to network contention while at the same time maintaining or enhancing the performance of scientific applications. The second study is to address the performance challenge in the existing MapReduce programming model. I have revealed a number of issues faced by the current MapReduce Programming model and proposed a novel virtual shuffling strategy to enable efficient data movement for MapReduce data shuffling, which is able to significantly reduce

disk I/O accesses and results in performance improvement and power consumption saving. The third study is on large-scale graph processing. I have designed and implemented a parallel community detection algorithm over distributed memory system, which can perform community analysis in real-time for massive graphs.

Acknowledgments

First of all, I would like to thank my supervisor Dr. Weikuan Yu for his guidance, instructions, and generous support throughout these years. I am fortunate to be his student and I am influenced by his energy, his working attitude, and his emphasis on attacking the real-work challenges and building large scale systems. I appreciate all his contributions that make my Ph.D. experience productive and stimulating.

Additionally, a special thank goes to my mentors Mr. Vinod Tipparaju (Now at AMD) and Dr. Jeffrey Vetter at Oak Ridge National Lab, who guided me on the programming models. I also appreciate Dr. Fabrizio Petrini and Dr. Fabio Checconi at IBM T.J. Watson research center who helped me on the parallel graph algorithm area. I also would like to thank my committee member Dr. Alvin Lim and Dr. Wei-shin Ku, and my university reader Dr. Shiwen Mao for their time, patience and suggestions that led to me improving this work.

I would like to thank all members in the PASL group in Computer Science & Software Engineering, Auburn University. The group has been a source of friendships as well as good advice and collaboration. It is an excellent and fruitful group, providing me a good research environment. I would like to acknowledge my group member Yandong Wang. We worked together on the Hadoop-A project, and I very much appreciated his enthusiasm, intensity and willingness towards research. Other past and present group members that I have had the pleasure to work with or alongside of are graduate students Yuan Tian, Cong Xu, Bin Wang, Zhuo Liu, Xiaobin Li, Yizheng Jiao, Teng Wang, Patrick Carpenter; visiting professors Bongen Gu and Chunxiang Wu who have come through the lab.

Finally, I am indebted to my wife Yuan He, my daughter Rachel, my son Jonathan, my grandma Shuzhang Yan, my parents Cunguang Que and Yanqing Lu, my parents-in-law Hongfei He and Shuzhen Li, my sister Bei Que, brother-in-law Liangyuan Lu, and niece

Zexi Lu who give me a warm family with never-ending support, care and encouragement in my whole life. All of these support me with love and joy through my academic study and research work.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ARMCI Aggregate Remote Memory Copy Interface

HiCOO Hierarchical Cooperation

PGAS Partitioned Global Address Space

HDFS Hadoop Distributed File System

Chapter 1

Introduction

Innovation in technologies pushes the revolution of computational power in our society. Researchers and engineers make use of the tremendous computational power to solve the complexity problems such as climate change, nuclear fusion, drug design, web log analytics and fraud detection. To this purpose, parallel programming models become the main bridge between human and hardwares and are evolving fast to adapt to the technological advances. Amongst these parallel programming models, the Global Address Programming (GAS) model becomes more and more popular and researchers in scientific domains use it as a popular interface to solve problems on the supercomputers, which are normally equipped with cutting edge hardware technologies.

In the meantime, the digital universe continues to expend rapidly, according to IDC [1], more than 40 zettabyte (1,000 exabytes) data will be generated by 2020. With the exponential growth in the digital universe, there is an urgent need to scalably and efficiently process the Big Data [2]. Introduced by google, MapReduce has been widely accepted by the community for large scale data analytics because of its applicability on low cost commodity hardwares as well as powerful fault tolerance characteristic. With data getting bigger, various social media datasets emerge as more complex, more semi-structured, and more densely connected. The complexity of such Big Data has been modeled explicitly as massive graphs with billion of edges, which require deeper processing for further understanding the data. It is critical to design scalable and efficient parallel algorithms to mine these real-world massive graphs. One of the representative problems in social media is community detection in complex social graphs.

In the rest of this chapter, We first introduce some background of GAS programming mode and its runtime system. Following that we provide an introduction on MapReduce. Then we describe the community detection problem. In the end, we provide an overview of research in this dissertation.

1.1 GAS programming model and Its Runtime System

Scientific applications such as climate modeling, life science, and energy production are normally computation intensive. Several supercomputing sites have deployed systems with extreme amounts computational power [3] to serve the need of solving such complex problems. For example, the Sequoia and Titan Cray XT5 system in the U.S. can perform tens of 10^{15} floating point operations per second (petaflop). While supercomputing systems grow to unprecedented number of processors (exscale supercomputers in the near future), scientific applications continue to face many challenges such as programming productivity, application scalability, and efficiency.

In this aspect, Global Address Space (GAS) or Partitioned Global Address Space (PGAS) models emerging as scalable alternatives because they have the ability to alleviate programming burden by supporting data access to both local and remote memory through a simple shared memory styled access. PGAS languages like Unified Parallel C (UPC) [4], Co-Array Fortran (CAF) [5], and GAS libraries such as Global Arrays (GA) Toolkit [6] are becoming increasingly popular. Recently, a slightly different category of PGAS model, termed Asynchronous Partitioned Global Address space model, has emerged to add additional capabilities such as remote method invocations. IBM X10 language [7] and Asynchronous Remote Methods (ARM) [8] in UPC have pioneered this new model.

All the above mentioned GAS languages and libraries use the services of an underlying communication library (which we refer to as the GAS Runtime) for serving their communication needs. GAS languages normally use this runtime as a compilation target to do the data transfers on distributed memory architectures. They have a translation layer that translates

a memory access to a corresponding data transfer on the underlying system. ARMCI (Aggregated Remote Memory Copy Interface) [9] is a popular GAS runtime that has been used to implement both PGAS languages (such as Co-Array Fortran) and GAS libraries (such as Global Arrays). ARMCI is highly scalable and has been ported to a variety of environments and platforms. Recently, a scalable implementation of ARMCI (described in [10]) was made available on the Jaguar Cray XT5 supercomputer at Oak Ridge National Laboratory.

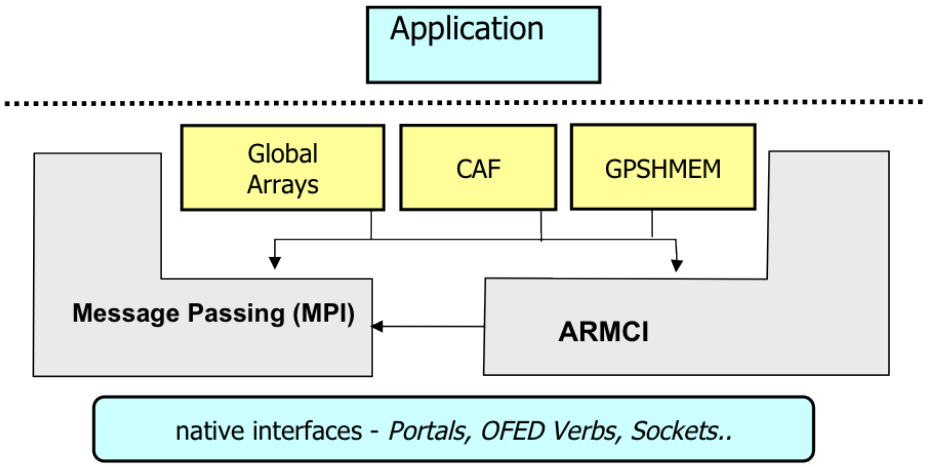


Figure 1.1: Typical Usage of ARMCI

Typical structure of an application using ARMCI is shown in Figure 1.1. ARMCI relies on a message-passing library and elements of the execution environment (job control, process creation, interaction with the resource manager) and provides all the communication need for the GAS languages and libraries. It uses the fastest available mechanism underneath to transmit data wherever possible. For example, it uses Portals library for inter-node communication on the Cray XT5 systems [10]. ARMCI offers an extensive set of functionalities in the area of RMA communication: 1) data transfer operations (Get, Put Accumulate); 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. Communication in most of the non-collective operations is implemented as one or more ARMCI communication operations. ARMCI also supports blocking and

non-blocking versions of contiguous, strided and vector data transfer operations along with Read-Modify-Write operations for the special need of scientific applications.

1.2 MapReduce Programming Model

In the era of Big Data [2], processing explosive amounts of data in a scalable, reliable and efficient manner to mine critical knowledge for human intelligence is becoming one of the most important challenges. For example, AT&T currently processes close to 20 petabytes of data every 24 hours, and Google processes more than 1 petabytes of information every hour [11]. To be able to process data-intensive analysis in a scalable and fault-tolerant manner in a distributed environment, Google introduced a distributed and parallel programming model called MapReduce [12]. Due to its ease of programming, scalability, especially highly fault-tolerant and applicability on low-cost hardware MapReduce paradigm has become the favor of many commercial enterprises such as web crawling, financial services and telecommunications.

Hadoop [13] is an open-source implementation of MapReduce, supported by leading IT companies such as Google and Yahoo!, and widely adopted and deployed in industry on several thousands of commodity machines. Hadoop implements MapReduce framework with two categories of components: a JobTracker and many TaskTrackers. TaskTrackers are managed by the JobTracker and launched on each computational node to perform the tasks they receives from JobTracker. Data processing is performed in parallel through two main functions: map and reduce. The JobTracker is in charge of scheduling the map tasks (MapTasks) and reduce tasks (ReduceTasks) to TaskTrackers. It also monitors job progress, collects run-time execution statistics, and handles possible faults and errors through task re-execution.

The general workflow of Hadoop MapReduce is shown in the Figure 1.2. Hadoop consists of three main execution phases: *map*, *shuffle*, and *reduce*. When a user job is submitted to the JobTracker, its input dataset is divided into many data splits and stored in HDFS. The

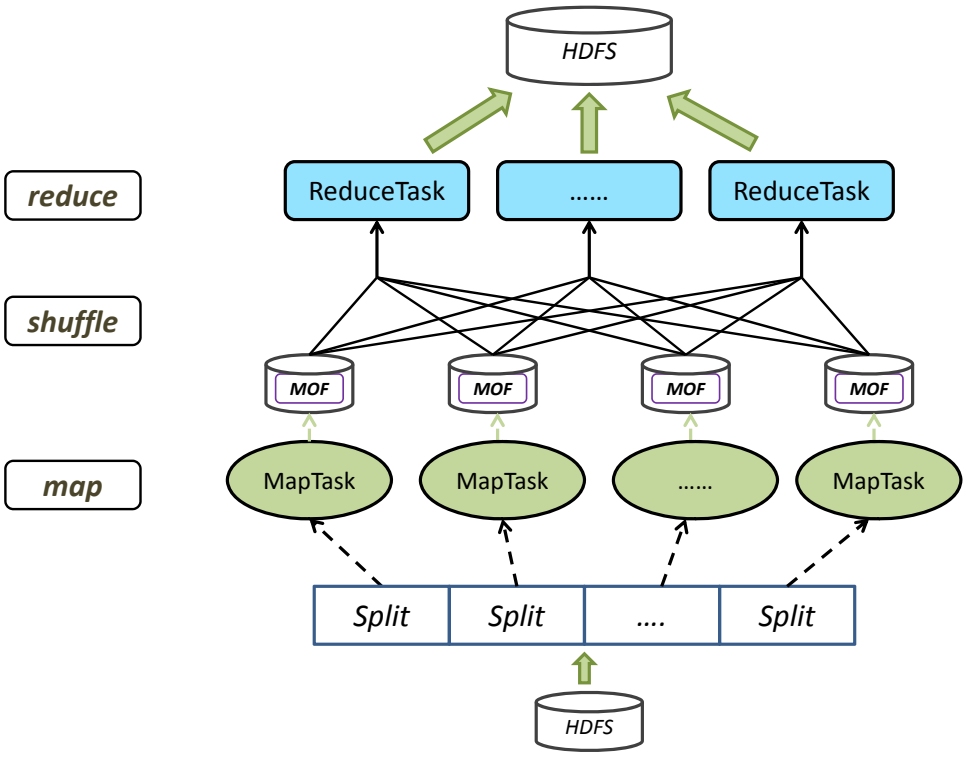


Figure 1.2: Workflow of MapReduce Programming Model

MapTask first reads the splits from HDFS and performs the map function. The corresponding output, which is called Map Output File (MOF), will be stored locally. As soon as MOFs are available, the ReduceTask starts fetching a partition (also called segment) that is intended for it, from all the MOFs, which leads to all-to-all shuffle. The ReduceTask also merges the segments while fetching. Once all the segments are locally available, the ReduceTasks starts processing the merged segments using the reduce function. The final result is then stored to Hadoop Distributed File System [14].

1.3 Community Detection in Graphs

The real-world graphs representation of the Big Data [2] depict the complexity relationship and become more and more popular format for data mining. There are many important kernels to serve the purpose of mining graphs such as *breadth first search*, *single*

source shortest path, and *community detection* etc. Amongst them, the *community detection* is the most challenging problem. A graph with communities is shown in the figure 1.3, where the vertices with densely connection are likely to form communities. The connections among communities are much sparser.

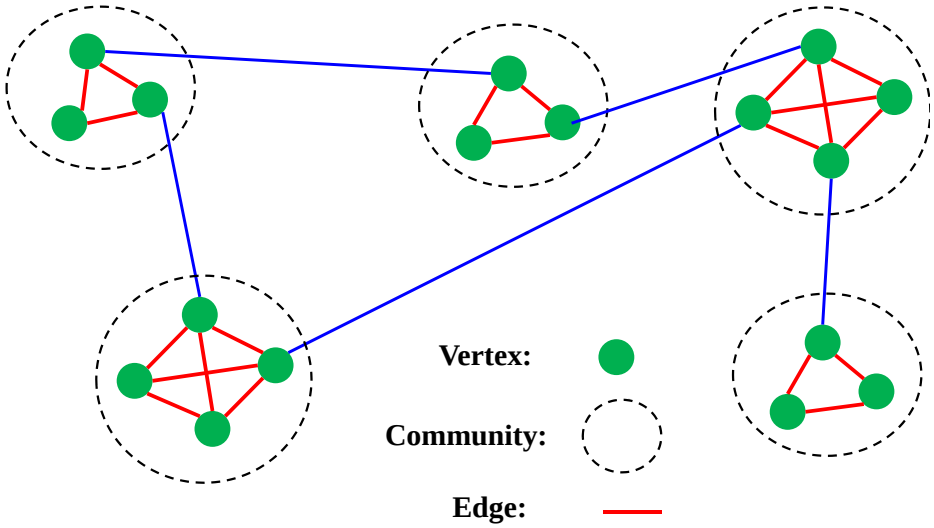


Figure 1.3: A Graph with Communities

Community detection spans many areas such as health care, social networks, systems biology, power grid etc. It has been so extensively investigated over the last few years [15]. The goal is to identify the modules and, possibly, their hierarchical organization. No riguous mathematical definition of the community structure is available yet. The modularity proposed by Newman is by far the most used and best known quality function to quantify community structure in a graph. High modularity indicates good partition, which corresponds to good modularity value. Modularity maximization is currently the most popular class of methods to detect communities in graph.

The technique used for modularity maximization can be categorized into four classes, namely greedy, simulated annealing, extremal optimization and spectral optimization. Greedy based optimization apply different approach to merge the vertices to form communities for higher modularity, which normally generates high quality communities and attracts a great

deal of research interest [16, 17, 18, 19, 20]. Simulated annealing adopts probabilistic procedure for global optimization on modularity [21, 22], which is slow and can only be used for small graphs. Extremal optimization (EO) is a heuristic search procedure and Spectral optimization takes use of the eigenvalues and eigenvectors of a special matrix for modularity optimization [23, 24, 25, 26, 27]. These two methods normally lead to poor results on large networks with many communities.

The focus of this dissertation is on scalable and efficient programming models for fast computation and data processing, which aims at solving the challenges comes from rapidly increasing of the computational power and exponential exploration of the data. To be specific, this dissertation makes the following research contributions:

1. A framework HiCOO [28], which supports scalable communication architecture in a representative GAS runtime system for scalable resource management and contention attenuation on petascale Cray XT5 systems, is proposed and designed;

2. Virtual Shuffling is proposed and designed as a new strategy which enables efficient data movement and relieves the disk contention for MapReduce programming model.

3. A novel parallel community detection algorithm is designed and implemented over distributed memory systems for tackling massive graphs.

The remainder of the dissertation is organized as follows. In Chapter 2, we present the problem statement, which reveals the challenges in the current GAS programming model and MapReduce infrastructure. In Chapter 3, we first review the prior work on how to improve the scalability of communication runtime. We then describe the prior work of MapReduce in a number of directions, including performance tuning, data communication, and Energy efficiency. Following that is a review on the parallel community detection algorithm. The detailed design and implementation will be explained in chapter 4. In Chapter 5, we present experimental results of performance evaluation with many different types of benchmarks,

which contains both micro benchmark and applications. We conclude the dissertation in Chapter 6 and point out directions for future research in Chapter 7.

Chapter 2

Problem Statement

This chapter discusses the detailed challenges to be investigated in this dissertation. Firstly, it presents the scalability and contention challenges of underlying GAS programming model runtime system towarding exascale. Then it explains the severe performance problem during the data movement phase of the MapReduce programming model. Finally, it discusses the challenges involved for mining massive graphs.

2.1 Challenges in GAS Runtime System

ARMCI has recently been enabled for Cray XT5 using the native portals communication library [10]. However, running a GAS model and its underlying GAS runtime in the context of a real scientific application at a scale similar to Jaguar (200,000+ cores) has brought forth a few staggering challenges. These challenges are a result of the characteristics and asynchronous one-sided features of the GAS runtime. The first is that of resource management, incurred by unpredictable communication patterns and communication resources (such as buffers) that need to be allocated to support it. The second challenge is that of network contention – allowing any process to access the address space of any other process and supporting load balancing at the same time create an environment that is prone to contention.

2.1.1 ARMCI Process Management for One-Sided Communication

ARMCI guarantees that its one-sided operations are fully unilateral, i.e., may complete regardless of the actions taken by the remote processes. In particular, polling the application by remote processes (implicitly when making a library call, or explicitly by calling

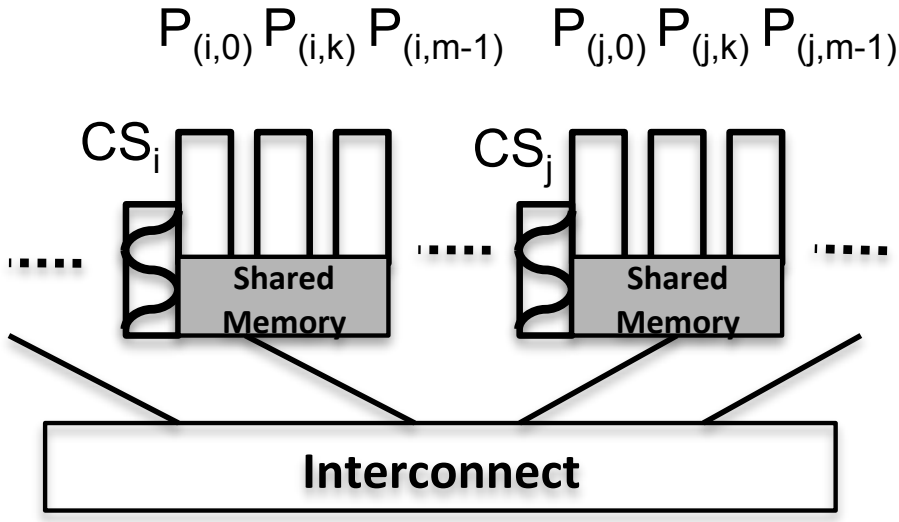


Figure 2.1: ARMCI Process Management

provided polling interface) is not required for communication progress. This is realized by introducing a communication helper thread (a.k.a communication server) at each compute node. This communication helper thread is created by the lowest ranked process (*master*) on a node. An area of shared memory is allocated for these processes. The communication server (CS) handles remote one-sided requests on behalf of all local processes, and exchanges data with them through the shared memory. Similar to what described earlier, the communication server pre-allocates buffers and related data structures for remote requests, in order to support direct one-sided communication for all operations (particularly for lock, unlock, accumulate, and noncontiguous data transfer operations) and allow one process to asynchronously initiate an operation without the involvement of the targeted process.

Figure 2.1 shows the process management of ARMCI. On two arbitrary nodes, i and j, each has a set of parallel processes. All processes have a global rank. Processes on node i are also denoted as $P_{(i,k)}$, $\forall\, k \in [0, m-1]$. An area of shared memory is allocated for these m processes. The lowest ranked process $P_{(i,0)}$ creates a separate thread as a communication server CS_i. The communication server CS_i communicates with all intra-node processes

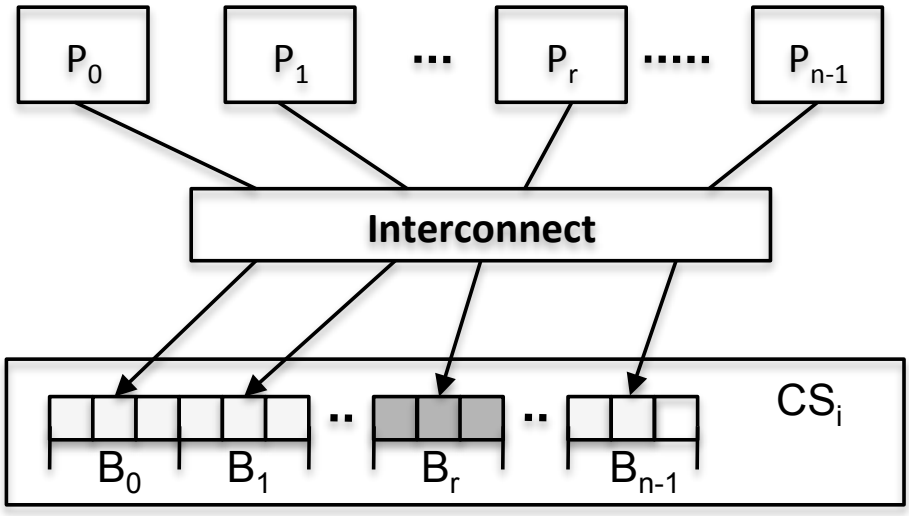


Figure 2.2: ARMCI Server's Request Buffer Management

through the shared memory and handles all incoming inter-node one-sided communication requests on behalf of them.

Every communication server has to pre-allocate request buffers for for all remote peer processes. Figure 2.2 shows the request buffer management of CS_i. Each processes is denoted based on its global rank P_r, $\forall\, r \in [0, n-1]$. A set of request buffers are allocated for each remote process, e.g. B_r for P_r.

2.1.2 Critical Challenges for GAS Runtime

To better formulate the memory resource management of ARMCI, We define *virtual topology* as a means to represent the graph of resource allocation. In the case of memory buffers for communication, a directed graph can represent the resource allocation of buffers amongst all nodes. A graph G: (V, E) consists of a set of vertices V and a set of edges E. A vertex i represents all processes and the CS on a single node i. A directed edge E(i, j) from i to j denotes the fact that there is a set of request buffers allocated on node i for tasks on node j. For an ARMCI application running on N nodes, this representation of buffer allocation forms a FCG with $N * (N-1)$ directed edges. There are $(N-1)$ outgoing edges

at each vertex (node), representing $N-1$ sets of buffers from $N-1$ remote nodes. Figure 2.3 shows the resource allocation graph for a 6-node case.

The directed graph representation of resource allocation in Figure 2.3 reveals two critical challenges that a GAS model (in our case, Global Arrays) poses to its underlying GAS runtime (in our case, ARMCI).

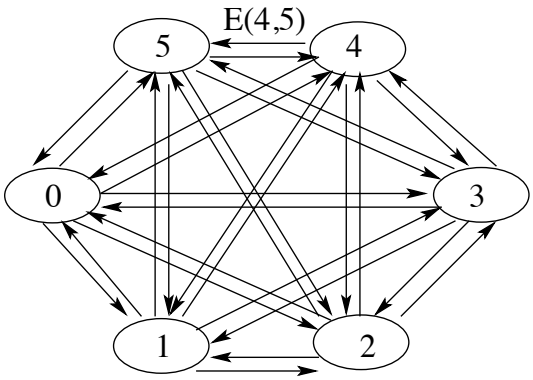


Figure 2.3: A Directed Graph Representing Resource Allocation for One-Sided Requests

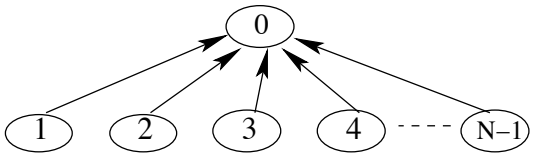


Figure 2.4: A Flat-Tree Representation of Contention among Communication Requests

Resource Management – The first challenge is on the allocation of resources for communication. Consider an example of the targeted systems for our research, the Cray XT5. Cray XT5 has Seastar2+ interconnect and uses the connection-less Portals messaging library as the lowest level communication protocol. To support the connection-less Portals interface, the Cray Seastar2+ allows for 256 simultaneous message streams. When additional streams need to be initiated (or in case of resource exhaustion), the Cray BEER (Basic End to End Reliability) protocol does the necessary flow control and handles reliability. This means that the resource allocation problem for ARMCI communication buffers (where a set of buffers needs to be allocated for every incoming edge as shown in Figure 2.3) maps to parallel message streams in Portals but at a different scale. The total request buffer requirement in ARMCI for the FCG would be roughly $N * B * M$, where N is the total number of processes (actually slightly smaller than N due to local processes), B the buffer size, and M the set of buffers per process. With only two 16-KB buffers per process, it would

require 1,024 MB per CS to support parallel programs with 32,000 processes, and 32 GB per CS on an future system with a million processes.

Contention – Another challenge revealed by the FCG model is the potential contention that could be caused by many concurrent requests to a single node. Because all nodes (vertices) are directly connected, the paths for requests from all nodes to traverse a virtual FCG and reach one node can be represented as a flat tree of depth 1. Figure 2.4 shows a tree representation of request traversal paths to Node 0. Such a flat tree is very vulnerable to transient hot-spot access scenarios, such as when thousands of processes simultaneously accessing one data element in an address space. These scenarios create a severe hot-spot contention problem in addition to the resource allocation problem described above. Under such scenarios, significant burden is placed on the physical network, which will be forced to adopt some throttling mechanisms, typically causing serious slowdown of the entire communication and jeopardizing the system productivity.

2.2 Challenges in the MapReduce Programming Model

Hadoop MapReduce has been widely adopted in industrial communities as the engine to perform data analytics. However, it is still facing a lot of challenges e.g. performance, resource management, quality of service, and so on. Amongst these challenges, the performance is a major concern of current research focus. As mentioned in Section 1.2, the MapReduce programing model contains three major phases (Map, Shuffle, and Reduce). All these three phases involve a great deal of I/O activities to read input and store output repetitively. Especially in the all-to-all shuffling, which crosses the network bisection and severely hurts the performance.

As explained in Section 1.2, right after the finish of the MapTasks, ReduceTasks start to fetch the <key, val> pairs from all the MOFs on remote MapTasks node. The fetched intermediate data has to be merged before the reduce function can be applied. During this phase, data is moved from the map nodes' disks rather than their main memories,

through the cluster network, to the disks of the reduce nodes, incurring both disk and network latencies. Even worse, due to the need of the scalability, inside the ReduceTasks, the Hadoop MapReduce constrains the number of concurrent connections to the MapTasks and performs the polling of the intermediate data sequentially and adopts a combination of in-memory merge and on-disk external merge to handle this process. However, this strategy can result in a slow merging phase when ReduceTask has a huge amount of intermediate data to handle. The reason is that, with increasing intermediate data size, on-disk merge becomes dominant in the merging phase because of limited amount memory. Consequently, the slow merging phase causes long delay to the reducing phase inside ReduceTask. In addition, disk bandwidth is a scare resource on the nodes, aggressive external merge can cause heavy disk contention among all the tasks running on the same node resulting in the performance degradation of the entire system, which has been observed from production MapReduce clusters [29].

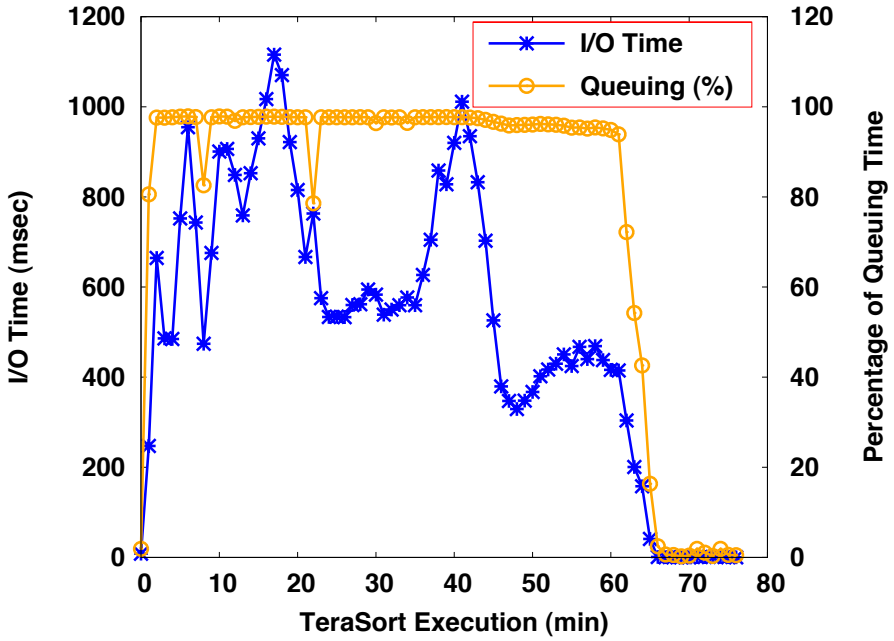


Figure 2.5: Disk I/O Contention in MapReduce Applications

To deeply understand the importance of this problem, we conduct a data-intensive MapReduce test case, running 200GB TeraSort on 10 slave nodes. We have examined the

wait (queuing) time and the service time of I/O requests during the execution. Figure 2.5 shows the results, where X-axis is the execution time, the left Y-axis is the I/O service time, and the right Y-axis is the queuing percentage. As shown in Figure 2.5, the wait time can be more than 1,100 milliseconds. Worse yet, most I/O requests are spending close to 100% of this time waiting in the queue, which means the disk is not able to keep up with the requests. Because the shuffling of intermediate data competes for disk bandwidth with MapTasks, which significantly overloads the disk subsystem. Disk subsystem becomes a serious bottleneck due to the severe disk I/O contention in data-intensive MapReduce programs, which entails further research on efficient data shuffling techniques.

2.3 Challenges in Parallel Community Detection

With the rapid growth of the digital universe, the real-world graphs tend to be extremely huge with billions of vertices and hundreds of billion edges. and follow the power law distribution. Those graphs decompose naturally into communities where vertices are densely connected within the community and have much sparser connection between the communities [30]. The communities from large graphs carry great scientific and practical value because they typically correspond to behavior or functional units of the network, such as social groups in a social graph. Community detection provides a valuable kernel to analyze and mine the Big Data. However, efficient analysis and processing on large-scale graphs remains a challenge, because of pool locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution [31, 32].

2.3.1 Problem Definition

A weighted graph G can be represented as a 2-tuple (V, E), where V denotes a set of vertices, E a set of edges. When $u, v \in V$, an edge $e(u, v) \in E$ has a weight $w_{u,v}$. The goal of community detection is to partition a graph into a set of disjoint communities C, as

described in Equations 2.1 and 2.2.

$$\cup c_i = V, \forall c_i \in C \tag{2.1}$$

$$c_i \cap c_j = \emptyset, \forall c_i, c_j \in C \tag{2.2}$$

When a graph is partitioned into communities, vertices in the same community are then densely connected while those in different communities are only sparsely connected. The quality of community detection algorithms is often measured by the metric *modularity* Q (see Equation 2.3) [33],

$$Q = \sum_{c_i \in C} \left[\frac{\Sigma_{in}^{c_i}}{2m} - \frac{\Sigma_{tot}^{c_i}}{4m^2} \right], \tag{2.3}$$

where $\Sigma_{in}^{c_i}$ is the sum of the weights from all internal edges of Community c_i, represented as $\sum w_{u,v}, \forall u, v \in c_i$ and $e(u,v) \in E$, $\Sigma_{tot}^{c_i}$ the sum of the weights from the edges incident to any vertex in Community c_i, represented as $\sum w_{u,v}, \forall u \in c_i$ and $e(u,v) \in E$, and m ($\sum_{e(u,v) \in E} w_{u,v}$) is the sum of the weights from all edges in the entire graph.

2.3.2 Sequential Louvain Algorithm

Modularity has been widely used to compare the quality of the communities obtained by different algorithms, hence an objective function to optimize [34] by many. However, modularity optimization is an NP-complete problem [35]. Thus research efforts have focused on approximation algorithms that are usually heuristic-based and yield suboptimal detection of communities. Among many different approaches, the state-of-the-art Louvain algorithm [16] adopts a greedy policy that can find high modularity communities in large graphs in a short time and unfold a complete hierarchical community structure. This algorithm has been

employed for community detection for a great variety of purposes.

$$\begin{aligned}\Delta Q_{u\to c_i} &= \left[\frac{\Sigma_{in}^{c_i}+w_{u\to c_i}}{2m}-\left(\frac{\Sigma_{tot}^{c_i}+w(u)}{2m}\right)^2\right] \\ &\quad -\left[\frac{\Sigma_{in}^{c_i}}{2m}-\left(\frac{\Sigma_{tot}^{c_i}}{2m}\right)^2-\left(\frac{w(u)}{2m}\right)^2\right] \\ &= \frac{w_{u\to c_i}}{2m}-\frac{\Sigma_{tot}^{c_i}*w(u)}{2m^2} \end{aligned} \tag{2.4}$$

The Louvain algorithm greedily maximizes the modularity gain $\Delta Q_{u\to c_i}$ when moving an isolated vertex u into Community c_i. It can be calculated by Equation 2.4, where $\Sigma_{in}^{c_i}$ and $\Sigma_{tot}^{c_i}$ have been defined in Equation 2.3, $w(u)$ ($\sum_{e(u,v)\in E} w_{u,v}$) is the sum of the weights of the edges incident to vertex u, $w_{u\to c_i}$ ($\sum_{v\in c_i} w_{u,v}$) is the sum of the weights of the edges from u to vertices in Community c_i. m is the same as defined in Equation 2.3.

The original Louvain algorithm only considers the undirected graphs. The algorithm tries to find the communities with the maximum modularity Q, as shown in Algorithm 1. It initially creates one community per vertex. For each vertex u, it then considers the neighbor v of u and evaluates the gain of the modularity(ΔQ) which could take place by removing u from its community and by placing it in the community of c_v^k. The vertex u is placed in the community $\hat{c_v^k}$ for which the ΔQ is maximum. If there is no positive gain, u stays in the original community (Lines 8–13). This process is applied repeatedly and sequentially for all nodes until no further improvement can be achieved and the first phase is then complete. The latest community information (C^{k+1}) and modularity (Lines 19–23) thus can be obtained. The second phase of the algorithm is to build a new graph whose vertices are now the communities found previously. To do so, the new vertex set V^{k+1} consists of the latest community (c_v^k), and the weights of the edges between the new vertices are given by the sum of the weight of the edges between vertices in the corresponding two communities [36]. The edges between vertices of the same community lead to self-loops for this community in

Algorithm 1: Sequential Louvain Algorithm

Input: k: current level;
$G = (V^0, E^0)$: graph representation at level 0;
C^0 community set at level 0 ;
c_u^0: the vertex u's community at level 0;

Output: C: community sets at each level;
Q: modularity at each level

$k \leftarrow 0$;
repeat
  $c_u^k \leftarrow u, u \in V^k$;
  $\Sigma_{tot}^{c_u^k} \leftarrow \sum w_{u,v}, e(u,v) \in E^k$;
  $\Sigma_{in}^{c_u^k} \leftarrow \sum w_{u,v}, \forall c_v^k = c_u^k$ and $e(u,v) \in E^k$;
  // Phase 1
  repeat
    for $u \in V$ **do**
      // Find the best community for vertex u.
      $\hat{c_v^k} \leftarrow \underset{\forall v, \exists e(u,v) \in E^k}{\arg\max} \Delta Q_{u \to c_v^k}$;
      // Update Σ_{tot} and Σ_{in}.
      $\Sigma_{tot}^{\hat{c_v^k}} \leftarrow \Sigma_{tot}^{\hat{c_v^k}} + w(u)$; $\Sigma_{in}^{\hat{c_v^k}} \leftarrow \Sigma_{in}^{\hat{c_v^k}} + w_{u \to \hat{c_v^k}}$;
      $\Sigma_{tot}^{c_u^k} \leftarrow \Sigma_{tot}^{c_u^k} - w(u)$; $\Sigma_{in}^{c_u^k} \leftarrow \Sigma_{in}^{c_u^k} - w_{u \to c_u^k}$;
      // Update the community information.
      $c_u^k \leftarrow \hat{c_v^k}$;
    if *No further improvement can be achieved* **then**
      exit loop;
  // Calculate community set and modularity
  $C^{k+1} \leftarrow \{c_u^k\}, \forall u \in V^k$;
  $Q^{k+1} \leftarrow 0$;
  for $c \in C^{k+1}$ **do**
    $Q^{k+1} \leftarrow Q^{k+1} + \frac{\Sigma_{in}^c}{2m} - (\frac{\Sigma_{tot}^c}{2m})^2$;
  print C^{k+1} and Q^{k+1};
  // Phase 2: Rebuild Graph
  $V^{k+1} \leftarrow C^{k+1}$;
  $E^{k+1} \leftarrow \{e(c_u^k, c_v^k)\}, \exists e(u,v) \in E^k$;
  $w_{c_u^k, c_v^k} \leftarrow \sum w_{u,v}, \forall e(u,v) \in E^k$;
  if *No improvement on the modularity* **then**
    exit loop;
  $k \leftarrow k + 1$;

the new graph (Lines 25–27). Then it repeats these two phases for the next level until the modularity cannot increase anymore.

2.3.3 Challenges for Designing Parallel Louvain Algorithm

The most computationally intensive part in the Louvain algorithm is the calculation of the modularity gain for all vertices (Lines 8–13 in Algorithm 1) and the ensuing reconstruction of next-level supergraph based on new community structures (Lines 25–27 in Algorithm 1). Throughout the modularity calculation and the supergraph reconstruction, the change of vertex connectivity and community structure for each vertex is immediately applied to the same computation for the next vertex. Simply speaking, computation in the algorithm has an widespread serial dependence at every level, from vertex to vertex and iteration to iteration. Thus, the parallelization of the Louvain algorithm on a distributed memory system cannot be achieved by simply partitioning all vertices and distributing the modularity calculation across compute nodes because the change of connectivity and community structure at one vertex is not available to the other vertices.

A parallel version of the Louvain algorithm needs to preserve the same convergence, modularity and community detection properties of the original sequential version. The lack of a global shared memory leads to a number of important challenges for the design of a parallel Louvain algorithm.

The first challenge is the calculation of modularity gain ΔQ for all vertices with dynamically updated community structures. Imaging in a parallel environment, the vertices are spread across many compute nodes. Each vertex needs to examine the modularity gain for joining its neighbors' communities, which requires repetitively gathering all edges to vertices in the corresponding communities ($w_{u \to c_i}$). Such gather operation requires a significant amount of communication and synchronization, and hinders the degree of parallelism.

The second challenge is the convergence property. The sequential algorithm guarantees that the modularity always increases when moving one vertex to a community because of

the greedy policy. However, this is not always true in a parallel environment. Because when vertices compute their ΔQ in parallel, each only sees a static snapshot of their neighbors based on the available pre-existing community structures. The convergence property of the greedy policy is no longer preserved. Vertices may end up exchanging obsolete community structures with little gain on modularity, resulting in the infinite movement of vertices, i.e., not converging.

A solution to these challenges requires not only in-depth examination of inherent computation and communication parallelism in the algorithm, but a novel heuristic that can dynamically control the vertex coverage of the calculation, purge obsolete or non-contributing vertices, and eventually converge with high-modularity communities.

2.4 Summary

In summary, this dissertation seeks to investigate efficient and scalable programming models for fast computation and data processing in the following directions:

- Scalable communication for GAS runtime system;
- Efficient data movement for MapReduce programming model; and
- Scalable parallel community detection over distributed memory systems.

My research during the Ph.D. program has contributed to the following publications:

1. Xinyu Que, Weikuan Yu, Vinod Tipparaju, Jeffrey S. Vetter, Bin Wang. Network-Friendly One-Sided Communication through Multinode Cooperation on Petascale Cray XT5 Systems. CCGRID 2011.
2. Weikuan Yu, Xinyu Que, Vinod Tipparaju, Jeffrey S. Vetter. HiCOO: Hierarchical cooperation for scalable communication in Global Address Space programming models on Cray XT systems. JPDC 2012.

3. Weikuan Yu, Xinyu Que, Vinod Tipparaju, Richard L. Graham, Jeffrey S. Vetter. Cooperative server clustering for a scalable GAS model on petascale cray XT5 systems. Computer Science - R&D 2010.

4. Weikuan Yu, Vinod Tipparaju, Xinyu Que, Jeffrey S. Vetter. Virtual Topologies for Scalable Resource Management and Contention Attenuation in a Global Address Space Model on the Cray XT5. ICPP 2011.

5. Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, Dhiraj Sehgal. Hadoop acceleration through network levitated merge. SC 2011.

6. Xinyu Que, Yandong Wang, Cong Xu, and Weikuan Yu. Hierarchical Merge for Scalable MapReduce. MBDS 2012.

7. Weikuan Yu, Xinyu Que, Yandong Wang, and Cong Xu. Virtual Shuffling for Efficient Data Movement in MapReduce. TC 2013 (under review).

8. Xinyu Que, Fabio Checconiy, Fabrizio Petrini, Teng Wang, and Weikuan Yu. lightning-fast Community Detection in Social Media: A Scalable Implementation of the Louvain Algorithm. SC 2013 (under review).

9. Vinod Tipparaju, Edoardo Apr, Weikuan Yu, Xinyu Que, Jeffrey S. Vetter. Runtime Techniques to Enable a Highly-Scalable Global Address Space Model for Petascale Computing. IJPP 2012.

10. Weikuan Yu, Yandong Wang and Xinyu Que. Design and Evaluation of Network-Levitated Merge for Hadoop Acceleration. TPDS 2013.

11. Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristi Cira, Bin Wang, Zhuo Liu, Bliss Bailey, and Weikuan Yu. Assessing the Performance Impact of High Speed Interconnects on MapReduce Programs. Invited paper to WBDB 2013.

Chapter 3

Related Work

A great deal of effort has been directed toward programming model. In this chapter we first review prior work on the communication runtime. Following that, we describe latest research on MapReduce. In the end, we review the research in parallel community detection.

3.1 Research On Communication Runtime

The scalability of communication runtime involves a number of complicated design issues, including process management, selection of connection models, data communication, communication buffer management, as well as flow control.

Runtime Communication Library: The design and implementation of MPI on Portals was first described by Brightwell *et al.* [37]. This has been one of the reference implementations for other programming models on top of Portals, in which communication protocols for different size messages are elaborated. Huang et al. [38] studied the scalability of communication for MPI on multicore clusters. Sun et al. [39] re-evaluated the impact of Amdahl's law in the multicore aera. Our work investigates the scalability of runtime communication through multinode cooperation. Bonachea *et al.* [40] recently ported GASNet to the Portals communication library on the Cray XT platform to support UPC and other GAS models. Generic issues such as enabling communication operations, handling requests/replies, and flow control were discussed. Global Arrays [41]and, more specifically, its runtime system, ARMCI [9], have been implemented for a while variety of high-performance architectures and interconnects. Nieplocha *et al.* [42] described an efficient implementation for cluster with Myrinet. Nieplocha *et al.* [43] described and evaluated protocols for optimizing communication on the Quadrics QsNetII hight-performance network interconnect, which observed

over 40% improvement in overall communication time. Tipparaju *et al.* [44] evaluated the implementation of ARMCI on the Portals network interface on the Cray XT3 and showed over 80% overlap for all the message sizes tested. Recently Tipparaju *et al.* [10] designed and implemented a scalable ARMCI communication library on Cray XT5 2.3 PetaFLOPs computer at Oak Ridge National Laboratory, and demonstrated its strength in enabling GA and a real world scientific application - NWChem - from small jobs up through 180,000 cores.

Runtime Communication Topology: Topologies for communication networks have been well documented in the textbooks [45, 46]. Exploiting scalable topologies for high performance communication networks has also been studied extensively in the literature, such as those in [47, 48, 49]. Our research represents an innovative use of classic topologies. By imposing mesh and cube topologies on top of small fully connected graphs (FCG), we introduced meshed FCGs (MFCG) and cubic FCGs (CFCG) to formalize challenging issues faced by today's petascale programming models.

Runtime Communication Algorithm: Numerous algorithms were investigated to support deadlock-free message routing in interconnection networks. In their classic paper, Dally et al. [50] proposed deadlock-free message routing algorithms, such as dimension-order routing, for multiprocessor interconnection networks using the concept of virtual channels. Duato et al. [51] investigated deadlock-free adaptive multicast routing algorithms on worm-hole networks using a path-based routing model. Lin and Lionel [52] compared different multicast worm-hole routing algorithms, such as dual-path routing and multi-path routing, for multicomputers with 2D-mesh and hypercube topologies. Our work builds on top of the dimension-order routing algorithm, and proposes the deadlock-free LDF (lowest dimension first) algorithm. LDF only needs to forward an ARMCI request once per dimension in MFCG, CFCG, and Hypercube. In addition, it allows partially populated MFCG and CFCG on any number of network nodes.

Runtime Resource Management: Many efforts studied the scalability of resource management for other contemporary programming models. Sur *et al.* [53] examined the

memory scalability of various MPI implementations on the InfiniBand network. Koop *et al.* [54] exploited the use of message coalescing to reduce the memory requirements for MPI on InfiniBand clusters. Chen *et al.* [55] optimized the communication for UPC applications through a combination of techniques including redundancy elimination, split-phase communication, and communication coalescing. Our work differs from these earlier studies by introducing new virtual topologies to reveal the challenges of resource management and contention in the ARMCI Global Address Space runtime system. To the best of our knowledge, this thesis is the first in literature to exploit the concept of virtual topology for systematic investigation of scalability and contention issues in Global Address Space programming models.

3.2 Research On MapReduce Programming Model

MapReduce is popularized by Google as a very simple but powerful program model that offers parallelized computation, fault-tolerance and distributing data processing [12]. Its open-source implementation, Hadoop, provides a software framework for distributed processing of large datasets [13]. We review related work in a number of directions.

MapReduce Data Communication: Kim et al. [56] improved the performance of MapReduce by reducing redundant I/O in the software architecture. But it did not study the I/O issue caused by the data shuffling between MapTasks and ReduceTasks. The closest work to this research is *MapReduce Online* as proposed by Condie et al. [57]. This work focused on enabling instant shuffling (so called online) of intermediate data from MapTasks to ReduceTasks. Essentially, MapReduce Online introduces direct data shuffling channels between MapTasks and ReduceTasks to avoid the creation of intermediate MOF files. In doing so, it requires the direct coupling of each MapTask with all ReduceTasks, and completely changes the fault handling mechanism of Hadoop. A failure of a MapTask or a ReduceTask is no longer a local event that can easily be recovered by re-launching the failed task. In addition, MapReduce Online requires a large number of TCP connections, which limits its

scalability. Our work does not require close coupling of data flow between MapTasks and ReduceTasks, allowing separated recovery from failures of either MapTasks or ReduceTasks. We propose virtual shuffling as a new strategy to enable data shuffling on-demand instead of physically moving, merging and storing data before they are processed by the reduce function.

Power and Energy of MapReduce Programs: Leverich et al. [58] modified Hadoop to allow scale-down of operational clusters which could save between 9% and 50% of energy consumption. They also outlined further research into the energy-efficiency of Hadoop. Lang et al. [59] closely examined two techniques, namely Covering Set (CS) and All-In Strategy (AIS), which could be used for the management of MapReduce clusters. They showed that AIS was the right strategy for energy conservations. Chen et al. [60] presented a statistics-driven workload generation framework which distilled summary statistics from production MapReduce traces and realistically reproduced representative workloads. This methodology could be useful for understanding design trade-offs in MapReduce. The same team also exploited and analyzed how compression could improve performance and energy efficiency for MapReduce workloads [61]. They proposed an algorithm which examines per-job data characteristics and I/O patterns, and decides when and where to use compression. Our work does not directly study energy conservation techniques, but evaluates the benefits of virtual shuffling in energy savings. This is complementary to previous research efforts. Our work documents a case study in conserving energy by reducing other related system activities such as disk access.

3.3 Research On Parallel Community Detection

It is nontrivial to find a solution for community detection problem. a superior approach always manifests itself in its three key features: scalability, accuracy, efficiency. Towards this end, Many algorithms have been attempted. however, only a handful of these methods consider parallelism, which is an indispensable characteristic in maintaining high efficiency.

These methods generally fall into two categories: algorithm based on shared memory and algorithm built upon distributed memory. With respect to the former, Riedy et al. [62] proposed a parallel algorithm by partitioning a graph into subgraphs and merging the intermediate subgraphs towards better graph property. following a principle that maximize the modularity or minimize the conductance. It is laudable in that the parallelism through shared memory enables fast computation, however, This solution was confined by specific hardware, precluding the possibility to be ported to other hardware platforms. Martelot et al. [63] came up with an multi-threaded algorithm for fast community detection with a local criterion that dispatches the work to multiple threads. This algorithm was efficient and fast, however, it did not have good accuracy. Bhowmic et al. [64] recently introduced a shared-memory implementation of Louvain method. Their work achieved modularity comparable to the sequential algorithm. This approach could scale to only a small number of threads, due to the built-in nature of shared memory.

Zhang et al. [65] proposed an algorithm that was built upon the mutual update between network topology and topology based propinquity. Their work generated community through a self-organizing process that distributes the workload among thousands of machines. But this algorithm contained too many synchronizations among different participating processes and could not scale. Yang et al. [66] implemented a parallel community detection scheme on top of the MapReduce [67] framework, which is based on maximal cliques. However this approach suffered from the runtime complexity and led to poor performance. Soman et al. [68] implemented a parallel algorithm based on the label propagation algorithm [69] on GPGPUs. Label propagation algorithms are fast but often does not deliver a unique solution, which requires further investigation on the community quality. In addition, algorithms fail to unfold the hierarchical organization, which is an important feature [70] displayed by most networked systems in the real world [15].

Furthermore, Soman et al. [68] implemented a community detection algorithm optimized for GPU architectures based on label propagation algorithm [69], Their technique could scale

with a number of cores/threads and delivered high modularity for large sized RMAT graphs. It makes decisions on community structure on top of local topological information, with linear time complexity. This algorithm performs well in that the label propagation technique inherently contains fine-grained parallelism and minimal synchronization, which aptly fall into the category of GPGPU based solution and achieves a higher speedup. However, Due to the inherent limitation of label propagation, this algorithm could not always deliver a unique solution, necessitating further investigation on the community quality.

Chapter 4

Design and Implementation

In this chapter We first present the design of the Hierarchical Cooperation which extends ARMCI with supporting indirect one-sided communication. Two key components multinode cooperation [71, 72] and virtual topology [73] will be described to show how they are able to address the challenges in the current runtime systems. Following that, We introduce virtual shuffling strategy for MapReduce programming model. The implementation of three-level segment table, on-demand merging, and dynamic and balanced merging subtrees will be explained in detail. In the end, we present our novel design on scalable parallel community detection algorithm for massive graphs over distributed memory systems.

4.1 Hierarchical Cooperation (HiCOO) for Scalable GAS Runtime System

Figure 4.1 shows the software architecture of Hierarchical Cooperation (HiCOO). On a system that supports GAS-enabled scientific applications such as NWChem, ARMCI will support the required one-sided operations, including data transfer, atomic and locks, memory management, and synchronization. HiCOO extends ARMCI with an indirect communication model for transmitting one-sided requests in these operations. It includes two key components: multinode cooperation and virtual topology. These two components are mutually dependent on each other for their functionalities. Multinode cooperation offers the fundamental communication mechanisms for different nodes and their communication servers to cooperate with each other for request handling. Virtual topology offers a formal model that defines the geometric relationship among all the nodes, and accordingly their distance in the topology hierarchy.

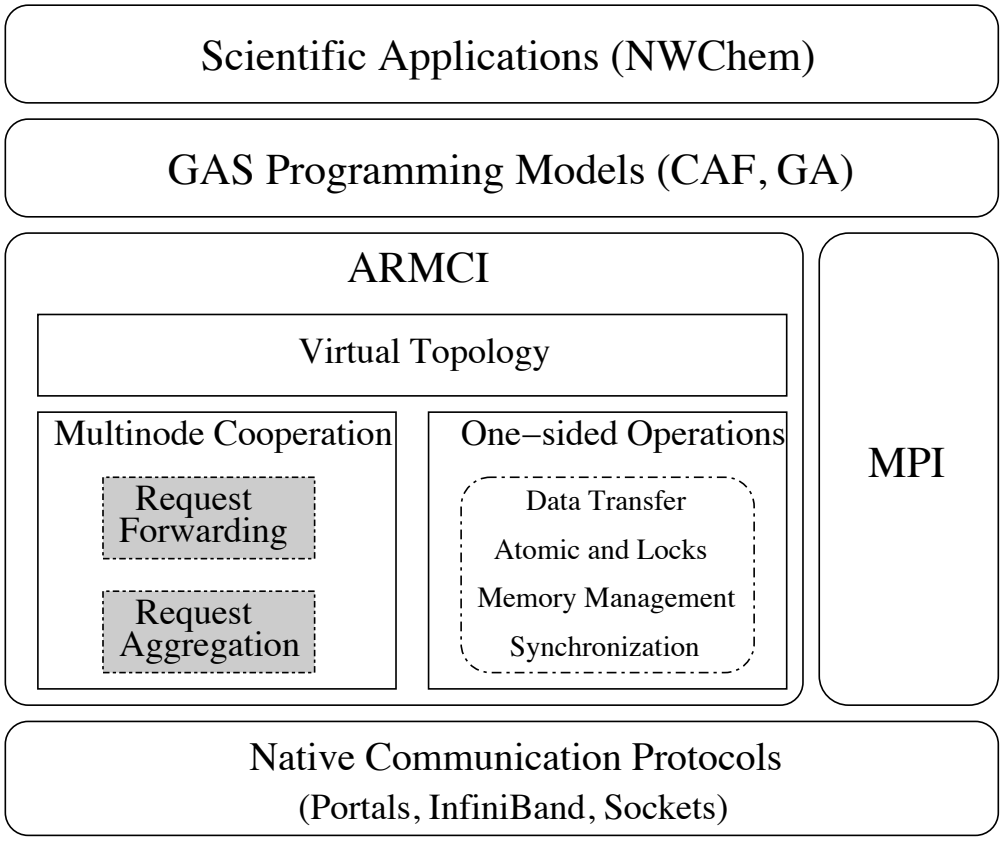


Figure 4.1: Software Architecture of Hierarchical Cooperation

4.1.1 Multinode Cooperation

Multinode cooperation is intended to address the scalability challenge of communication buffers, as well as the associated network contention, caused by one-sided messages in ARMCI's original direct communication model. It is supported through two communication mechanisms: request forwarding and request aggregation. We will focus on describing these two mechanisms in more detail.

Multinode cooperation fundamentally addresses the scalability issues of direct one-sided request messages. Instead of allocating one set of buffers for all remote processes on each node, multiple nodes form a cooperative multinode group to allocate buffers. Communication servers on these nodes divide incoming requests from outside processes amongst themselves. For example, for a program with N processes, one communication server roughly has to

preallocate $N-1$ sets of communication buffers in the original ARMCI. When a K-node group is formed through multinode cooperation, one communication server will only need to preallocate $(N-1)/K$ sets of communication buffers. Because of the division of requests among servers, a multinode group effectively reduces each server's communication buffer requirement by the size of the multinode group. The servers in a multinode group then cooperate and handle one-sided requests from processes outside the group. When one request reaches any server in the multinode group, it will be forwarded to the actual target server.

With multinode cooperation, most of one-sided communication requests are no longer sent directly to the destination communication server. This brings in another beneficial feature. The risk of network contention caused by many requests to a single hot-spot target node is significantly alleviated, because requests are first buffered by cooperative nodes in a multinode group, and aggregated if they arrive closely with each other in time. Request aggregation is described in more detail below.

The original ARMCI has a very simple communication model to support direct one-sided operations. Figure 4.2(a) shows the flow of request and reply between a pair of processes (P_r and P_t). The communication server CS_T (co-located with P_t) receives the request from P_r on behalf of P_t. As the requested operation completes, CS_T returns a corresponding reply or acknowledgment (ack/rep) to P_r. This forms a direct request/reply pair and a simplified flow control scheme between P_r and CS_T.

The key of multinode cooperation is its indirect request communication model. This is achieved through request forwarding and request aggregation. Figure 4.2(b) shows the flow of requests and replies in multinode cooperation. Three processes (P_{r0}, P_{r1}, and P_{r2}) are initiating three one-sided requests (R0, R1, and R2) to a target process (P_t), through the communication server (CS_I) at the same intermediate node. CS_I receives these requests, and detects that they are targeting for the same communication server CS_T. So these requests are aggregated together into a single request and sent to CS_T. Only one acknowledgment is

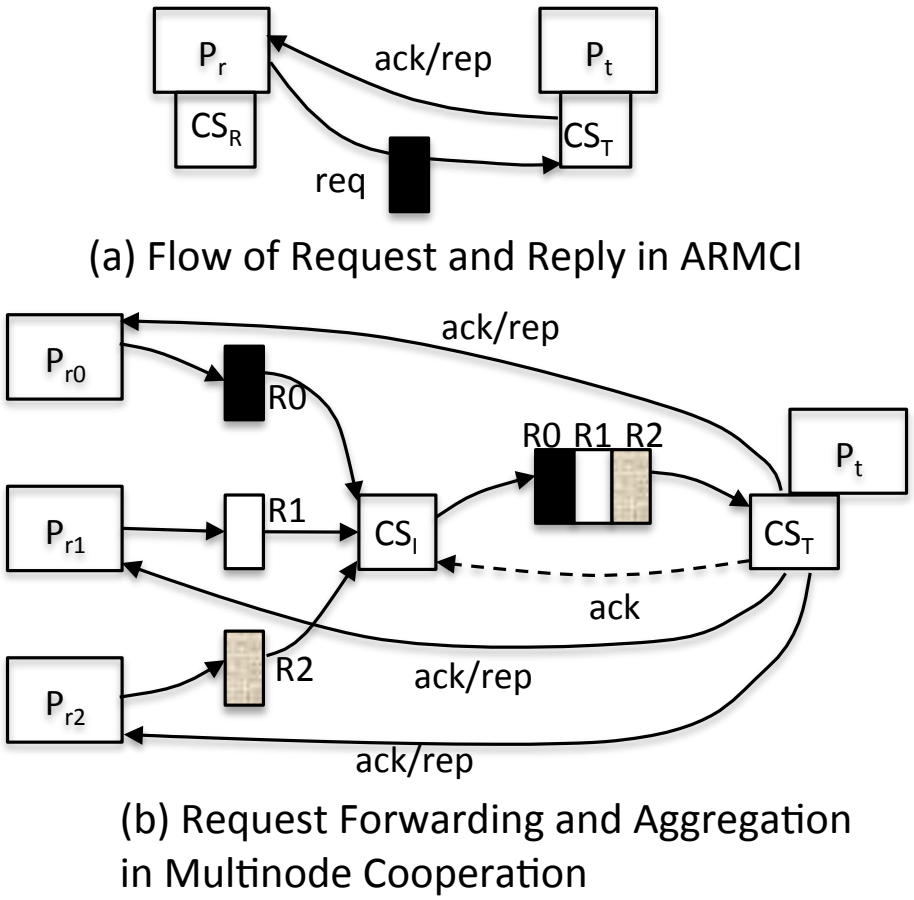


Figure 4.2: Request Handling in ARMCI and Multinode Cooperation

needed for the aggregation request. CS_T receives a combined request, and processes the embedded requests separately. In the end, it sends back individual replies or acknowledgments back to three requesting processes.

Request forwarding can be viewed as a special case of the same diagram, where requests are not allowed to be aggregated together. When a request arrives at CS_I, it is immediately forwarded to CS_T. There must be a separate acknowledgment for every request message.

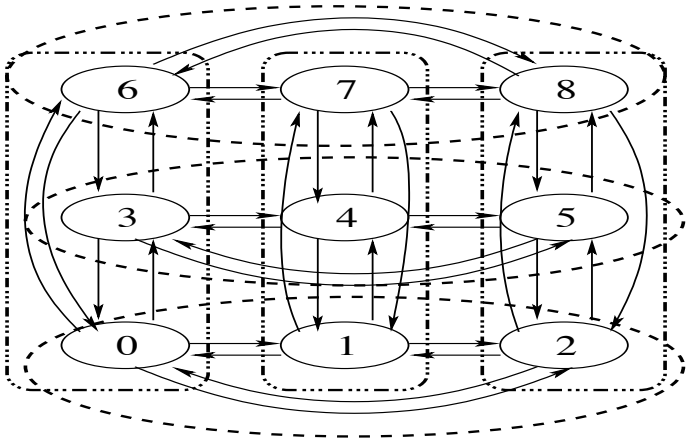
Event-Driven Aggregation Window – To allow request aggregation, a communication server must hold on to one request and wait for the arrival of more requests. When requests arrive closely within each other, there are plenty of opportunities to aggregate requests. However, the communication server should not keep a request for too long when no

more requests arrive in time. On the other hand, the communication server cannot busy wait for the arrival of new requests, which would consume a lot of CPU cycles. We address this issue through an event-driven aggregation window. Upon the arrival of a new request, the communication server records its timestamp. It is then blocked, waiting for the arrival of more requests. Every portals message generates an event on the communication server, and wakes up the communication server to perform possible request aggregation. A request will be forwarded when the aggregation window expires. An extra event is introduced to wake up the communication server when no portals messages are communicated. Within a multinode group, an empty message is periodically initiated by a communication server to its peer. This message will generate an event to wake up blocked communication servers, thereby breaking a potential stalemate caused by a held request.

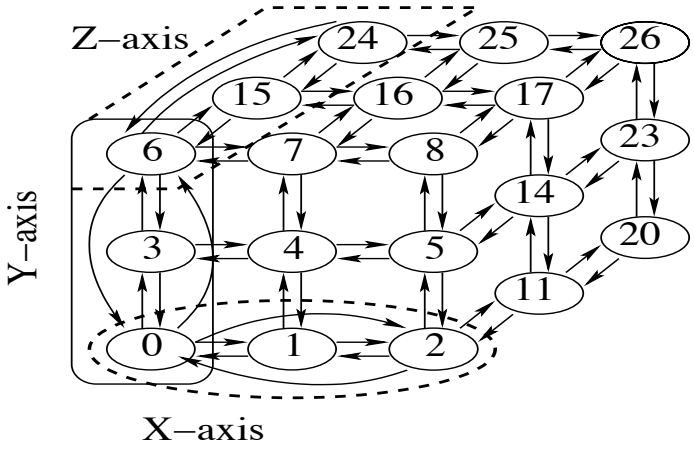
4.1.2 Virtual Topology

As discussed in Section 2.1, the default resource allocation in ARMCI leads to a serious scalability challenge. More importantly, its resource dependence relationship (irrespective of any underlying physical network topology) can cause contention when some processes become hot-spots to the communication requests. A virtual topology FCG can precisely reflect the state of resource allocation and contention. It also suggests that alternative virtual topologies may offer a solution for scalable resource management and contention attenuation. We first introduce two new virtual topologies: MFCG and CFCG, and examine various features of these two, along with a canonical topology Hypercube. Then we describe the details of request routing in realizing these topologies.

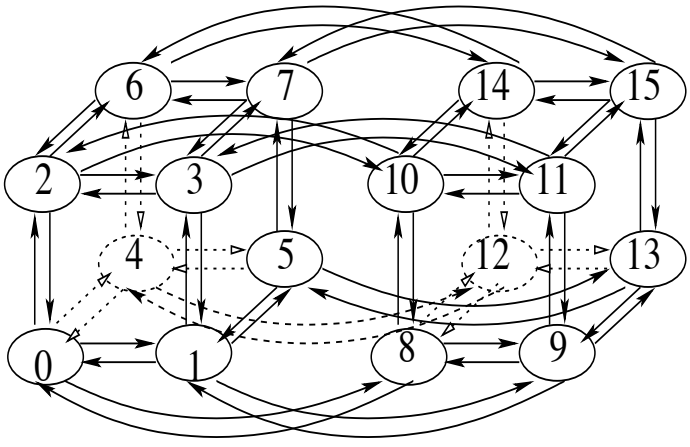
4.1.2.1 Comparisons of Three Virtual Topologies



(a) Meshed FCGs



(b) Cubic FCGs



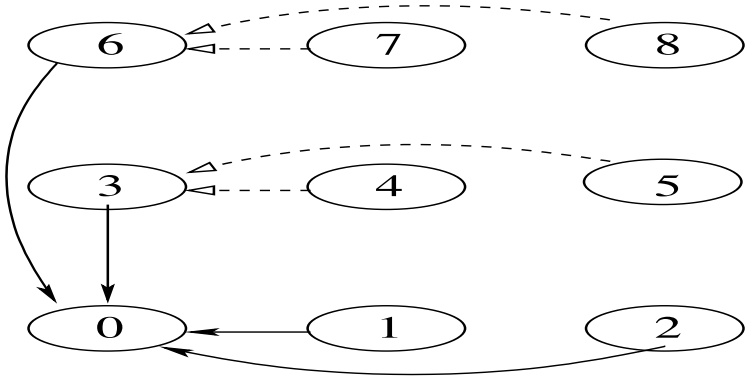
(c) Hypercube

Figure 4.3: Three Virtual Topologies (For clarity, not all vertices/edges are shown in CFCG)

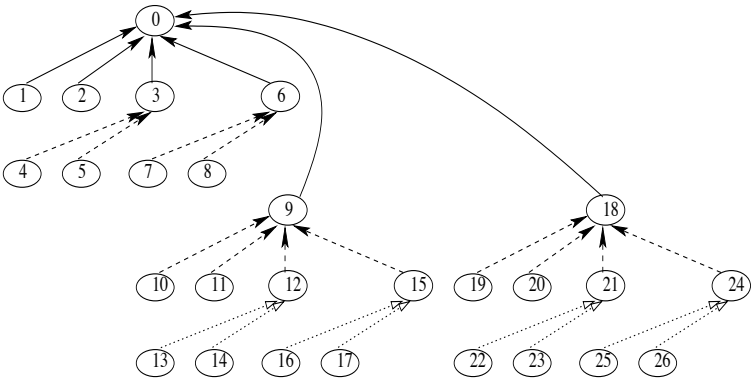
MFCG – The first virtual topology we have introduced is called Meshed Fully Connected Graphs (MFCG for short). Figure 4.3 (a) shows an example of MFCG, in which all nodes are virtualized as vertices in a $X \times Y$ mesh (in this case, $X = 3$ and $Y = 3$). Nodes with the same Y-offset are fully connected. That is to say, they all dedicate request buffers to each other. The same policy is applied to nodes with the same X-offset. Thus, for an arbitrary $X \times Y$ MFCG, an individual node has $(X - 1)$ outgoing edges on X-dimension and $(Y - 1)$ outgoing edges on Y-dimension. A request routing mechanism is provided to exchange requests between a pair of nodes that are not directly connected. Therefore, using MFCG, the number of request buffers on each node decreases to $O(\sqrt{N})$, instead of $O(N)$ in FCG.

MFCG is also beneficial in alleviating contention. Figure 4.4 (a) shows request paths for nodes in a 3×3 MFCG to reach Node 0. Two types of request paths are possible: the first type is used by the nodes that are directly connected to Node 0; and the second type is used by the nodes that are not directly connected. These paths form a tree of height 2 and rooted at Node 0. Compared to the flat tree as shown in Figure 2.4, the contention is reduced to $O(\sqrt{N})$. One may rightfully argue that contention as depicted in Figure 4.4(a) does not reflect the actual contention in the physical network. The purpose of scalable virtual topology is to offer a convenient tool that can cope with network contention at a software level, instead of leaving the contention issues completely to the network hardware.

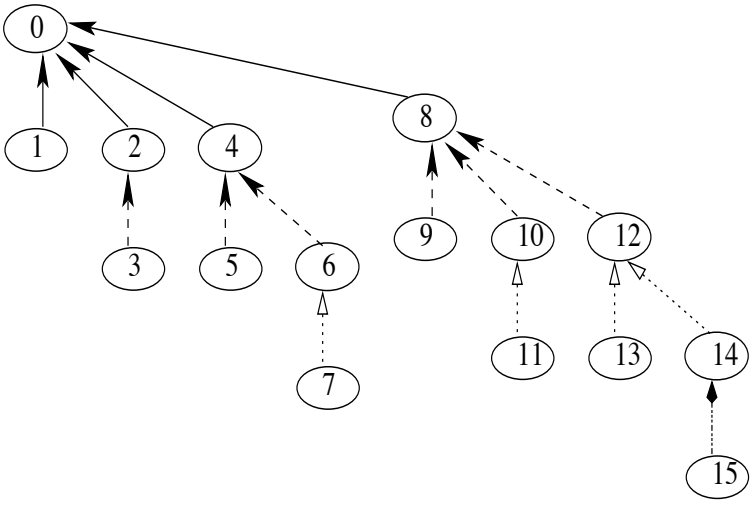
CFCG – Another virtual topology we introduced is Cubic Fully Connected Graphs (CFCG). Figure 4.3 (b) shows an example of CFCG, in which all nodes are virtualized as vertices in a $X \times Y \times Z$ cube (in this case, $X = 3$, $Y = 3$, and $Z = 3$). The nodes with the same offsets on two dimensions are fully connected as an FCG. For an arbitrary $X \times Y \times Z$ CFCG, an individual node have $(X - 1)$ outgoing edges on X-dimension, $(Y - 1)$ outgoing edges on Y-dimension, and $(Z - 1)$ outgoing edges on Z-dimension (to clarify, not all vertices/edges are shown for CFCG). Using CFCG, the number of request buffers on one node scales in the order of $O(\sqrt[3]{N})$, instead of $O(N)$ with FCG. A request may have to be



(a) Tree for MFCG



(b) Trinomial (or K-nomial) Tree for CFCG



(c) Binomial Tree for Hypercube

Figure 4.4: Tree Representations of Request Paths in Virtual Topologies

forwarded maximally two times before reaching its destination.

Figure 4.4(b) shows the tree representation of request paths for nodes in a $3 \times 3 \times 3$ CFCG to reach Node 0. These directed paths form a trinomial tree of height 3 and rooted at Node 0. For a system with N nodes, the tree of request paths rooted at an arbitrary node will be k-nomial tree where k = $\sqrt[3]{N}$. Compared to the flat tree in Figure 2.4, network contention is then reduced by an order of $O(\sqrt[3]{N})$, at the expense of up to 2 forwarding steps to deliver a request.

Hypercube – As discussed above, CFCG is more scalable in resource allocation than MFCG and FCG, despite more steps for request transmission. One may wonder if a virtual topology of even higher dimension could be a worthy solution. So we investigate the third virtual topology, Hypercube. Figure 4.3(c) shows 16 nodes that are connected as a Hypercube. Each node is directly connected to $\log_2 N$ nodes (4 in this case). Figure 4.4(c) provides a tree representation of request paths from all nodes to Node 0. For N nodes, it is essentially a binomial tree of depth $\log_2 N$. Using Hypercube, the number of request buffers required on one node scales in the order of $O(\log_2 N)$. Two nodes may be separated by up to $\log_2 N$ dimensions apart. Therefore, up to $(\log_2 N - 1)$ transmissions are needed for a request to reach its destination. On the other hand, at each depth of a request path tree, contention is reduced by an order of $O(\log_2 N)$.

4.1.2.2 Request Routing in Virtual Topologies

We have implemented MFCG, CFCG, and Hypercube in ARMCI on Jaguar. The support for request routing is the key to realizing these virtual topologies. Communication servers on intermediate nodes are used to transmit a request from the original process to the target server. Upon the arrival of a request, the target sends a response (or acknowledgment) directly to the original process. If an intermediate server (or the target) detects that the request is routed from an upstream server, it sends an acknowledgment to the upstream server. To support multidimensional topologies such as MFCG, CFCG, and Hypercube, our implementation also allows a request to be transmitted multiple steps.

For correct request routing, the actual implementation of virtual topologies requires proper handling of two important issues: (a) how to determine the order of routing; and (b) how to enable virtual topologies, MFCG and CFCG, when the number of nodes can only be configured as partially-populated topologies (mesh or cube), e.g., a prime number that cannot be evenly divided. As mentioned earlier, we include Hypercube only to examine its tradeoff in resource management and contention, compared to MFCG and CFCG. For the investigative purpose, we only support hypercube when the number of nodes is a power of 2.

Lowest-Dimension-First Routing – Multiple communication steps are needed for an ARMCI request to properly reach its destination, in multi-dimensional virtual topologies such as MFCG, CFCG and Hypercube. Each step corresponds to a relationship in which an upstream node is dependent on the availability of request buffer at the downstream node. If the routing of requests were to happen arbitrarily, it would create cyclic dependences and lead to deadlocks in a multi-dimensional virtual topology.

Algorithm 1 Lowest Dimension First Routing

1: {Dimension: k}
2: {Current Node: $S = (s_0, s_1, ..., s_{k-1})$}
3: {Destination Node: $T = (t_0, t_1, ..., t_{k-1})$}
4: $D \leftarrow S$ {Initialize D as the next node}
5: $i \leftarrow 0$
6: **while** $(D \neq T)$ **do**
7: **if** $s_i \neq t_i$ **then**
8: $D \leftarrow (s_0, s_1, ..., s_{i-1}, t_i, s_{i+1}, ..., s_{k-1})$
9: {Forward the request to the next node, D}
10: **end if**
11: $i \leftarrow i + 1$
12: **end while**

We develop a lowest-dimension-first (LDF) protocol to ensure deadlock-free routing in virtual topologies. Algorithm 1 illustrates the selection of next node for request routing in LDF. For two nodes $S = (s_0, s_1, ..., s_{k-1})$ and $T = (t_0, t_1, ..., t_{k-1})$ on a virtual topology with k dimensions, LDF always chooses the lowest dimension i on which S and T differ. A request

is then forwarded to the next destination D, which is a number derived by replacing s_i of S with t_i. Since the order of routing is established in an monotonic dimension order, breaking any cyclic dependence. Therefore LDF is deadlock-free. When the number of nodes allows virtual topologies to be fully populated as meshes, cubes, or hypercubes, LDF as shown in Algorithm 1 works perfectly.

Routing on Virtual Topologies with Any Number of Nodes – Routing in a virtual topology is similar to routing in a physical interconnect. In the case of a fully populated two-dimensional MFCG, LDF can be reduced to the classic turn model [74] that was designed for 2-D meshes. However, the key difference is that a virtual topology is very dynamic and frequently partially populated. For this reason, each node frequently changes its position from one topology to another. It is important that deadlock-free routing be enabled on virtual topologies (MFCG and CFCG) with any number of nodes.

We achieve that by strictly ordering all nodes in a lowest dimension first manner. For a virtual topology G with dimension k, the lower order dimensions are first populated with available nodes. Only the highest dimension, $k-1$, is allowed to be partially populated. Assume that a virtual topology G has M as its highest ranked node, where $M = (M_0, M_1, ..., M_{k-1})$. With all nodes ordered this way, we extend the LDF algorithm slightly. It allows routing only when the next destination D is a number smaller than or equal to M. An extra condition, "if $(D \leq M)$", is introduced to Algorithm 1 before a request is forwarded. With this extension, if routing paths of a set of requests did not violate this extra condition, there would not be a deadlock because their routing paths are determined by Algorithm 1. For a possible deadlock to occur, one request must have violated this condition once in its path. This is not possible because the nodes are strictly ordered and no node can have a rank higher than M (by definition). Therefore, it prevents any circle in request routing. The listing of the extended LDF algorithm is not included here, due to the simplicity of this addition.

4.2 Virtual Shuffling for Efficient Data Movement in MapReduce

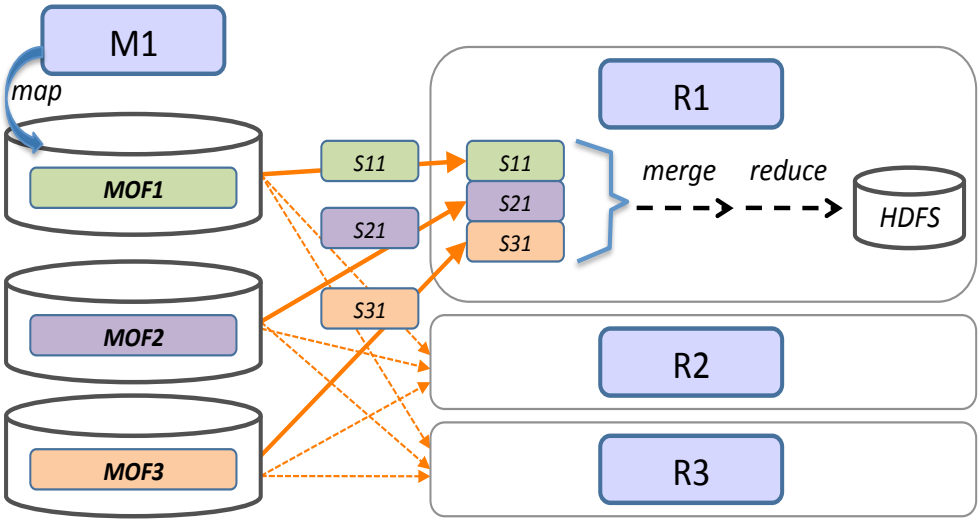
As discussed in Section 2.2, the default Hadoop MapReduce programming model involves a large number of I/O accesses in three phases, especially for data-intensive application, which severely hurts the performance. In this dissertation, I undertake a different effort to investigate the issue of disk I/O contention in data shuffling phase of MapReduce programming model. As shown in Figure 4.5, we take a new perspective at data shuffling of MapReduce programs. In the default Hadoop implementation, intermediate data segments are pulled by ReduceTasks in their entirety to local disks, and then merged before being reduced for final results. This is shown by Figure 4.5(a). Because the physical movement of segments across disks, we refer to this strategy as physical shuffling.

Inspired by the classic concepts of virtual memory and demand paging, we propose a virtual shuffling strategy to enable efficient data movement for MapReduce programs. Figure 4.5(b) shows the general idea. Instead of moving data segments to local disks before starting the reduce function, virtual shuffling allows a ReduceTask to fetch only a minimal set of segment attributes and create a virtual segment table that records the actual locations of remote segments. Virtual shuffling delays the actual movement of data until the ReduceTask requests data to be reduced. At that point, virtual shuffling employs on-demand merging to fetch data in small blocks into memory, merge and send them directly to the reduce function. In doing so, virtual shuffling greatly reduces the number of disk accesses of physical shuffling, and enables efficient data movement. In order to overcome the disk I/O problem of physical shuffling, virtual shuffling needs to address three important issues:

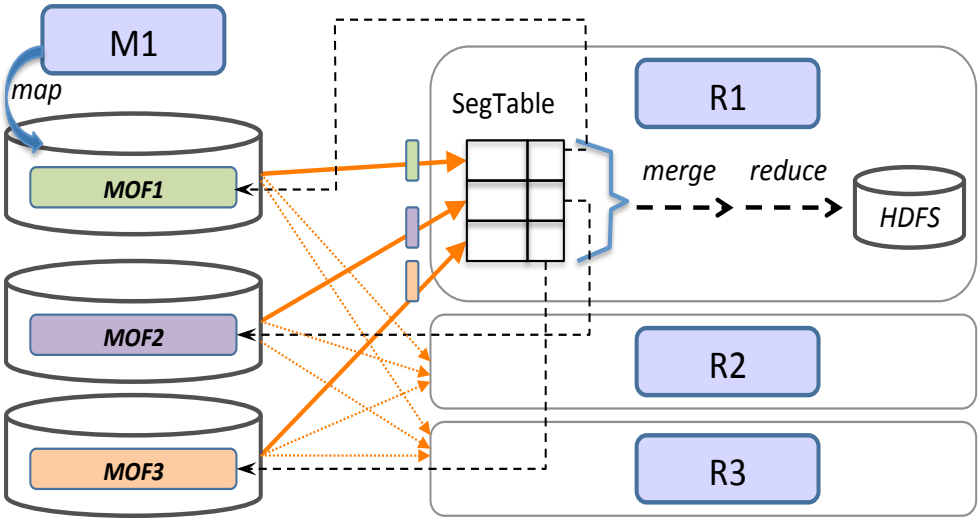
1. How to scalably represent intermediate data segments in a virtual manner?

2. How to minimize the impact of actual shuffling of data?

3. How to dynamically coordinate and balance data shuffling and merging without degrading the performance?



(a) Physical Shuffling



(b) Virtual Shuffling

Figure 4.5: Comparisons of Different Shuffling Strategies

4.2.1 A Three-Level Segment Table

We draw our inspiration from the classic concept of virtual memory in designing virtual shuffling. To manage many intermediate data segments produced by MapTasks, we design a three-level table to organize them in a scalable manner. As shown in Figure 4.6, at the completion of a MapTask, its data segment is not physically copied for merging at a ReduceTask. Instead, a Segment Table Entry (STE) is created at the lowest level–Segment

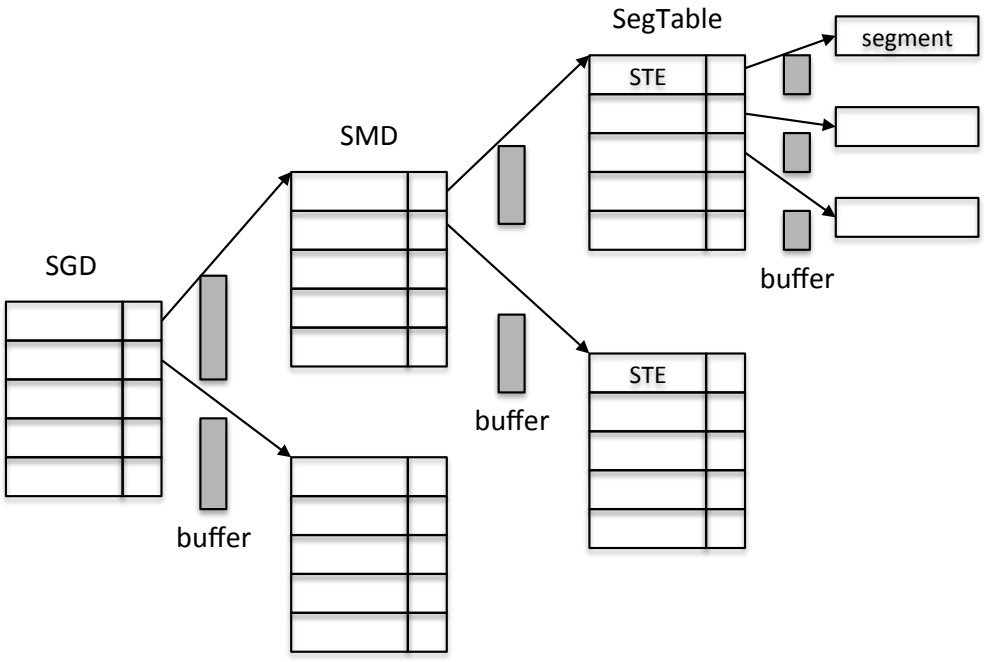


Figure 4.6: Design of A Three-Level Segment Table for Virtual Shuffling

Table (SegTable)–to represent the segment in a virtual manner. The STE includes several attributes of the segment such as its first <key, val> pair, its total length, its source MapTask, as well as its physical location on the remote disk. The number of STEs in a SegTable is a tunable parameter based on the computation, memory, and I/O resources. Many SegTables are organized into a Segment Middle Directory (SMD), in which each entry represents a SegTable. Many SMDs in turn are organized as a Segment Global Directory (SGD).

In addition, three kinds of memory buffers are used as interfaces across different levels of entries. For example, an SGD will interface with its SMDs through a Segment Global Buffer (SGB), an SMD with its SegTables through a Segment Middle Buffer (SMB), and a SegTable with its segments through a Segment Table Buffer (STB). With this three-level hierarchical table, if there were memory pressure, we can keep only a few active SegTables and their ancestral SMDs and SGDs in memory, while other SegTables are temporarily stored on disk.

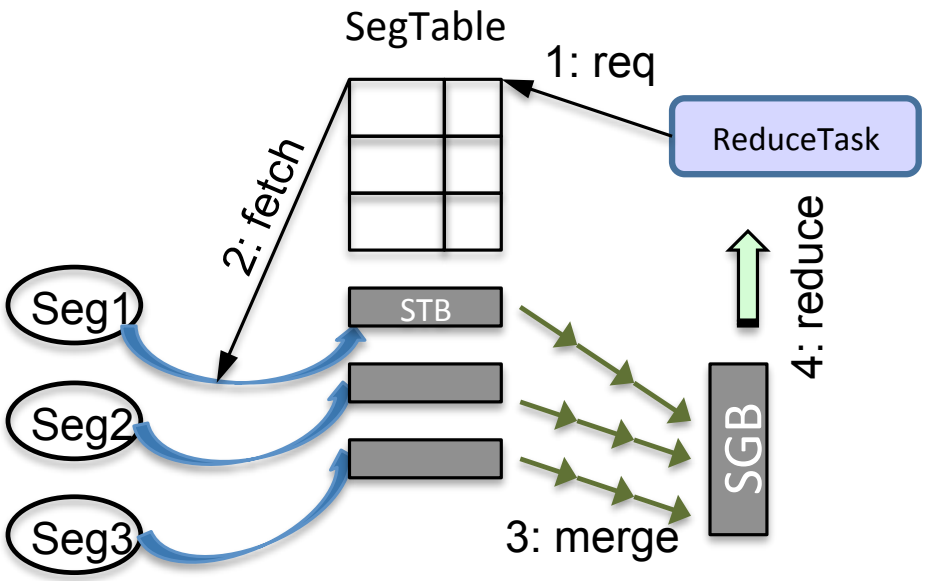


Figure 4.7: On-Demand Merging

4.2.2 On-Demand Merging

We design virtual shuffling not to eliminate data movement, but to hide its cost within the reduce phase of MapReduce. To this end, virtual shuffling mimics the concept of demand paging and realizes on-demand merging to minimize the impact of actual data movement. Figure 4.7 shows the operation of on-demand merging. When the ReduceTask needs to reduce some data, it initiates a data request to the segment table, which in turn triggers the fetching of data blocks (which contain more intermediate <key,val> pairs from MapTasks) from remote segments. These blocks will then be buffered at the SEBs. Based on the virtual segment table, these <key,val> pairs in SEBs will then be merged through segment buffers such as STBs and SMBs, and finally into SGBs. The data in SGBs are available to be reduced. To avoid synchronously waiting on the completion of these steps, two sets of buffers are provided at each interface. This enables double buffering and overlaps the on-demand merging of incoming data with the reducing of previous data. Data from each segment are brought in sequentially as small blocks. One block will be fetched into an SEB only when it is the next block to be merged. On-demand merging is built on top of our previous work

network-levitated merging [75]. While network-levitated merging strives to lift data merging up above disks, on-demand merging emphasizes the importance of hiding and minimizing the cost of shuffling to the reduce phase. On-demand merging does not preclude the need of flushing data to disks, as will be described in Section 4.2.3.

4.2.3 Dynamic and Balanced Subtrees

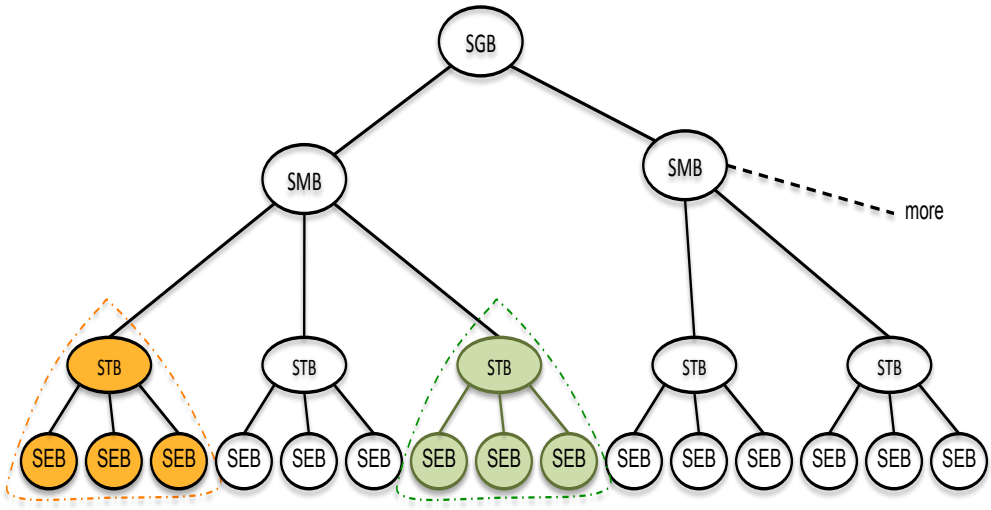


Figure 4.8: Dynamic and Balanced Subtrees for Concurrent Merging

With a hierarchical segment table, all virtual segments are essentially organized into a merging tree in which the leaves are the SEBs. If on-demand merging with double buffering were to activate all leaves, there would be a need of $2N$ SEBs, where N is the number of total virtual segments. For an application with a dataset (S) and a data split size (B), it will then have $N = \frac{S}{B}$ segments. Assume a split size of 64 MB, SEB of 32 KB, and the use of double buffering for blocks from each segment, the amount of memory needed for all SEBs would be $\frac{S}{1024}$ Bytes per ReduceTask, i.e., 1 GB memory for a MapReduce application with 1 terabyte of data. Clearly, this does not allow good scalability for applications with petascale data and beyond.

We employ a dynamic orchestration mechanism to manage the merging of virtual segments. Instead of activating all leaves, we organize the whole tree as many subtrees, each

composed of a SegTable and its SEBs. At any time, only a limited number of subtrees are actively merging data. As shown in Figure 4.8, two subtrees are currently active in merging its SEBs into STBs. The merged data will be further merged to SMBs and/or SGBs. To balance the merging progress at different subtrees, previously active subtrees will be deactivated to allow other subtrees to make progress.

There is an intriguing issue here. At the time when a subtree is to be deactivated, their SEBs usually contain data that are not yet merged to the STB. Worse yet, the use of double buffering means that, for any segment, one SEB has data left to be merged while the other SEB is waiting on data to be fetched remotely. A decision needs to be made on either dumping the data including the remaining data in one SEB and the data in flight for the other SEB or flushing the data to the disk, to make memory available for other subtrees. To improve the utilization of data in memory, we prevent a subtree from fetching more data into SEBs when its STB is already 70% full, and also provide a grace period of 0.5 second (a configurable parameter) in the deactivation of a subtree, allowing more data in SEBs to be consumed (merged) into the STB. For a subtree that still has data left in its SEB, we by default dump the data. A user option is also allowed to flush the data to disk. The reason for dumping data by default is to avoid frequently writing small data blocks to, and reading them back from, disks.

4.2.4 Hierarchical Merge Design Issues

Virtual shuffling is applied to MapReduce through hierarchical merge technique [76] using a two-level hierarchy of priority queues as shown in Figure 4.9. At the very bottom, a linear array (called *treeset*) is used to sort the incoming segments based on their size. Once the number of segments goes over a threshold, the segments are moved into a Child Priority Queue (CPQ). More segments will lead to the creation of more CPQs. After all segments have arrived, the remaining segments in *treeset* are moved to the last CPQ. All CPQs are then organized into a root priority queue (RPQ), which merges data from CPQs into an

additional staging buffer. The segments are spread into many small CPQs. The novelty of Hierarchical Merge is to minimize the number of merges down to 2 and yet keep the merging process in memory.

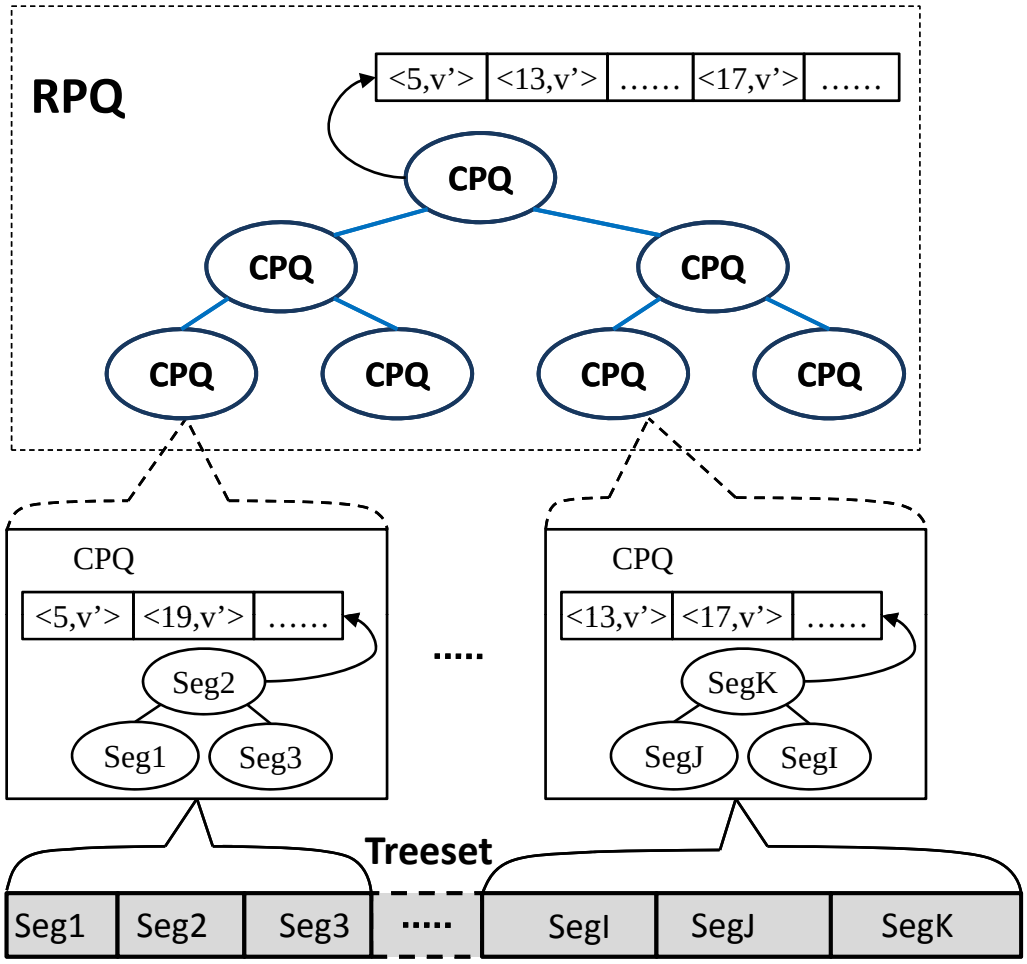


Figure 4.9: Hierarchical Merge Algorithm

There are several issues require further investigation. Since the Hierarchical Merge involves extra merging steps and might cause overhead, it is critical to manage the merging process of RPQ and CPQs smoothly. In addition, limited buffers are shared by all the segments, it is also important to manage buffers efficiently during the assignment and eviction.

4.2.4.1 Merging Orchestration

The merging of the RPQ and that of CPQs are overlapped through the use of multiple worker threads and memory double buffering. Because of this, the merging process of RPQ and CPQ can be conducted asynchronously on different temporary buffers through different workers. One thread, namely the primary worker, is dedicated to the RPQ and several other threads, named secondary workers, handle all the CPQs which depends on the available buffers. To balance the workload among all the secondary workers, our implementation equally assigns CPQs among them, which also guarantees fairness among them.

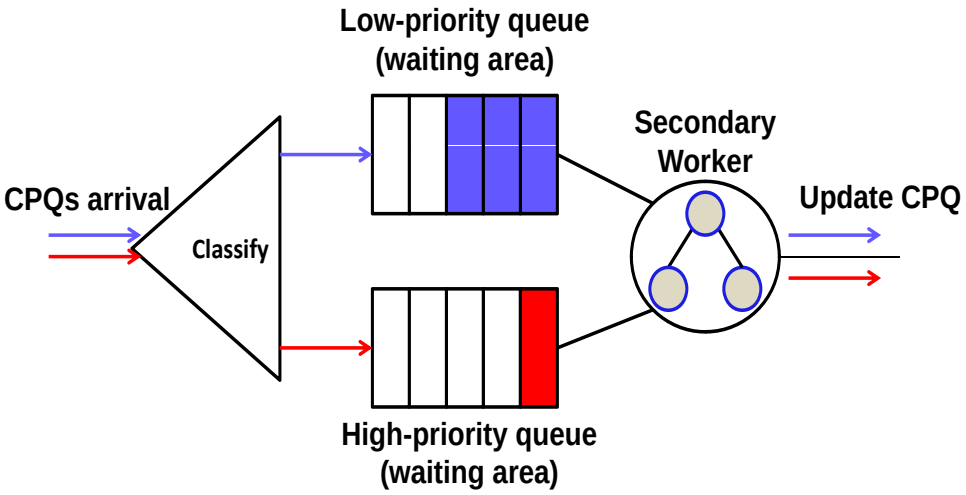


Figure 4.10: Priority-based Scheduling Policy

In order to pipeline the merging at different levels, a scheduler is needed to minimize the stall, which could be caused by the unavailability of the intermediate merged result. We apply *Priority-based Scheduling* to fulfill this requirement. Figure 4.10 shows the scheduling policy. Each secondary worker has two waiting queues for the CPQs belong to it. One queue is for high-priority CPQs, which require to be processed immediately, otherwise the primary worker will be blocked. The other queue is for low-priority CPQs, which can be delayed. Normally, the low-priority queue of a secondary worker is not empty and it contains several CPQs. The secondary worker merges the CPQs from the low-priority queue in a First In

First Out (FIFO) manner and updates them accordingly. CPQs are normally inserted into the corresponding low-priority queue. Only when the merging of the RPQ is blocked and waiting for the update of a specific CPQ, this CPQ has the highest priority and will be insert into the high-priority queue. If such scenario occurs, the corresponding secondary worker will be interrupted (if it is working on other CPQ) to process and update that CPQ with the highest priority, which eventually enables the merging of RPQ and resumes the pipeline.

4.2.4.2 Buffer Management

Hierarchical Merge improves the scalability of Hadoop-A by using less memory. Our buffer manager is able to efficiently manage these limited resource while maintaining the performance improvement. A Buffer Manager is designed to be responsible for the buffer assignment. When a segment is created, two buffers are allocated to it. One is for fetching, called FetchingBuffer, the other one is for merging, named MergingBuffer. When a CPQ is partially merged, all of the buffers hold by its segments should be returned to the Buffer Manager so that other CPQs can have their fair share of the resources. However, simply returning the buffers may cause waste of unused data which shall be avoided to the maximum degree. So, instead, the Buffer Manager records the addresses of the available buffers, leaves them within the CPQs and only evicts the buffer from the CPQ on demand.

In addition, a naive strategy to assign a random buffer to a CPQ can also be problematic. For example, when both the fetching and merging buffers become available, if the MergingBuffer is firstly evicted, the merging thread has to stall until the data in the MergingBuffer is restored when the CPQ is activated for merging again. On the contrary, if the FetchingBuffer is evicted, then the intermediate merge can continue making progress and the cost of restoring the data can be hidden, without delaying the merging phase. In addition, when there are unused buffers available, returning the buffers that contain recently fetched data can cause unnecessary data movement costs. To address these performance issues, we

organize all the available buffers within a queue and sorted them according to their usage status, and assign the next buffer from the head of the queue.

4.2.5 Fault Tolerance

Hadoop adopts restart fault-tolerance model to handle the node failures. During the job execution, the JobTracker periodically communicate with TasksTrackers through heartbeat message. If a TaskTracker fails to communicate with JobTracker for a period of time (by default, 1 minute in Hadoop), JobTracker will assume that TaskTracker has crashed. The JobTracker chooses another TaskTracker to re-execute all MapTasks, if it is in map phase (or ReduceTasks if it is in reduce phase), that previously ran at the failed TaskTracker.

Our virtual shuffling does not need complicate efforts to this simple and clean fault-tolerance model. If the jobs failed is in the map phase, JobTracker will choose another TaskTracker to re-execute the MapTasks. Meanwhile, all the ReduceTasks will re-fetch the segment headers belonged to the failed MapTasks, which leads to an update of the globale segment table. Similarly, if the jobs failed is in the reduce phase, another TaskTracker will re-execute the failed ReduceTasks and the global segment table will be created and all the intermediate data will be shuffled again. Virtual shuffling only renovates the shuffle strategy inside ReduceTasks, which can be viewed as an independent component and is not coupled with the TaskTracker. MapTask execution is completely decoupled from ReduceTask execution which is maintained as it is in the vanilla Hadoop. This property of virtual shuffling contributes to a less effort on the fault tolerance inside Hadoop.

4.3 A Scalable Parallel Community Detection Algorithm for Distributed Memory Systems

To address the modularity maximization challenges discussed in Section 2.3.1, we have introduced (1) a novel hash-based data organization for computation and communication parallelization; and (2) a novel heuristic that controls the fraction of vertices for hierarchical

supergraph construction. Together, these techniques are integrated into our parallel Louvain algorithm. Furthermore, we have designed a communication runtime that is specially optimized for lock-free message generation and fast communication on Blue Gene/Q systems.

4.3.1 Hash-Based Data Organization

We linearly organize the vertices and partition them among compute nodes. Each node is assigned a set of vertices, which we refer as the node *owns* the vertices. The same node is responsible for all the information related to the vertices. 1D partition is a good fit because our parallel algorithm requires multiple iterations and a synchronization is only needed after all iterations.

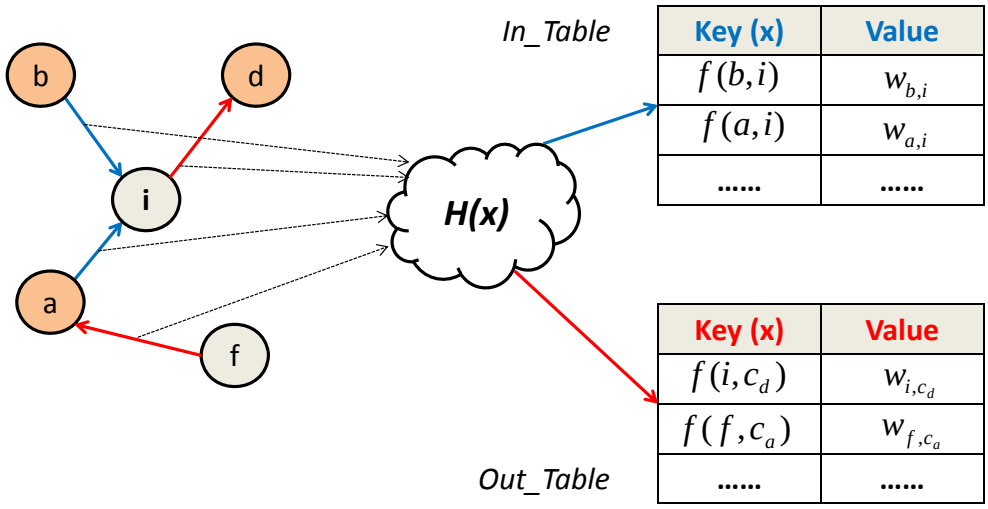


Figure 4.11: Edge Hashing

Instead of using the traditional sorting based implementation, we introduce a hash based data organization to gather all edges and reduce communication and synchronization. The edges belonging to the vertices on a compute node are managed by two distinct hash tables, *In_Table* and *Out_Table*. Figure 4.11 shows the hash mechanism for two representative vertices i and f on one processing node along with all the neighbors (a, b, d). We derive our data organization scheme based on the original Fibonacci hash function. The resulting hash function $H(x)$ used for our purpose is defined in Equation 4.1, where M is the size of the

hash table, W is $2^{64}-1$, and ϕ is the *golden ratio*.

$$H(x) = \lfloor \frac{M}{W} \cdot ((\phi^{-1} \cdot W \cdot x) \, mod \, W) \rfloor \tag{4.1}$$

As shown in the figure, both hash tables are hashed on edges. The difference is that the *In_Table* manages the in-edges (the edges with destination as i) and the *Out_Table* manages the out-edges (the edges with source as i). These tables use different keys. For *In_Table* the key field is represented by a tuple composed of the source vertex u and the destination vertex i of the edge being hashed. It is denoted as $f(u,i)$, $\forall e(u,i) \in E$, as shown by Equation 4.2. λ is a constant.

$$f(a,b) = (a << \lambda) \,|\, b \tag{4.2}$$

For *Out_Table*, the key field is represented by a tuple composed of the source vertex and the destination vertex's community for the corresponding edge, denoted as $f(i,c_u)$, $\forall u \in c_u, e(u,g) \in E$. For the *In_Table*, the value field represents the weight of the corresponding edges denoted as $w_{u,i}$. Because different neighbors of one vertex may belong to the same community, the value field in the *Out_Table* actually represents the sum of weights of all edges from the vertex u to a specific community, i.e., $w_{u \to c_i}$ in Equation 2.4.

4.3.2 A Novel Heuristic for Convergence

We have examined and analyzed the detailed behavior of the convergence property of the sequential Louvain algorithm through extensive simulation study. In particular, we study the trace of the movement of the vertices using a variety of graphs and observe an exponential relation between the movement of the vertices and the number of iterations. By using statistic regression to quantify such relationship, we identified a dynamical threshold ϵ,

$$\epsilon = |V| * \alpha * e^{\frac{1}{\beta * iter}}, \tag{4.3}$$

which identifies the fraction of the vertices that needs to be updated during each iteration of the inner loop. The threshold decreases exponentially with the number of iterations as shown by Equation 4.3, where $|V|$ is the current size of the graph, *iter* is the number of iterations of the current inner loop, α and β are the parameters obtained through regression training and analysis. ϵ can be translated to the threshold $\Delta\hat{Q}$ by sorting the $\Delta Q_u, \forall u \in V$ and selecting the bottom one from the top ϵ fraction. This purges out the vertices that will not contribute much to the modularity gain in the following iterations.

Once $\Delta\hat{Q}$ is obtained, the vertices can update the community information in parallel. $\Delta\hat{Q}$ helps preserve the same convergence and has been examined with large social graphs on modularity community quality as detailed in Section 5.3.2.

4.3.3 Parallel Louvain Algorithm

The pseudocode for the parallel Louvain algorithm is shown in Algorithm 2. The algorithm starts by initializing all the data structures. At the very beginning, the *In_Table* contains all the in-edges information of the vertices owned by each nodes. Each vertex's community is set to itself and *Out_Table* is set to empty. The algorithm starts with StatProp, which is a function to exchange messages on the community state. Once all the messages have been delivered and hashed in place, the *Out_Table* is initialized. We then invoke Refine, which corresponds to the inner loop of the sequential algorithm and allows the vertex move to different communities based on the modularity gain. After it converges to a certain point we reconstruct the graph (GraphReconstruction) and prepare for the next round of execution. The algorithm halts when there is no more movement of vertices.

4.3.3.1 Community State Propagation

We have designed an efficient communication runtime for message exchange and synchronization on top of our hash based implementation, which will be elaborated in Section 4.3.4. Here we only explain the algorithm. As shown in Algorithm 3, all the threads sequentially

Algorithm 2: Parallel Louvain Algorithm.

Input: k: current level;
p: processor;
$G = (V^0, E^0)$: graph representation at level 0;
C_p^0 community set at level 0 owned by processor p ;
V_p^0 vertices set at level 0 owned by processor p ;
c_u^0: the vertex u's community at level 0;

c_{bu}^0: the vertex u's best community at level 0;
m_u^0: the vertex u's maximum modularity gain at level 0;

$In\_Table_p^0$ In_Table at level 0 owned by processor p ;
$Out\_Table_p^0$ Out_Table at level 0 owned by processor p ;

Output: C: community sets at each level;
Q: modularity at each level

$k \leftarrow 0$;
$In\_Table_p^k \leftarrow ((u,v), w_{u,v}) \forall v \in V_k^p, \forall e(u,v) \in E$;
repeat
    $c_u^k \leftarrow u, u \in V_p^k$;
    $Out\_Table_p^k \leftarrow \emptyset$;
    `StatProp` () ;
    // Refine the vertices' community until it meets certain threshold.
    `Refine` () ;
    $k \leftarrow k + 1$;
    print C^k and Q^k;
    // Reconstruct the Graph.
    `GraphReconstruction` () ;
    if *No improvement on the modularity* **then**
        exit loop;

scans its own partition of the In_Table and send messages to the destination node, who owns the corresponding vertex (u in the algorithm) (Lines 4–6). In the meantime, the threads also receive messages from others and insert/update hashed edges to the Out_Table (Lines 8–13), using operations such as sequential scan and insert/update.

Algorithm 3: Community State Propagation

```
1  function StatProp
2  begin
3      // Scan In_Table and send messages.
4      for ((u, v), w) ∈ In_Table_p^k do
5          c ← C_v^k ;
6          send (u, c, w) to processor p'(u ∈ V_p')
7      // Update Out_Table.
8      for ((u, c), w) received do
9          hash tuple ((u, c), w) into Out_Table_p^k ;
10         if ∃((u, c), w') then
11             w' ← w' + w ;
12         else
13             linear probing until find a free bucket and place the tuple
14 end
```

4.3.3.2 Refine

Algorithm 4 depicts the Refine routine. It starts with initializing c_{bu}^k and m_u^k. The Out_Table contains all the neighbors' communities and the accumulated weights to the communities for the vertices owned by that compute node. Each node starts scanning the hashed edges (u, c) and then calculates the modularity gain of putting vertex u to Community c. It updates the corresponding c_{bu}^k and m_u^k if needed, as shown by Lines 7–10. Consider that different threads may process the same vertex u concurrently. The updates on the best community in c_{bu}^k and the maximum ΔQ in m_u^k need to be atomic. After all the Out_Tables have been examined, each vertex has the information on the best community to join.

For the need to converge, we then build a histogram based on m_u^k and calculate the updated threshold $\Delta\hat{Q}$ according to Equation 4.3. Lines 14–17 update the community vector

Algorithm 4: Refine.

```
1  function REFINE
2  begin
3      c_bu^k ← u, u ∈ V_p^k ;
4      m_u^k ← 0, u ∈ V_p^k ;
5      repeat
6          // Find best candidate community.
7          for ((u, c), w) ∈ Out_Table_p^k do
8              if ΔQ_{u→c} ≥ m_u^k then
9                  c_bu^k ← c;
10                 m_u^k ← ΔQ_{u→c}
11         // Build histogram.
12         All_reduce and build a histogram to calculate updating threshold ΔQ̂;
13         // Updating the community information.
14         ∀u ∈ V_p^k  if m_u^k ≥ ΔQ̂ then
15             // atomically update Σ_tot
16             Σ_tot^{c_bu^k} ← Σ_tot^{ĉ_bu} + w(u) ;  Σ_tot^{c_u^k} ← Σ_tot^{ĉ_u} − w(u) ;
17             c_u^k ← c_bu^k ;
18         StatProp ();
19         // Update Σ_in.
20         for ((u, c), w) ∈ Out_Table_p^k do
21             if (c = c_u^k) then
22                 // atomically update Σ_in
23                 Σ_in^c ← Σ_in^c + w
24         // Calculate community set and modularity
25         C_p^k ← {c_u^k}, ∀u ∈ V_p^k ;
26         Q_p^k ← 0 ;
27         for c ∈ C_p^k do
28             Q_p^k ← Q_p^k + Σ_in^c/2m − (Σ_tot^c/2m)^2 ;
29         Q^k ← Allreduce(Q_p^k);
30         if No improvement on the modularity then
31             exit loop;
32     C_p^{k+1} ← {c_u^k}, ∀u ∈ V_p^k ;
33     V_p^{k+1} ← C_p^{k+1} ;
34 end
```

and the corresponding Σ_{tot} based on $\Delta\hat{Q}$. The rest is to calculate the new modularity (Lines 18–29). For this purpose, we need the latest Σ_{in}^{c}, which is the sum of the weights of all edges inside Community c. We first perform a community state propagation to refresh the latest community information and weights (Line 18). Then we update the Σ_{in} for each community (Lines 20–23). Finally, we calculate the modularity (Lines 27–29). The procedure terminates when no further vertex movement can increase the modularity, and the community and modularity information is printed out.

Algorithm 5: GraphReconstruction.

```
1  function GraphReconstruction
2  begin
3      for ((u, c), w) ∈ Out_Table_p^k do
4          send (c_u^k, c, w) to processor p'(c ∈ C_p'^k)
5      // Reconstruct In_Table.
6      for (u, v), w) received do
7          hash tuples ((u, v), w) into In_Table_p^k ;
8          if ∃((u, v), w') then
9              w' ← w' + w ;
10         else
11             linear probing until find a free bucket and place the tuple
12 end
```

4.3.3.3 Graph Reconstruction

The novelty of our edge hashing is that the In_Table describes the initial graph and the Out_Table represents the graphs for communities over iterations.

When the edges are managed by the two hash tables, the reconstruction of graph can be easily achieved through an exchange of messages between Out_Table and In_Table. The GraphReconstruction routine is shown in Algorithm 5. It reconstructs a graph whose vertices are now the communities found during the Refine phase. To do so, the weights of the edges between the new vertices are given by the sum of the weight of the edges between vertices in the corresponding two communities[20]. Edges between vertices of the

same community lead to self-loops in the new graph. Lines 3–4 scan the Out_Table and send the aggregated edges information to the owner of the vertex (community). Upon receiving the messages, Lines 6–11 finally prepare the In_Table for the next round of execution. Our hash based graph reconstruction avoids intensive computation compared to the traditional graph relabeling technique.

4.3.4 Communication Runtime

Communication is always the major performance concern for parallel algorithms. Taking advantage of the hash based data structure, we have carefully designed a powerful communication runtime, which is able to serve the need of exchanging the edge information for graph algorithm with close to maximum bandwidth. The MPI based implementation is highly portable and agnostic to any parallel systems. We also optimized the runtime for special network such as Blue Gene/Q network.

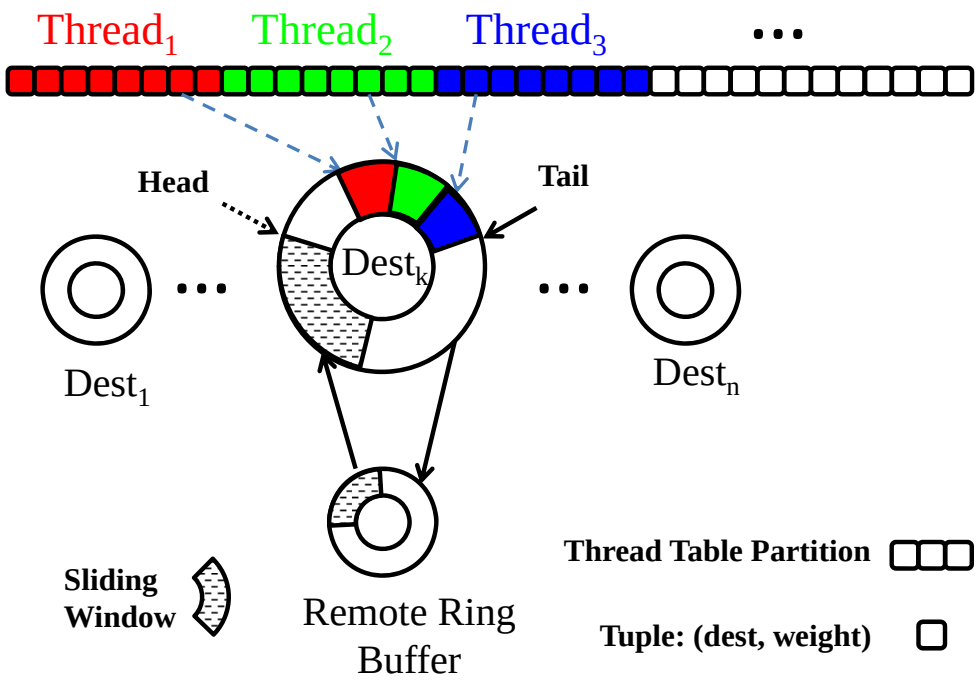


Figure 4.12: Communication Runtime

Figure 4.12 shows the working mechanism of the runtime, where a representative node is presented. As mentioned in the previous algorithm, The In_Table contains the in-edge

information and the Out_Table contains the out-edge information for the vertices owned by the node. The worker threads on the node partition these two tables and manage their own region independently. The communication happens when all the nodes exchange the edge information between the In_Table and the Out_Table.

On each node, the runtime allocates a ring buffer for each destination node ($Dest_i$ in Figure 4.12). Each ring buffer contains a sliding window and two pointers, *head* and *tail*, for flow control with the remote ring buffer. To communicate messages, the threads start scanning their own In_Table region and then write messages (8 bytes or 16 bytes) into the ring buffer for the destination. Since small messages do not yield good bandwidth, our communication runtime will aggregate messages until a window of 512 bytes is filled up and then send out the window as a whole.

On the sender side, multiple threads write into the same ring buffer. To minimize the contention cost, the *tail* pointer controls the write position. It is increased atomically by the threads. If one thread has acquired an available slot, it does not mean that it can write the message immediately because the corresponding buffer may still have outstanding communication. The write to the available slot is granted only if the *head* and the *tail* fall into the same window, which means the buffer is available. When a window is filled up with messages, the aggregated message is sent to the destination node. The *head* pointer is updated when the message completes. Our communication runtime can use either one-sided *put* or two-sided *send*.

On the receiver side, it is the responsibility of receiving threads to process the arrived messages and hash them into the Out_Table. In terms of the sequential scan operation, the worker threads can perform such operations without interference with each other. For the insert/update operations, contention happens when multiple threads hash messages to the same bin. A lock operation will be inefficient for concurrent threads. We adopt a lock-free design for insert/update operations based on the *Compare-and-Swap* (CAS) primitive. Update/insert operations on the same bin will happen only if the CAS operation is successful.

4.3.4.1 Special Design for Blue Gene/Q network

Blue Gene/Q is a five-dimensional (5D) torus, with direct links between the nearest neighbors in 5 directions. Each link runs at 2 GB/s (2 GB/s send + 2 GB/s receive). The network supports point-to-point messages, collectives and barriers/global interrupts over the same physical torus. To achieve a high processing rate, our communication runtime has leveraged several features from the special Blue Gene/Q architecture.

(1) SPI communication layer: The inter-node communication in our communication runtime is implemented at the SPI level, a thin software layer that allows direct access to hardware resources such as injection and reception DMA engines of the network interface. Each thread has private injection and reception queues and its communication does not require locking. In addition, our collective operations are completely nonblocking. Barriers and all-reduces are overlapped with the program execution. These operations take advantage of the collective acceleration units in the network routers that can perform reduce operations at the line rate, along with combining and broadcast capabilities.

(2) L2 Flush: To send a message, the Blue Gene/Q network interface can read data directly from the L2 cache. This provides an opportunity to flush the write queue directly into the L2 cache instead of memory. Our communication runtime exploits this feature. To send a message to the network, the sending thread only makes sure that the window buffer is present in the L2 Cache before calling the send function.

Chapter 5

Performance and Evaluation

In this chapter we present experimental results from the systematical performance testing that has been performed on our implementation.

5.1 Performance Evaluation of HiCOO

Our evaluation of HiCOO were conducted on Kraken and Jaguar supercomputers (both Cray XT5 systems). Both computers have dual hex-core Opteron processors, a total of 12 cores per node. Because of the similarity in the processor architecture and interconnection network, we will not distinguish between Kraken and Jaguar, but mention the number of processes in our experimental results.

5.1.1 Analysis of Memory Management and Contention Attenuation

In this section, we describe our experiments that evaluate the impact of different virtual topologies on memory management and contention attenuation. Performance results from these topologies are compared to the original ARMCI that uses the FCG pattern for request buffer allocation.

5.1.1.1 Scalable Memory Management

Jaguar runs the Compute Node Linux operating system. On each node, the /proc file system reports the memory footprint of all processes as the resident working set size (VmRSS). We create an ARMCI program that reports VmRSS from all processes. This number represents the total memory consumed by an ARMCI process at runtime before any additional application-level memory consumption.

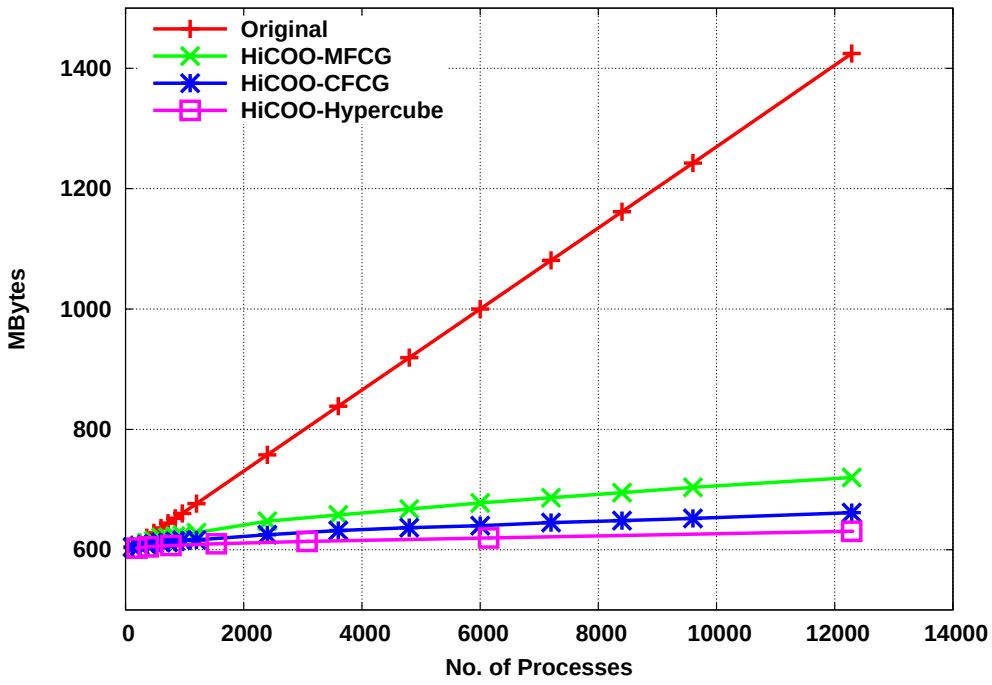


Figure 5.1: Scalability Virtual Topologies for Memory Management

We measure the impact of virtual topologies on memory resources. Our experiments are conducted with 12 processes per node. All processes start with a memory consumption of about 612 MBytes. However, due to the allocation of request buffers by the internal CS, a master process requires more memory for an increasing number of remote processes. The size of each buffer in CS is 16KB; and the number of buffers per process is 4. Figure 5.1 shows the memory consumption of master processes, using different virtual topologies. As expected, the memory requirement of the original ARMCI increases linearly. On 12,288 processes, the original has a memory consumption of 1,424 MBytes, an increment of 812 MBytes, on top of 612 MBytes that is needed to run a few processes. The other three virtual topologies provide much better scalability in terms of memory resources. Compared to the original, HiCOO-MFCG, HiCOO-CFCG, and HiCOO-Hypercube cut down the increment in memory consumption significantly, by 7.5, 16.6, and 45 times, respectively.

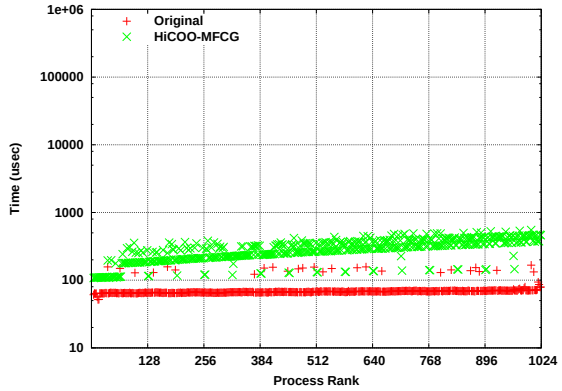
5.1.1.2 Contention Attenuation

Virtual topologies are also designed to address the other critical challenge, hot-spot contention in the GAS runtime. We evaluate contention for all one-sided ARMCI operations, and observe that virtual topologies are beneficial to the contention caused by lock, accumulate, noncontiguous data transfer, and atomic operations. Herein presented are results for two representative operations, noncontiguous vector data transfer and atomic Fetch-&-Add operations.

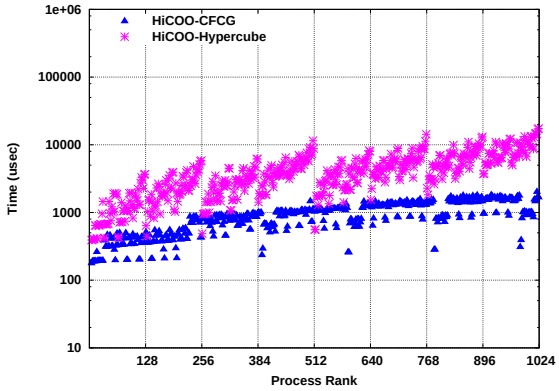
Description of Contention Experiments We define hot-spot contention as the percentage of processes in a program that are contending for communication to a single process, or access to a single data element. It is understood that such contention can arise from sources outside of a program, e.g., from other programs or system services. But, for practical purposes, we consider those beyond the scope of this study, and focus on hot-spot contention within a program.

We use programs with 1,024 processes for contention assessment, 4 processes per node across 256 nodes. These numbers provide a reasonable balance between the need of many nodes to exhibit contention and the need of clarity in visualizing all data points of the results. In these programs, each process (except those on the same node with Rank 0), prepares its data as needed (vectored or strided data in the case of noncontiguous data transfer operations), and then performs one or more one-sided operations to Rank 0. This is then repeated for 20 iterations. The average time for these iterations is taken as the time to complete an operation between the respective process and Rank 0.

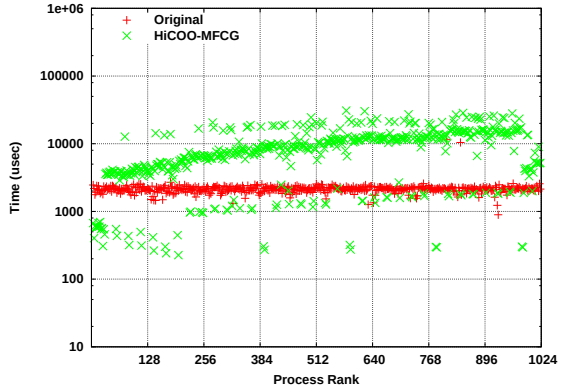
Measurements are collected under three different contention scenarios. In the first scenario, each process sequentially performs its own one-sided operations to Rank 0, repeats for 20 iterations, and records the time. At the same time, all other processes are idle in a barrier. This effectively measures the performance of one-sided operations between Rank 0 and all other processes, without any contention. In the second scenario, each process sequentially



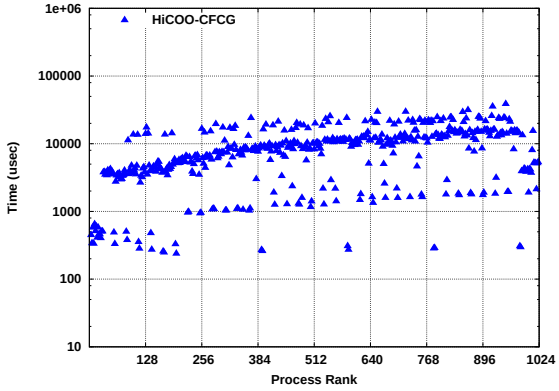
(a) Original-ARMCI & HiCOO-MFCG with No Contention



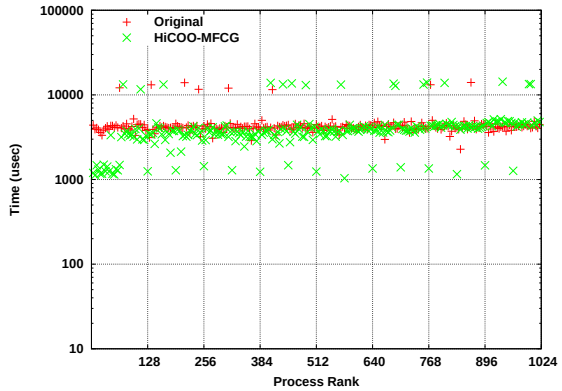
(b) HiCOO-CFCG & HiCOO-Hypercube with No Contention



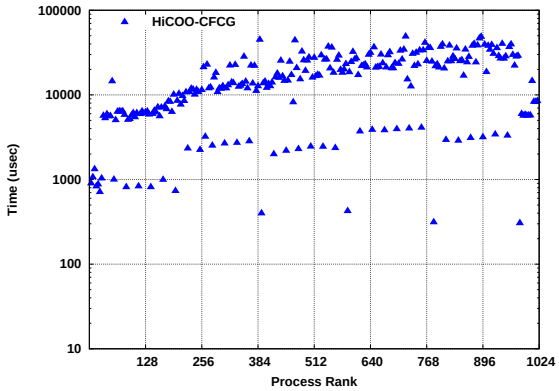
(c) Original-ARMCI & HiCOO-MFCG with 11% Contention



(d) HiCOO-CFCG with 11% Contention



(e) Original-ARMCI & HiCOO-MFCG with 20% Contention



(f) HiCOO-CFCG with 20% Contention

Figure 5.2: Vectored Data Transfer Operations Under Different Contention

performs the same number of operations to Rank 0, for the same number of iterations. However, in the meantime, one in every nine processes performs the same operations to Rank 0, while the remaining processes are idle in a barrier. Therefore this corresponds to 11%

contention. The third scenario is very similar to the second one, except that one in every five processes concurrently invokes one-sided operations to Rank 0. This then corresponds to 20% contention.

Noncontiguous Data Transfer Operations We conduct experiments to measure the performance of vectored put and get operations as representatives of noncontiguous data transfer functions. Figure 5.2 shows the time of vectored put operations from all remote processes to Rank 0. Comparisons are provided among varying levels of contention (no contention, 11% contention, and 20% contention).

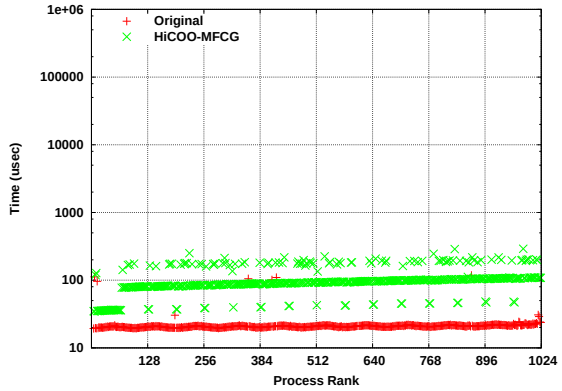
Figures 5.2(a) and (b) show the comparisons under no contention. Several behaviors are revealed by this figure. First, the use of MFCG, CFCG and Hypercube increases the time to complete noncontiguous data transfer operations between Rank 0 and other processes. Second, even though all processes are one step away from Rank 0 in the original ARMCI, the time to complete noncontiguous data transfers gradually increases with process rank. This suggests that the distance between a processes and Rank 0 in the underlying physical topology would play a role and contribute to the increased performance. This increment of time is magnified by the use of MFCG, CFCG and Hypercube. In particular, the results from HiCOO-Hypercube indicate that using a topology with very high dimensions for minimal memory consumption does not provide a good tradeoff to the performance. Third, with MFCG, the performance numbers from all processes form several distinct curves, representing differences in their (virtual-) topological relationship with respect to Rank 0. The same can be observed for HiCOO-CFCG and HiCOO-Hypercube as shown in Figure 5.2 (b).

Figures 5.2 (c), (d), (e), and (f) show performance comparisons with increased contention. HiCOO-Hypercube is not included in (e) and (f) because it takes too long to get a complete set of numbers. While contention increases the time to complete noncontiguous data transfer operations for all cases, it is evident that all virtual topologies exhibit contention resilience. While the performance of vectored put operations is degraded by nearly

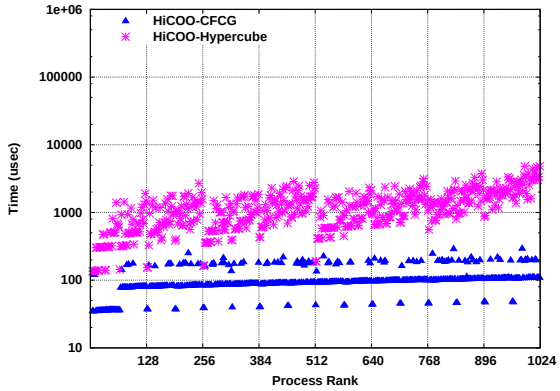
two orders of magnitude due to contention in the original ARMCI. With 20% contention, it becomes faster to complete noncontiguous data transfer operations for nearly all processes in the case of HiCOO-MFCG, compared to the original ARMCI. Comparing Figures 5.2 (c) and (e) it is interesting to note that HiCOO-MFCG also reduces the variations among all processes at higher hot-spot contention. The operation time for the group of processes in the middle has been brought down. This counterintuitive observation is because of the execution behavior of ARMCI communication server. When more processes are actively forwarding requests, they stay in the polling mode for handling requests and therefore have better response time in average. In summary, these results demonstrate that virtual topologies, such as MFCG and CFCG, can attenuate the pressure of many contending noncontiguous data transfer operations, and lead to graceful resilience to contention.

Atomic Fetch-&-Add Operations We measure the performance of fetch-&-add as a representative of atomic operations. Figure 5.3 shows the time for fetch-&-add operations from all remote processes to Rank 0. Comparisons are provided among different virtual topologies, and among varying levels of contention (no contention, 11% contention, and 20% contention).

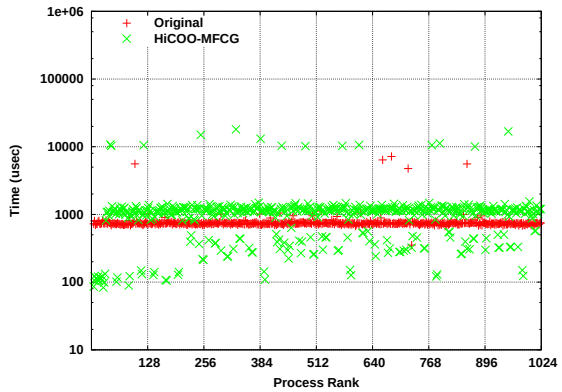
Figures 5.3 (a) and (d) show the comparisons under no contention. Similar observations can be made for atomic operations as revealed by Figures 5.2 (a) and (d). To be brief, these include (1) the use of MFCG, CFCG and Hypercube topologies increases the time to finish atomic operations under no contention; (2) the time of an atomic operation increases with a higher ranked process, suggesting a correspondence to the distance between the process and Rank 0 in the underlying physical topology; and (3) the performance numbers of atomic operations from all processes form several distinct groups, representing their relationship in the virtual topologies.



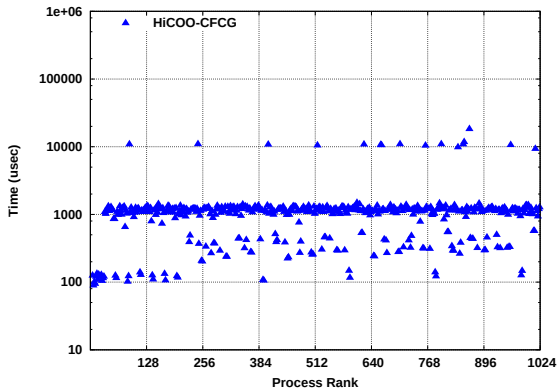
(a) Original-ARMCI & HiCOO-MFCG with No Contention



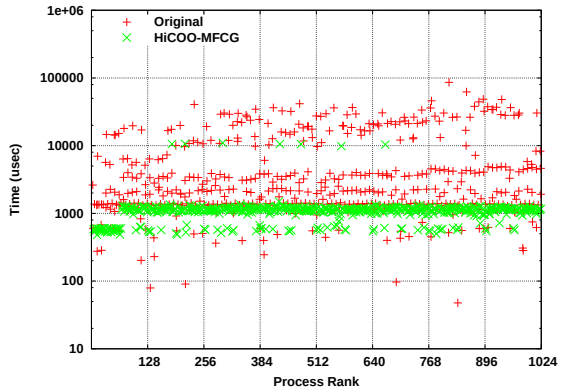
(b) HiCOO-CFCG & HiCOO-Hypercube with No Contention



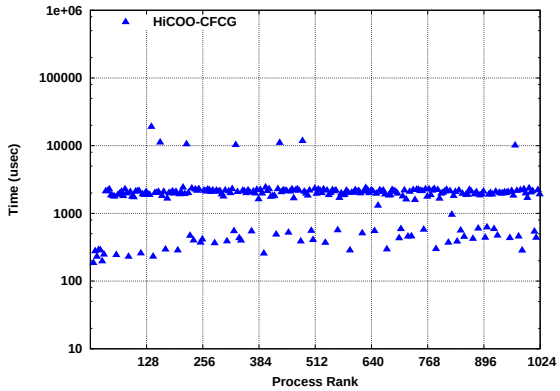
(c) Original-ARMCI & HiCOO-MFCG with 11% Contention



(d) HiCOO-CFCG with 11% Contention



(e) Original-ARMCI & HiCOO-MFCG with 20% Contention



(f) HiCOO-CFCG with 20% Contention

Figure 5.3: Fetch-&-Add Operations Under Different Contention

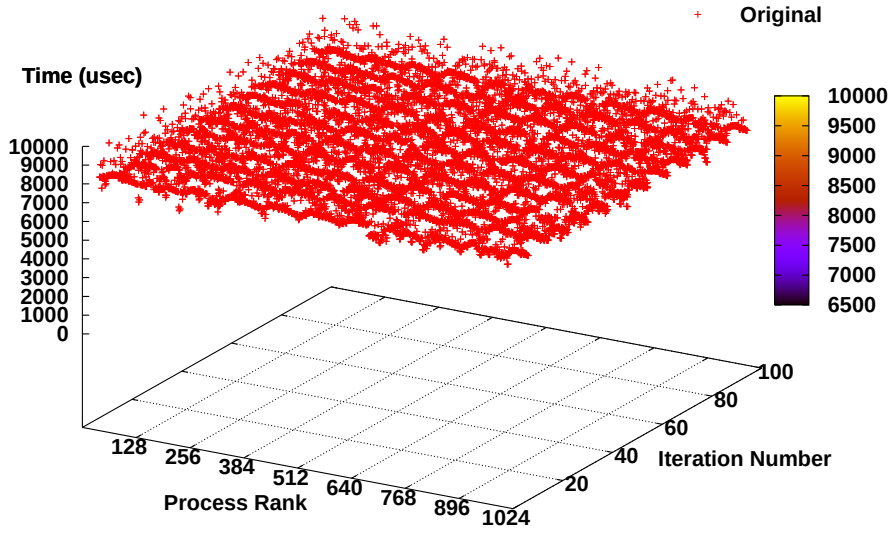
Figures 5.3 (b), (c), (e), and (f) show comparisons with increased contention. Again, hypercube was not included in (e) and (f). While contention increases the time to complete atomic operations for all cases, it is also evident that all virtual topologies exhibit

contention resilience. With 20% contention, it becomes faster to complete atomic operations for nearly all processes using HiCOO-MFCG than the original ARMCI. Under the same level of contention, even with HiCOO-CFCG, the time for fetch-&-add is shorter for a majority of processes compared to the same with the original ARMCI. These results again demonstrate that virtual topologies, such as HiCOO-MFCG and HiCOO-CFCG, can greatly attenuate the pressure of contending atomic operations.

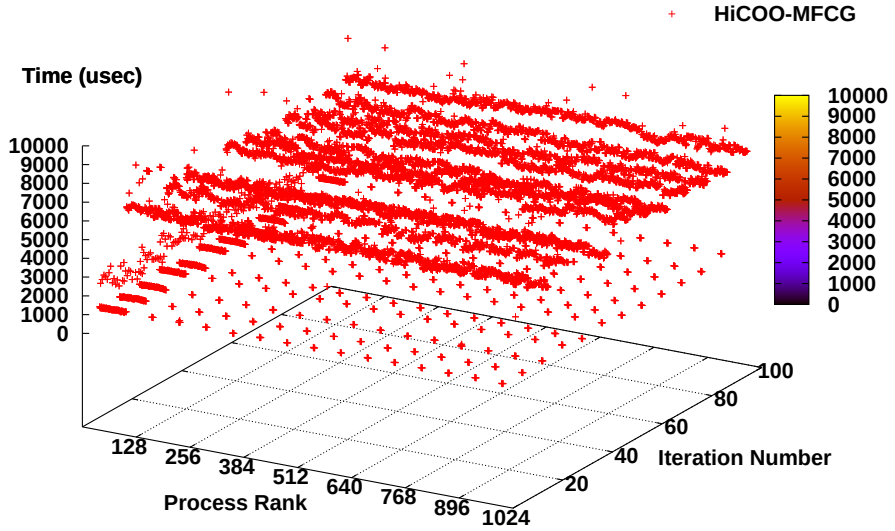
Furthermore, we investigate the benefits of HiCOO-MFCG for fetch-&-add operations under 100% contention, i.e., all processes concurrently performs atomic fetch-&-add operations to Rank 0. In this test, within two consecutive barrier (and ARMCI AllFence) operations, all process concurrently invoke 10 fetch-&-add operations, record the time, and measure the average for these 10 operations on their own. This is repeated for 100 iterations. Figure 5.4 shows the performance comparison between the original ARMCI and HiCOO-MFCG, with 1,024 processes and 100 iterations. With 100% contention, it takes 8,000+ μsec in average for a process to complete an atomic operation when using the original ARMCI. In contrast, when using HiCOO-MFCG, contention is dramatically reduced. Many processes finish within 2,000 μsec; nearly all processes complete in 6,000 μsec. This proves that the virtual topology HiCOO-MFCG is especially useful in attenuating the impacts of heavy contention.

5.1.2 Performance of Communication Operations and Scientific Applications

We have shown that virtual topologies can be very beneficial to reduce memory footprint and attenuate contention that would occur to hot-spot processes. It is important to find out how HiCOO will impact the ARMCI communication operations and what benefits they have to real applications, and how HiCOO will benefit real applications. In the rest of experiments, we focus on HiCOO using the default topology MFCG.



(a) Original-ARMCI



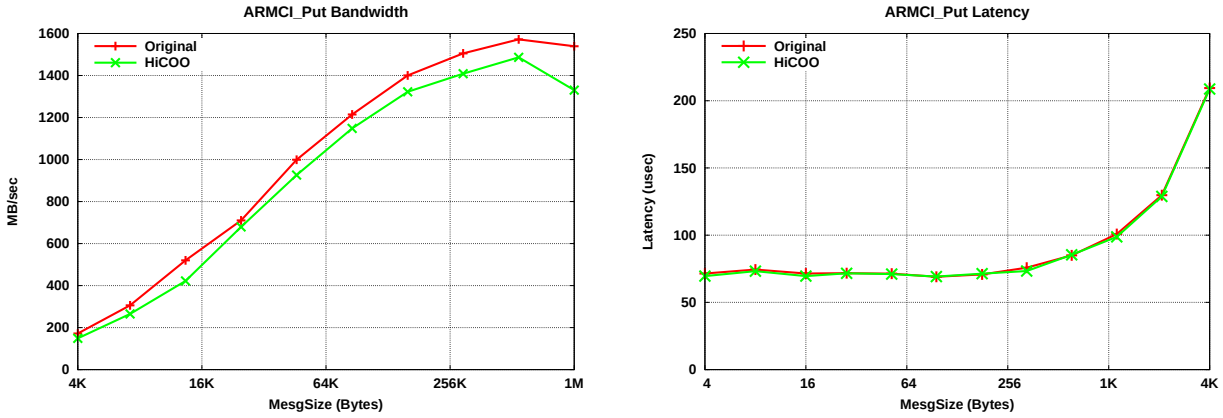
(b) HiCOO-MFCG

Figure 5.4: Fetch-&-Add Operations under 100% Contention

5.1.2.1 ARMCI One-sided Operations

ARMCI offers a rich set of one-sided communication primitives for GAS programming models. These include (1) contiguous and noncontiguous data transfer operations, (2) atomic

operations, (3) locks, and (4) synchronization operations. While multinode cooperation is intended to address challenges faced by direct one-sided communication in the original ARMCI, it is important to measure the performance impact of multinode cooperation to these one-sided operations.

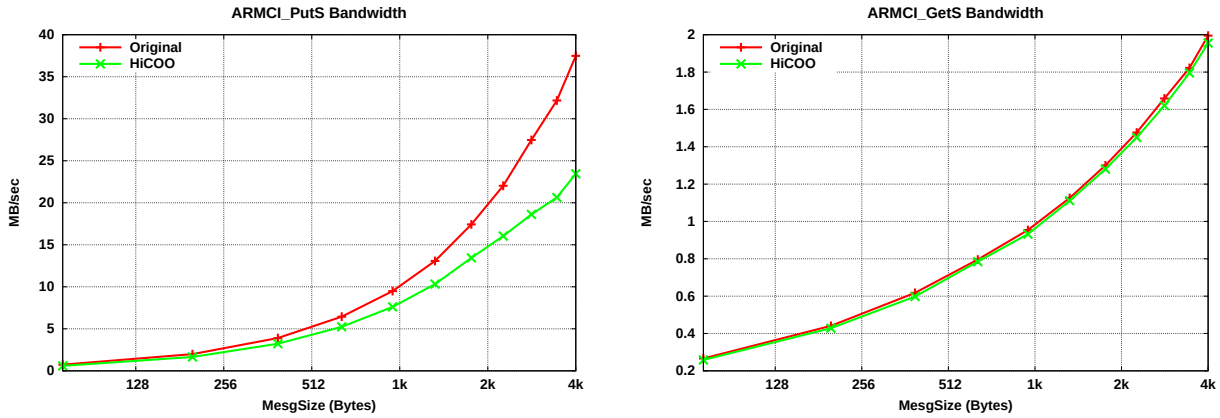


(a) ARMCI_Put Bandwidth (b) ARMCI_Put Latency

Figure 5.5: ARMCI_Put Latency and bandwidth

Contiguous Data Transfer Operations ARMCI supports contiguous data transfer operations, including direct put and direct get. On the Cray XT5, these direct put/get operations transfer contiguous data directly between source and destination memory, using native portals put and get operations on the Seastar2+ network. No one-sided requests are sent for these operations, and communication servers are not involved for these operations. We measure the performance of direct put/get operations across 16 nodes, each with 12 processes. These nodes form four groups of cooperative nodes. Our latency and bandwidth tests are different from the conventional ping-pong latency and stream-based bandwidth tests. 16 nodes are used to mimic the presence of message forwarding and compare the performance between the original and HiCoo cases. Since there are 12 processes on each node, the latency and bandwidth numbers are measured when each node (and its network card) has a heavy load of communication generated by many processes. Figure 5.5 shows the latency and bandwidth performance comparison between ARMCI and multinode cooperation.

It is clear that our design of multinode cooperation has very little impact on the performance of contiguous data transfer operations. Note that, for succinctness, we only show the performance for direct put operations. The comparison is the same for direct get operations.

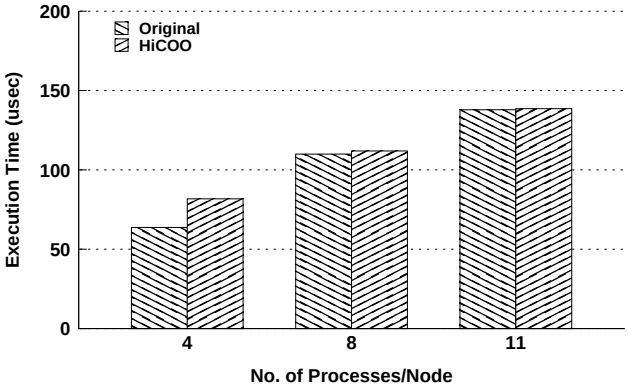


(a) ARMCI_PutS (b) ARMCI_GetS

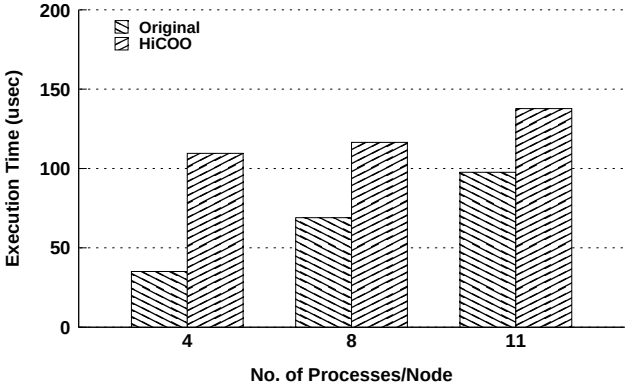
Figure 5.6: Bandwidth of Noncontiguous Operations

Noncontiguous Data Transfer Operations Multidimensional data arrays are commonly adopted by scientific applications for numerical analysis and matrix calculation. When such an array is decomposed into many parallel processes, each process typically owns a noncontiguous set of data elements. ARMCI supports the movement of such noncontiguous data through vectored I/O and strided I/O. The former is a generalized I/O format that describes noncontiguous data segments with a series of <addr, length> pairs; the latter is an optimization when segments are of the same length and distance from each other.

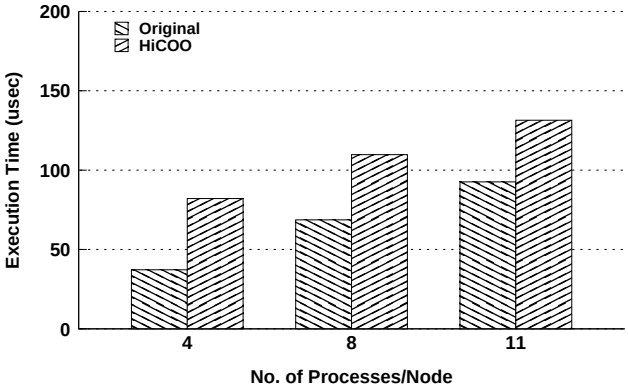
We have measured the performance of ARMCI strided data transfer. Our experiments are conducted on sixteen nodes each with 11 processes. Processes on the first node are paired with, and initiate one-sided ARMCI_PutS (and ARMCI_GetS) operations to, their counterparts on the last node. Figure 5.6 shows the performance results of ARMCI for short messages, with and without multinode cooperation. ARMCI_PutS requests are usually large and contain data inside. So they cannot be merged. Large requests with data need to be forwarded to the target server separately as the size of aggregation buffer is limited. On the other hand, ARMCI_GetS requests have to retrieve data separately. This leads to



(a) ARMCI_Lock



(b) ARMCI_Accumulate



(c) ARMCI_Fetch & Add

Figure 5.7: Performance of Atomic and Synchronization Operations

very low bandwidth for ARMCI_GetS operations in general. But there is little difference between the original ARMCI and HiCOO, These results indicate that, while the performance of noncontiguous data put operations can be affected by the additional overhead of request forwarding, HiCOO is effective in minimizing such overhead with its request aggregation and hierarchical cooperation mechanisms, resulting in close performance to the original ARMCI.

Atomic and Synchronization Operations ARMCI supports a number of atomic and synchronization operations for GAS models. These include lock, accumulate, and fetch-&-add. The lock operation acquires a specified mutex on the target process on behalf of an initiating process. The accumulate operation atomically updates one or more variables on the target process. The fetch-&-add operation retrieves an integer variable at a remote location, and at the same time atomically updates the value by an integer.

We measure the performance of these operations across 16 nodes. These nodes are grouped into four sets of cooperative nodes. All processes are paired with each other for atomic and synchronization operations. In order to evaluate the performance of multinode cooperation, we tested different numbers of processes (4, 8, and 11) per node. For example, a process on Node 0 initiates lock, accumulate, or fetch-&-add operations 1000 times (after the first 50 warm-up operations) to its counterpart on Node 15. The average time is calculated as the time for an operation.

Figure 5.7 shows the performance results for all three operations. HiCOO achieves performance comparable to the original ARMCI for atomic lock operations, but it causes performance degradation for the accumulate and fetch-&-add operations. The different performance comparison is due to the underlying communication of these operations. Atomic lock operations do not transmit actual data between processes, but accumulate and fetch-&-add operations do.

Taken together, our microbenchmark evaluation results indicate that while HiCOO strives to minimize memory consumption, its indirect one-sided communication does cause performance overhead to atomic and synchronization operations. Care must be taken to achieve a good tradeoff between the memory consumption and the cost of atomic operations for applications need to use frequent atomic operations.

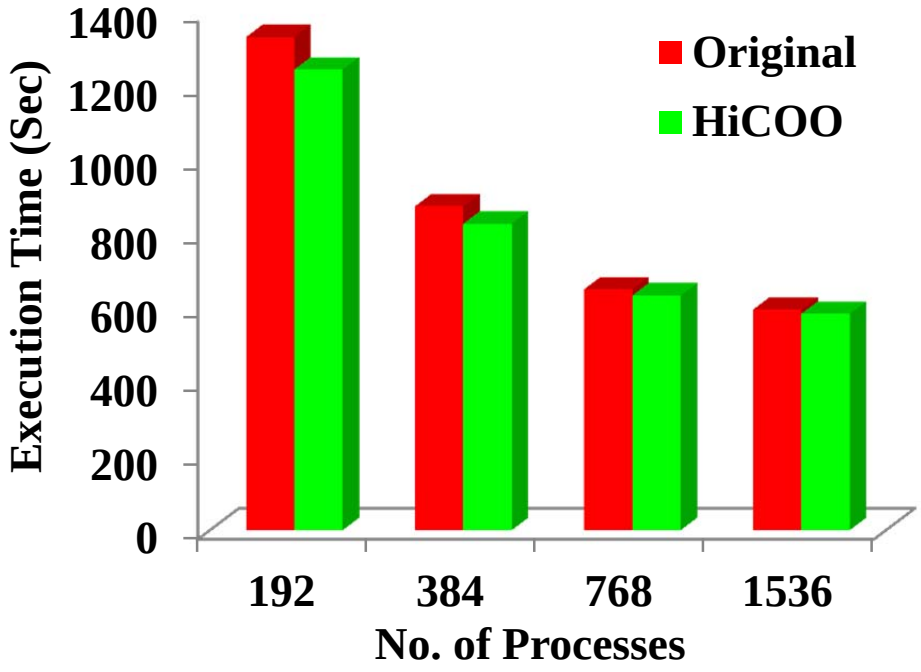


Figure 5.8: The Performance of NAS LU

5.1.2.2 NAS LU Application

The LU application in the NAS parallel benchmark suite [77] has been ported to the ARMCI runtime. It can scale to hundreds or a couple of thousand processes. We evaluate the performance impact of HiCOO to LU at this scale. Figure 5.8 shows the performance of LU using HiCOO on a varying number of processes. As shown in the figure, HiCOO performs better or similar to the original ARMCI. At a lower number of processes, the benefit of HiCOO is slightly higher. Two observations can be made about these results. First, the LU application does not suffer much from hot-spot contention. Second, the reduction in memory footprint does not directly lead to the reduction in execution time, which is quite reasonable. On the other hand, these results are encouraging because they demonstrate that, despite the additional forwarding steps on ARMCI operations such as non-contiguous data transfer and atomic accumulation, HiCOO still brings comparable or better performance for applications such as LU.

5.1.2.3 A Large-Scale Application: NWChem

To examine the benefits of our techniques to the real-world scientific application, we focus on NWCHem [78], which is a widely used large-scale computational chemistry package. It contains many methods for computing properties of molecuar and periodic systems using standard quantum mechanical descriptions of the electronic wavefunction or density. We evaluate HiCOO using two most widely used electronic structure methods in NWChem: the SiOSi3 method for Density Functional Theory (DFT) and the water model of Coupled Cluster (CC) in its CCSD(T) incarnation. DFT is the workhorse of electronic structure for its balance between computational cost and accuracy (1998 Nobel prize in Chemistry), whereas the more expensive CC method, in its CCSD incarnation, is labeled as the "gold standard" [79] because of its remarkable accuracy.

The performance of SiOSi3 is shown in Figure 5.9. HiCOO clearly performs better than the original ARMCI. HiCOO reduces the total execution time by as much as 52%. These results suggest that SiOSi3 is very prone to hot-spot contention, in which case HiCOO is very effective in mitigating the impact of contention.

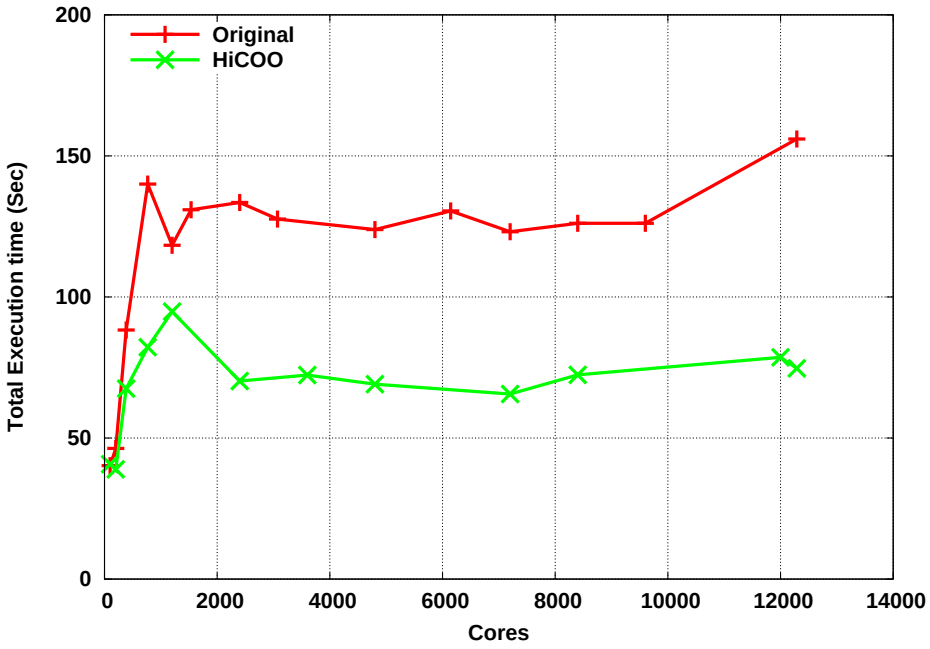


Figure 5.9: DFT SiOSi3 Execution Time

Figure 5.10 shows the performance of CCSD(T) water model when using the original ARMCI and HiCOO. ARMCI generally performs better than HiCOO, except in one case at 10,000 cores. This result suggests that the total execution time for the water model does not benefit from HiCOO. The primary benefit of HiCOO is the ability to significantly reduce memory consumption of ARMCI low-level runtime library (as detailed in Section 5.1.1.1). This spares much more memory to be used by applications and help them achieve better scaling.

These application evaluation results demonstrate that HiCOO can hit the best balance of memory consumption, the need of request forwarding, and contention attenuation for the GAS runtime. With much reduced memory consumption at the runtime level, HiCOO in general performs comparably to the original ARMCI. Particularly when an application is experiencing hot-spot contention, HiCOO can mitigate the impact of contention and lead to significantly reduced total execution time.

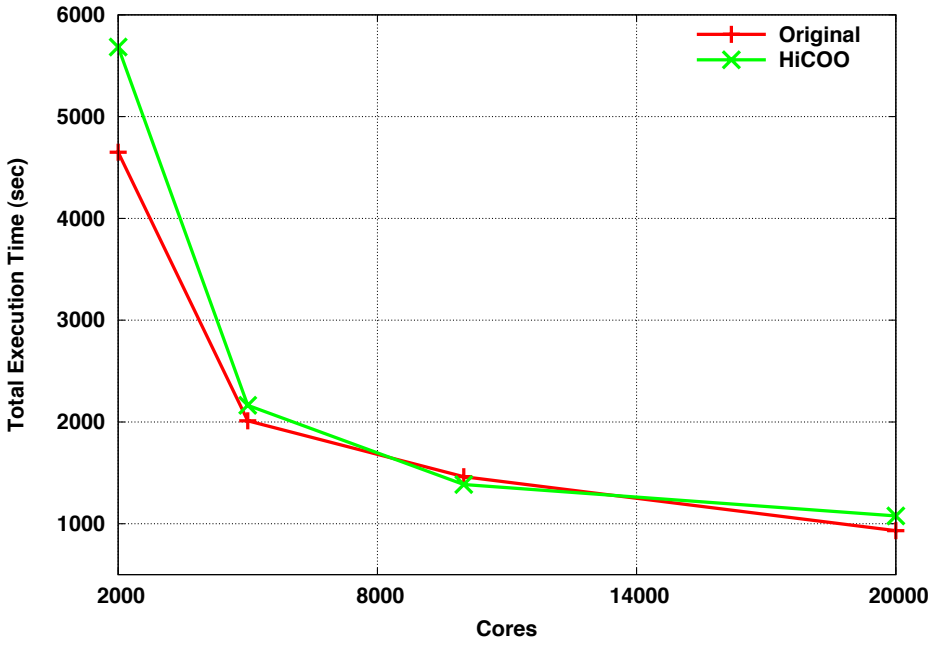


Figure 5.10: $(H_2O)_{11}$ CCSD(T) Execution Time

5.2 Performance Evaluation of Virtual Shuffling

The original Hadoop only supports physical shuffling. We implement virtual shuffling based on our previous work on Hadoop Acceleration [75], Hadoop-A. Hadoop-A used to support only network-levitated merge. For a fair comparison, we implement physical shuffling inside Hadoop-A. The three-level segment table is designed to hold as many as 1024 entries per level, and more than 1 billion virtual segments together. But we realize that it is not necessary to populate the entire SGD table. So our implementation keeps only one entry at the SGD level. SGB and SMB buffers are equivalent and unified into a single level. Nonetheless, for future MapReduce programs with big data sets, our implementation can be easily turned into a real three-level segment table and host more than 1 millions virtual segments. With this simplification, <key,val> pairs from an SEB only need to go through memory-based data merging twice before being delivered to the reduce function. In addition, with tree-based virtual segments, while subtrees are alternatively activated and merge SEBs into STBs, STBs also need to be merged to SGBs (no separated SMBs as mentioned above). All these merging tasks are undertaken by the same pool of merging threads. The reduce function can get delayed if there are no data in SGBs. To minimize such delays, we enforce a higher priority to the task of merging STBs into SGBs, allocate and activate a thread when there is a need to refill SGBs with more <key,val> pairs.

Experimental Testbed – Our experiments were conducted on PASL cluster of 21 compute nodes from Auburn University. Each node is equipped with dual-socket quad-core 2.13GHz, Intel Xeon processors and 8 GB of DDR2 800 MHz memory, along with 8x PCI-Express Gen 2.0 bus. All nodes are equipped with Mellanox ConnectX-2 QDR Host Channel Adapters, which can run in either InfiniBand mode or 10Gigabit Ethernet mode. These nodes are connected to both a 108-port InfiniBand QDR switch and a 48-port 10GigE Vantage switch. We use the InfiniBand software stack, OFED [80] version 1.5.3.2 released by Mellanox. Each node has one 500GB, 7200 RPM, Western Digital SATA hard drive and Hadoop version 0.20.0 was used.

Test Benchmarks – We evaluated virtual shuffling with a number of popular public benchmarks as shown in table 5.1, which have been heavily used for web crawling, and compared with physical shuffling. These include the *TeraSort*, *Grep*, and *WordCount* test programs that are distributed as part of Hadoop package and InvertedIndex, TermVector, and SequenCount developed by Faraz *et al.* [81]. Grep searches in a text file for a predefined expression and creates a file with matches. WordCount counts the number of occurrences of different words in a data file. TeraSort is a popular benchmark that measures the capability of a program in sorting a large-scale dataset. InvertedIndex takes a list of documents as input and generates word-to-document indexing. TermVector determines the most frequent words in a set of documents and is useful in the analyses of a hosts relevance to a search. SequenCount generates a count of all unique sets of three consecutive words per document in the input data. In addition, we examined a representing benchmark of Hive [82], which is a a high-level query language that is designed to facilitate user queries for data processing and analysis over Hadoop.

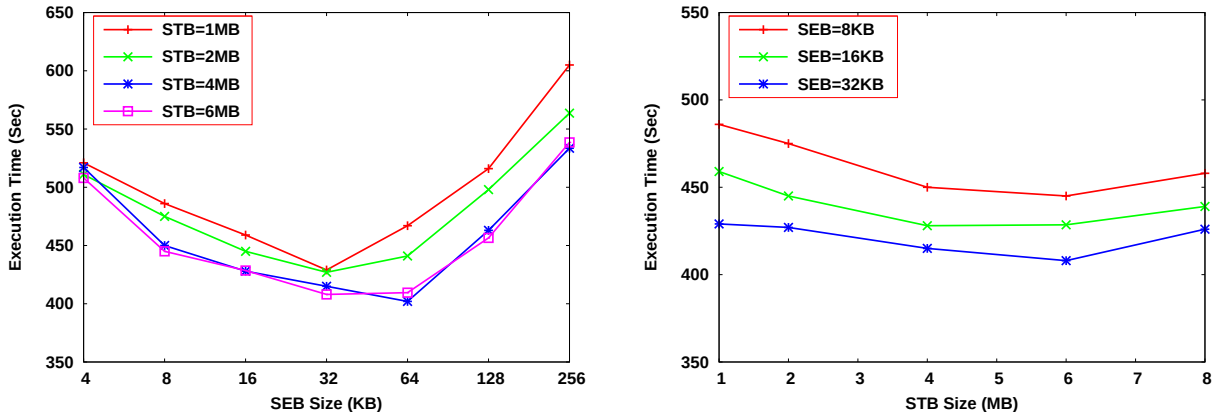
Table 5.1: Benchmarks Description

Benchmark	Description
B1	Grep (20GB)
B2	WordCount (20GB)
B3	InvertedIndex (20GB)
B4	TermVector (20GB)
B5	SequenceCount (20GB)
B6	Hive (Order By) (60GB)
B7	TeraSort (512GB) (512GB)

5.2.1 Parameter Tuning of Virtual Shuffling

Virtual shuffling enables a seamless flow of data, starting from MOF (Map Output File) segments, going through a series of steps including fetching, buffering, and merging, and finally reaching the reduce function at ReduceTasks. A number of important parameters, such as SEB and STB buffer sizes and the number of virtual segments in a subtree, need to

be tuned for this pipeline to work efficiently. In our tuning tests, we have screened a variety of different data sizes on a number of different nodes. For brevity, we present only a few representative case studies, running the Terasort program with the data size being 128GB and the data split size 128MB. This results in a total of 1024 data splits in the job, which also equals to the number of data segments per ReduceTask.



(a) SEB Buffer Size

(b) STB Buffer Size

Figure 5.11: Tuning of Memory Buffer Sizes

5.2.1.1 SEB and STB Buffer Sizes

In the three-level segment table, SEB and STB are the primary interfaces for merging data segments into the SGB. Their sizes can affect the effectiveness of the fetching and merging processes. It is important to understand how they impact the performance of MapReduce programs. To do this, we first fix the number of virtual segments in a subtree, which we choose as the square root of the total number of segments.

Figure 5.11(a) shows the tuning of SEB buffer size. For a fixed STB buffer size, with an increasing SEB buffer size, the execution time decreases. This is because, when a large SEB buffer is used, the data fetching speed can catch up with the merging speed of the active subtree, which benefits the pipeline of fetching and merging. However, when the SEB buffer size goes further up, the execution time becomes worse. This is because more data from SEBs are dumped when a subtree is deactivated. The same data often have to be re-fetched

from remote segments. Resulting in wasteful and repetitive disk accesses. This delays the effectiveness of data fetching and stalls the entire pipeline. As shown in the figure, for the performance curves with different STB buffer size, the bottoms of these curves are different. The best SEB buffer sizes are 32KB and 64KB for different STB buffers. Since the STB buffer sizes are different, the fetching speed and merging speed on the subtrees are different as well. The bottom only happens at the balanced point where the data fetching speed catches up well with the merging speed.

To reveal more performance impact of the STB buffer size, we conduct another experiment to tune the STB buffer size with a fixed SEB buffer size. Figure 5.11(b) shows the results of this tuning experiment. With an increasing STB buffer size, the job execution time gets shorter and reaches the lowest point when the STB size is between 4MB and 6MB. After that, the job execution more or less stays flat. The underlying cause of this behavior is again the interplay between the speed of data fetching and that of merging. In another word, there is a competition between the speed of data fetching and data merging. The optimal performance occurs when the data fetching and merging are well balanced. Thus, when the size of the subtree is fixed, the job execution time can benefit from a large STB buffer until it is big enough to hold all SEBs buffers. For the rest of the papers, we use 4MB STB buffers and 64KB SEB buffers unless otherwise specified.

5.2.1.2 Virtual Segments in a Subtree

The number of the virtual segments in a subtree is another factor which impacts the performance. Because it directly affects the concurrency of data merging, it is important to understand its performance implications. Figure 5.12 shows the results of our tuning experiment, in which we fix the STB size to be 4MB. As subtrees grow in size, the number of total subtrees is decreased, which consequently reduces the load of merging threads. That is to say, the merging threads are able to merge active subtrees efficiently without having to deactivate many of them. If subtrees grow further in size, one active subtree will trigger many

SEBs to fetch data from remote segments. In addition, requests from many ReduceTasks will contend for different segments from the same intermediate data file. The program execution is then affected and results in longer completion time.

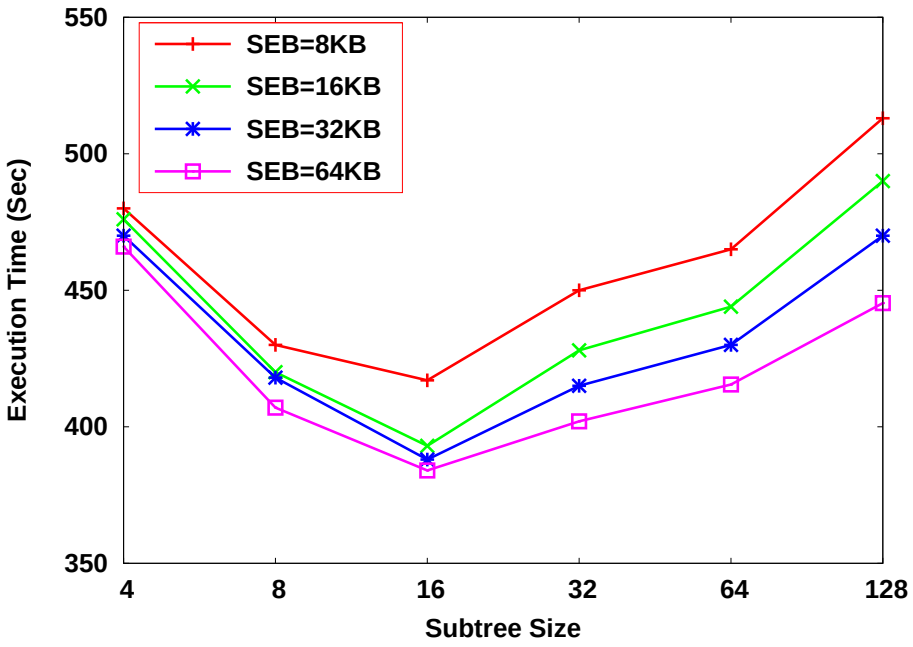


Figure 5.12: Tuning of Virtual Segments in a Subtree

Given this intriguing behavior, we speculate that there is a theoretical relationship between the subtree size and the job execution time. On one hand, smaller subtrees will lead to more of them, causing a linear increasing complexity in managing subtrees (activating and deactivating). On the other hand, virtual segments in a subtree need to be fetched remotely, thus more of them will lead to a linear increasing cost of data movement. In addition, Hadoop uses a heap-based priority queue for merging <key,val> pairs, which is inherited by virtual shuffling as well. This implies that there is a logarithmic complexity in merging SEBs into STBs and the same in merging STBs to the SGB.

To gain more insight on the relationship, we provide a simplistic analysis here. Let's assume the total number of segments is M, and the subtree size is x. The total number of subtrees is then $\frac{M}{x}$. The cost of managing these subtrees and merging their STBs can be denoted as $f_1 = O(\frac{M}{x} + log\frac{M}{x})$. The cost of fetching SEBs of a subtree and merging

them into the STB can be denoted as $f_2 = C * O(x) + O(logx)$, where C ($C > 1$) is a constant factor that represents the higher cost of fetching data remotely, compared to that of managing subtrees locally. Ideally, the merging processes conducted on STB and SEB would be completely asynchronous and be able to to achieve a pipeline without any stalls, when the two costs are the same. The lower bound for x can be solved by letting $f_1 = f_2$. Approximately, x is equal to $\sqrt[(C+1)]{M}$. With the total number of segments being 1024, the theoretical lower bound would be a number close to 16, depending on the exact value of C. As shown in Figure 5.12, this conjecture matches well with our empirical results.

5.2.2 Benefits to Job Execution

5.2.2.1 Overall Performance

We run Hadoop TeraSort benchmark with different data sizes and different numbers of nodes. Each slave runs 8 MapTasks and 4 ReduceTasks concurrently. Figure 5.13 shows the performance of Terasort on 20 nodes using different shuffling strategies, where the total amount of physical memory is 160GB.

Table 5.2: Test Case Description

Test Cases	**Shuffle Strategy**	**Transport Protocol**	**Network**
Hadoop on 10GigE	Physical Shuffling	TCP/IP	10GigE
Hadoop on IPoIB	Physical Shuffling	IPoIB	InfiniBand
Hadoop-A on RoCE	Physical Shuffling	RoCE	10GigE
Hadoop-A on RDMA	Physical Shuffling	RDMA	InfiniBand
Hadoop-A on RoCE	Virtual Shuffling	RoCE	10GigE
Hadoop-A on RDMA	Virtual Shuffling	RDMA	InfiniBand

Three different cases are included in the comparison: virtual shuffling as implemented in Hadoop-A, physical shuffling in Hadoop-A, and physical shuffling in the original Hadoop. Hadoop-A tests were run with InfiniBand RDMA (Remote Direct Memory Access) and RoCE (RDMA over Converged Ethernet) transport protocol. The original Hadoop was run with InfiniBand IPoIB and 10GigE. Because of its scalability limitation, we did not include our previous work network-levitated merge in this comparison.

As shown in the figure, physical shuffling in Hadoop-A performs slightly better than the original Hadoop for all cases. This is because the use of the high-speed RDMA protocol compared to the IPoIB and 10GigE protocol. While RDMA is beneficial to data movement on the network, the performance bottleneck lies with the disk I/O performance when physical shuffling is used. Therefore, marginal benefits are observed when the transport protocol is replaced. Among the three cases, virtual shuffling consistently performs the best and improve the overall performance by up to 27%, which demonstrates that virtual shuffling is able to speed up the data movement and boost the performance. Virtual shuffling also shows good scalability on different networks, which demonstrates consistent performance improvement compared to the physical shuffling. Because of the comparable performance, we use the original Hadoop as the representative implementation of physical shuffling for the rest of the paper and we conduct all the rest tests on InfiniBand network.

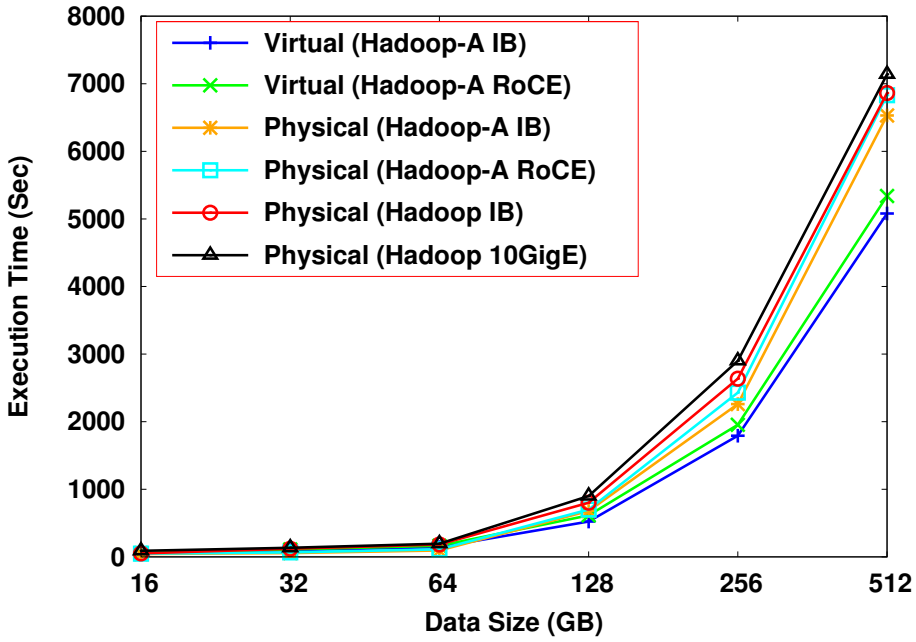


Figure 5.13: Performance Comparison of Different Shuffling Strategies

Figure 5.14 shows the performance of different shuffling strategies on seven representative benchmarks as shown in table 5.1, Hive Benchmarks (Order By, 60GB), Terasort

(512GB), and 20GB for the rest benchmarks. As shown in the figure, virtual shuffling improves the performance of most of the benchmarks but not all of them. The workload of these benchmarks can be categorized into two types. The first type is Reduce-heavy workloads which include InvertedIndex, TermVector, SequenceCount, Hive OrderBy and Terasort. For these applications, MapTasks generate a large amount of intermediate data which has to be fetched by all the ReduceTasks. This causes heavy all-to-all network traffic. As shown in Figure 5.14, virtual shuffling is able to significantly improve the performance of these benchmarks by 23.4%, 20%, 18% 54.6%, and 26%, respectively, for InvertedIndex, TermVector, SequenceCount, Hive OrderBy and Terasort.

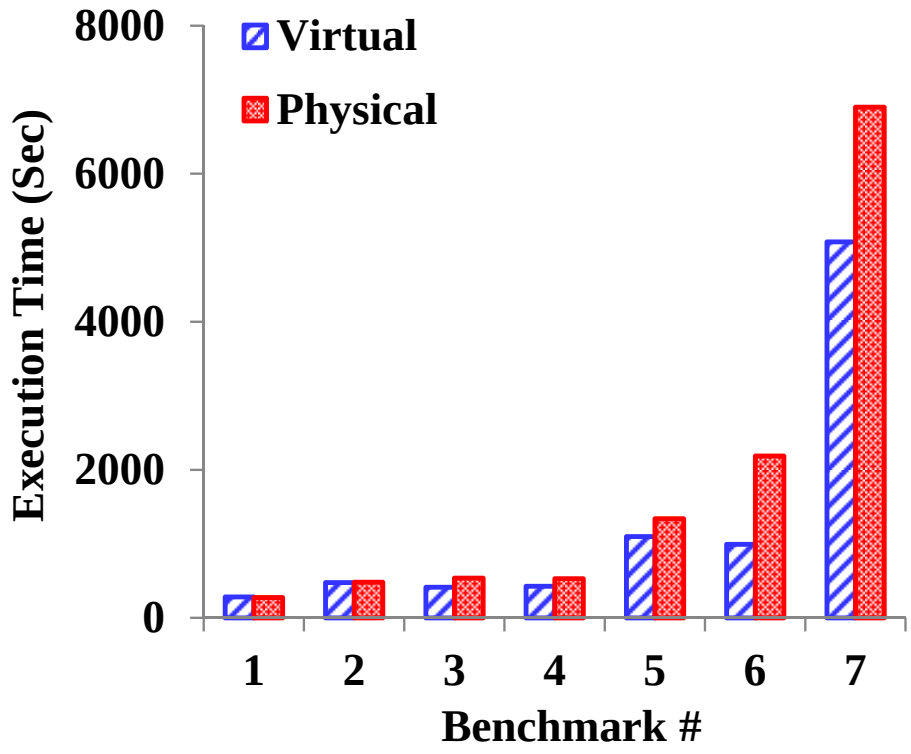


Figure 5.14: Performance of Different Benchmarks

The other type is Map-heavy workloads which include Grep and WordCount. These workloads, on the other hand, do not benefit much from virtual shuffling due to their small amount of intermediate data, which requires little data movement during data shuffling. Thus, their data shuffling phase is mostly CPU bound. Overall, these performance results indicate that virtual shuffling can significantly improve the performance of Reduce-heavy

MapReduce applications, and the benefit dwindles for the applications that do not have much intermediate data.

5.2.2.2 MapTask Improvement

In the original Hadoop Mapreduce, a job is spilt into multiple MapTasks. These mapTasks are launched in a wave manner on slave nodes. One wave can only execute limited number of MapTasks in parallel which is normally the number of cores on one node. Right after the first wave of MapTasks finished, ReduceTasks are launched and compete for the disk bandwidth with later waves of MapTasks running on the same node. For physical shuffling, this leads to severe disk contention and cascade impact to the job execution time. Through virtual shuffling, the ReduceTask elimites huge amount of disk access, which dramatically reduces the disk contention. This consequently benefits all the MapTasks running on the same node. Figure 5.15 shows the MapTasks execution time comparison between physical shuffling and virtual shuffling. The reduction of average MapTask execution is increased linearly with the input size growing and we see at most 53.8% reduction on average Map-

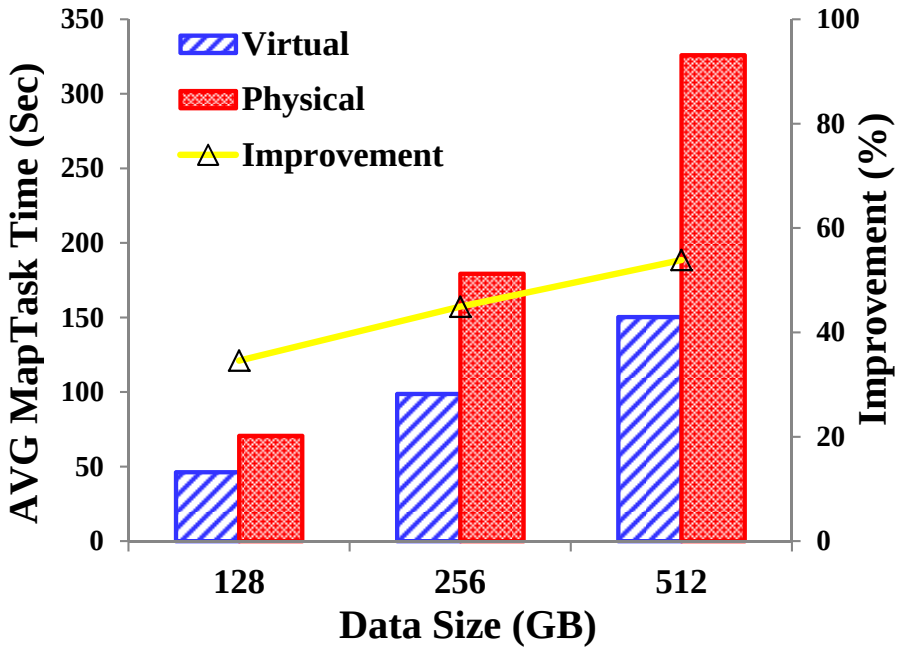
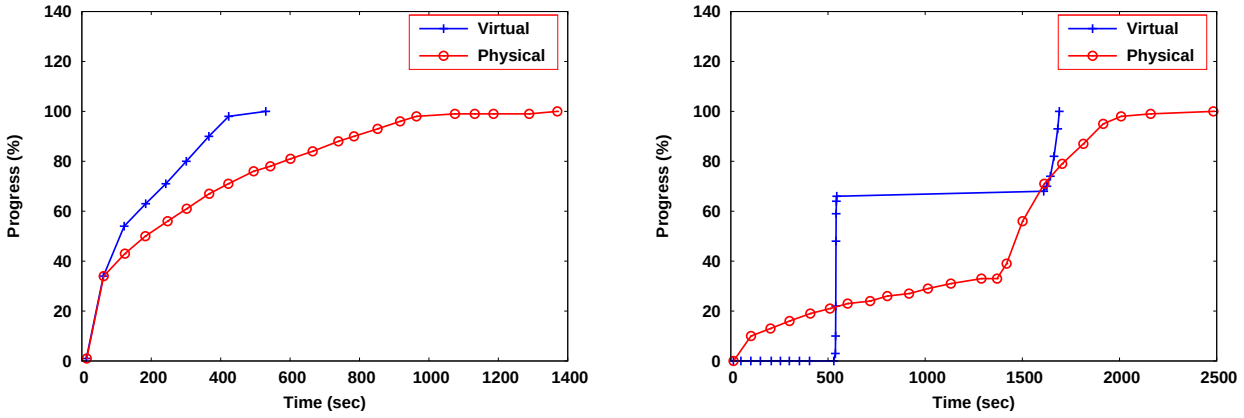


Figure 5.15: MapTask Improvement

Task execution time with the 512GB input size. The hierarchical merge enable map phase to finish much sooner without competing with co-located ReduceTasks for the scare disk bandwidth, which is also a strong indication of the potential benefits for virtual shuffling on future manycore environment.

5.2.2.3 Progress of TeraSort Execution

We compare the progress of TeraSort program execution using different shuffling strategies. The results are shown in Figure 5.16. The Y-axis shows the percentage of completion for Map and Reduce Tasks. The X-axis shows the progress of time during execution. Figure 5.16(a) shows that MapTasks of TeraSort complete much faster with virtual shuffling, especially when the percentage of completion goes over 50%. This is because MapTasks are launched as multiple waves of tasks. Right after the first wave of MapTasks finished, under the physical shuffling strategy, ReduceTasks are launched and compete for the disk bandwidth with later waves of MapTasks. This leads to severe disk contention and a cascading impact to the job execution time.



(a) MapTask (b) ReduceTask

Figure 5.16: Progress Diagrams of TeraSort

In contrast, virtual shuffling eliminates such disk contention to MapTasks, which consequently benefits the progress of MapTasks. Similarly, figure 5.16(b) shows that ReduceTasks are completed faster with virtual shuffling. Note that in the case of physical shuffling, the

progress of ReduceTasks is reported while the data are being merged. However, with virtual shuffling, we do not report progress until the completion of all MapTasks, and then near the completion of merging all segments. This is reflected in the figure as seemingly slow initial progress for virtual shuffling. Virtual shuffling actually still makes progress on ReduceTasks. Once it begins reporting, the progress in terms of percentage jumps up quickly, first at the completion of MapTasks and then at the end of merging, as indicated by the two jumps in the figure.

5.2.2.4 Scalability

Being able to leverage more nodes to process large amounts of data is an essential feature of Hadoop. We want to ensure virtual shuffling can deliver scalability in a similar manner. So we measure the total execution time of TeraSort in two scaling patterns: one with fixed amount of total data (200GB) and an increasing number of nodes, and the other with fixed data (10GB) per node and an increasing number of nodes. The aggregated throughput is calculated by dividing the total size with the program execution time.

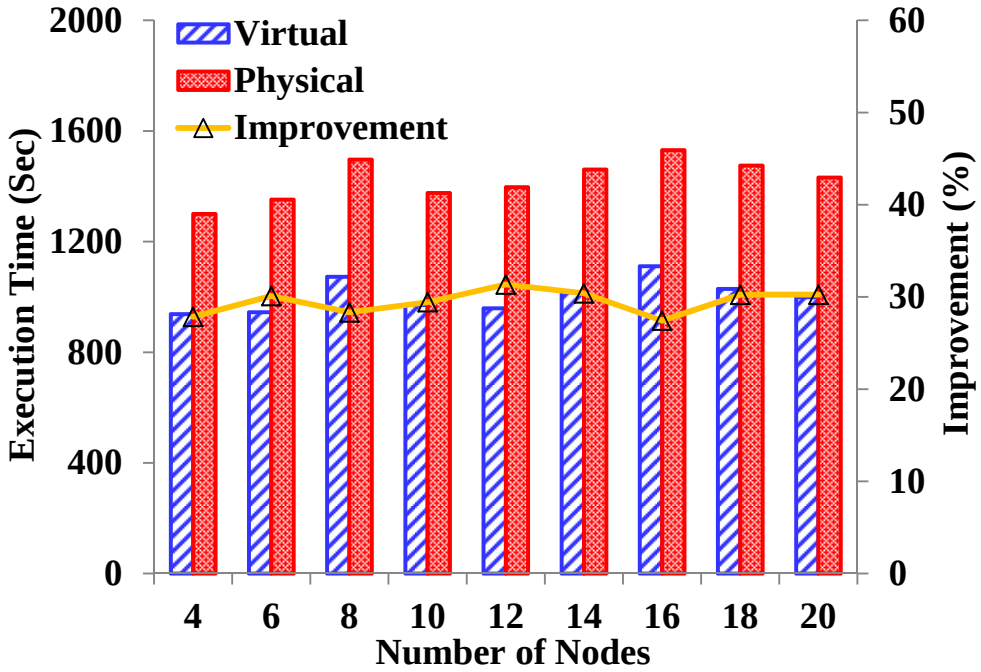


Figure 5.17: Scalability with a Fixed Data Size per Node

Figure 5.17 shows the scalability comparison between virtual shuffling and physical shuffling with a fixed data size (10GB) per node Both of them can achieve linear scalability. With the average amount of shuffled data per node is more or less similar, virtual shuffling demonstrates consistent improvement over physical shuffling. On average, virtual shuffling can speed up the execution time by approximately 30% and improvement throughput by 43%. Figure 5.18 shows the scalability comparison between virtual shuffling and physical shuffling with a fixed size of total data (200GB). Again both of them can achieve good scalability. Virtual shuffling can cut the execution time by up to 33%, compared to physical shuffling. Conversely, this results in a throughput improvement of 49.2%. Note that with the increasing of the compute node, the average amount of shuffled data per node is decreasing, which means the percentage of the shuffling time over the total execution time decreases. Hence, the virtual shuffling shows less improvement in this case. It indirectly reflects that virtua shuffling can bring more benefits for data intensive applications. To summarize, compared to physical shuffling, these results adequately demonstrate better scalability of virtual shuffling for large-scale data processing.

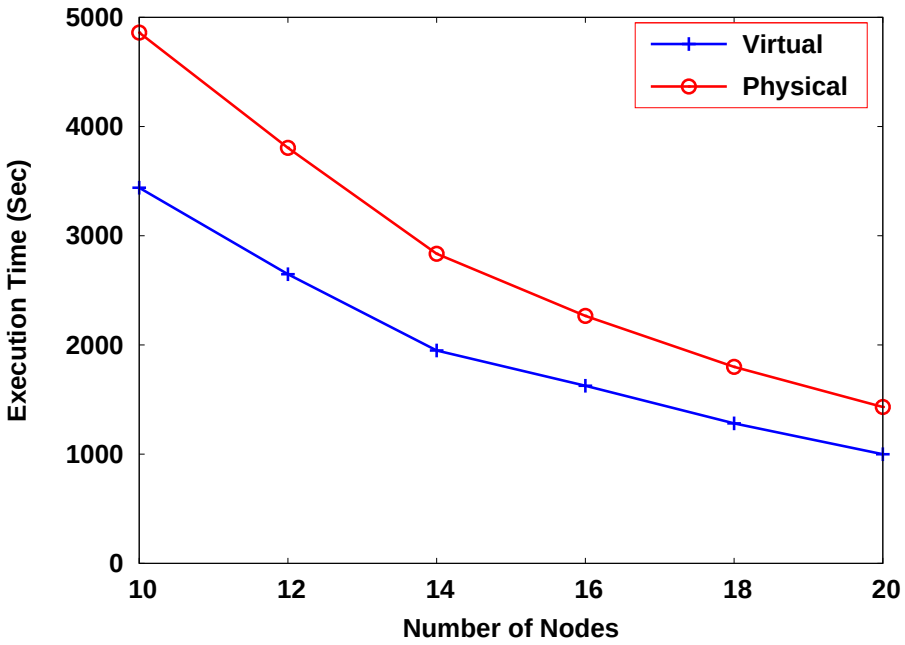


Figure 5.18: Scalability with a Fixed Data Size for the Program

5.2.3 CPU Utilization

CPU utilization is an important performance metric. Virtual shuffling is able to take the advantage of high speed interconnect, which is expected to lower the CPU utilization. In this section, we quantifies the CPU utilization benefits of virtual shuffling. We measure CPU utilization during the execution of TeraSort every 2 seconds. The percentage of CPU usage for 8 cores is recorded. We then take the average across all slaves at the same timestamp. Figure 5.19 shows the comparison of the average CPU utilization between virtual shuffling and physical shuffling. These results are from a TeraSort program on 12 slave nodes. Similar comparisons are observed for TeraSort on different number of nodes. Clearly, virtual shuffling has less CPU utilization compared to physical shuffling. Cumulatively, virtual shuffling

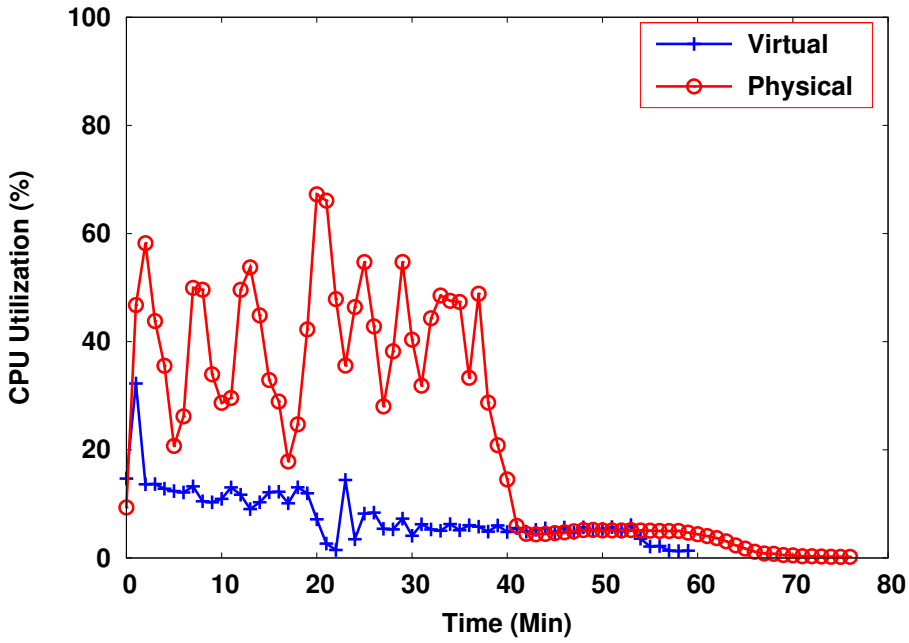


Figure 5.19: Comparison of CPU Utilization

has a CPU utilization of 18.7% at the time of its job completion, compared to 29.3% for physica shuffling. Relatively, the reduction is 36.2%. Note that virtual shuffling has higher CPU usage towards the end of its completion, during which it is running a pipeline of shuffle/merge/reduce operations. Besides, shortening the application execution time, this experiment demonstrates that virtual shuffling is able to significantly lower CPU utilization.

5.2.4 Benefits to I/O and Power Consumption

Virtual shuffling is designed to alleviate the severe disk contention problem in MapReduce infrastructure. In this section, we analyze the detailed I/O behavior of virtual shuffling and compare it to physical shuffling. The benchmark used is Terasort and we conduct the experiments on 12 slaves nodes with a fixed size of input data (250GB).

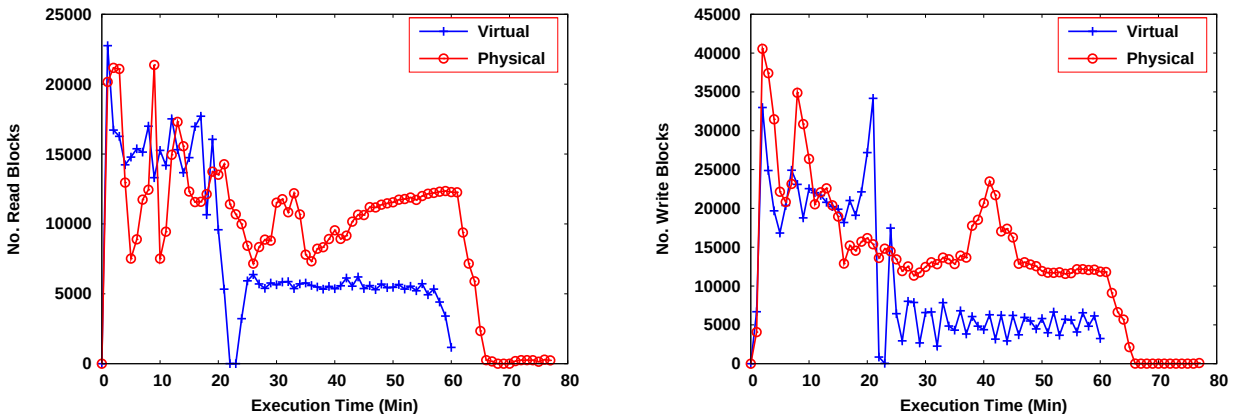
5.2.4.1 Profile of I/O Accesses

Table 5.3: I/O Blocks

	READ	WRITE	Total (Kblocks)
Virtual	31,088	40,833	71,921
Physical	45,031	64,039	109,070

We trace the *vmstat* output every second on all the slave nodes at run time. Table 5.3 shows the average total number of read and write blocks on a slave node. As you can see, our virtual shuffling significantly reduced the disk accesses for both read and write operations, by 30.9% and 36.2% respectively. The total reduction is up to 34.1%. Figure 5.20 provides the run-time profile of read and write blocks, which indirectly reflects the number of requests issued during the program execution. The more read and write blocks are issued, the more traffic is generated which essentially increases the disk contention and hurts the performance. Note that total disk read and write blocks are different because intermediate files are read and written in different ways and these I/O activities are concurrent with those to the Hadoop Distributed File System.

When disk bandwidth is a scarce resource, high disk I/O traffic can lead to long queuing time of I/O requests which essentially degrades the performance of the original Hadoop. However, virtual shuffling is able to reduce the disk I/O traffic and support efficient data movement. In order to further understand the benefits of disks accesses reduction, we analyze the service time and the wait time of I/O requests. The service time is the time taken to complete one I/O request and the wait time includes the queuing time and the service time.



(a) Read Blocks (b) Write Blocks

Figure 5.20: Run-time Profile of I/O Accesses

Figure 5.21(a) shows the details for both virtual shuffling and physical shuffling. Several I/O behaviors can be observed from this figure. First, virtual shuffling has the similar I/O service

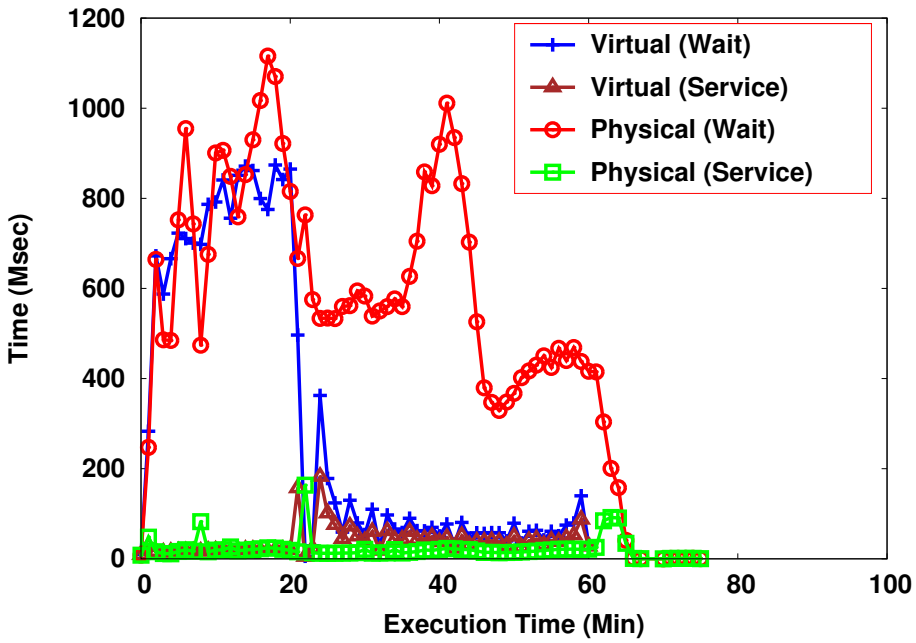


Figure 5.21: Dissection of Request Wait and Service Times

time as physical shuffling. Second, virtual shuffling leads to similar or lower I/O wait time during the first 20 minutes, which correspond to the mapping phase of the execution. As the execution progresses into the reducing phase, the I/O wait time is significantly reduced. This is because virtual shuffling significantly reduces disk accesses. Third, for physical shuffling,

especially during the reducing phase, most of the I/O requests spend more than 95% of their turnaround time waiting in the queue, which means the disk is not being able to keep up with the requests. On the contrary, I/O requests only spend around 40% of the total time waiting in the queue with virtual shuffling.

Taken together, the experiment demonstrates that virtual shuffling is able to efficiently alleviate disk contention and leave it in an efficient working status, thereby significantly reducing the execution time.

5.2.4.2 Power Consumption

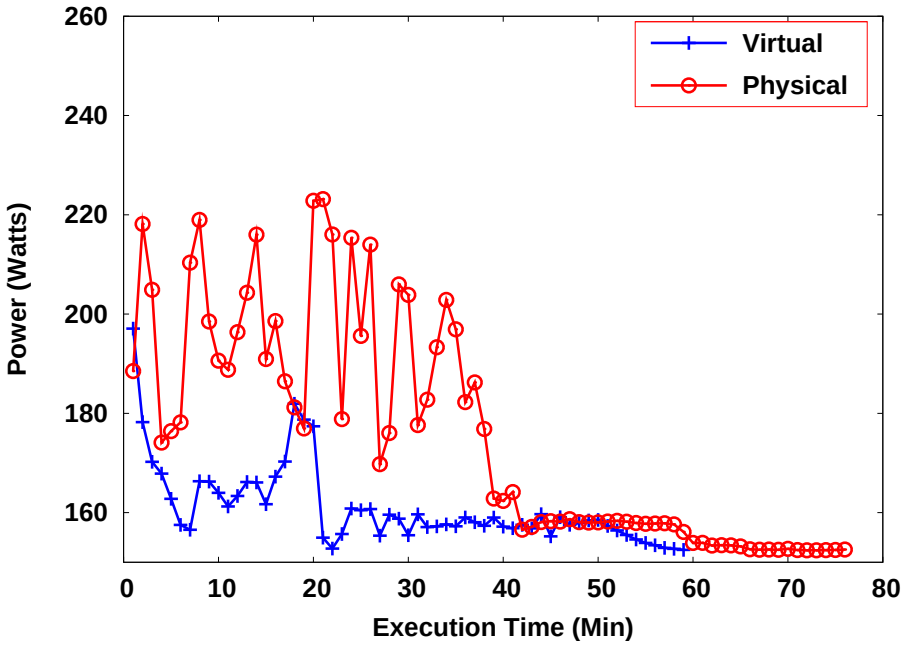


Figure 5.22: Comparison of Power Consumption

To examine the energy implication of virtual shuffling, we attach WattsUp PRO/ES power meters to several compute nodes and measure their power consumptions at a per-second interval. WattsUp meters have a simple serial-USB interface that allows us to record the power profile of MapReduce programs in a fine-grained manner into a tracefile. We then plot the power profile based on the trace files. The power is recorded every second. For clarity, we plot the power consumption profile on a per-minute basis. Figure 5.22 shows

the run-time profile of the power consumption per minute. The average draw for virtual shuffling was 160 watts with a standard deviation of about 8 watts, while 180 watts on average with a standard deviation 23 watts for physical shuffling. Compared to physical shuffling, the average power consumption of virtual shuffling is reduced by 12%. It suggests that, by reducing disk accesses, virtual shuffling can lead to significant savings on run-time power consumption for MapReduce programs.

5.3 Performance Evaluation on Parallel Community Detection Algorithm

In this section we report the experimental evaluation of our parallel algorithm. We first examine the quality of the community detection. Then we analyze the features of our algorithm in detail. In the end, we evaluate the scalability of our parallel algorithm.

Table 5.4: Evaluation of Real World Graphs

Category	Name	Description	# Vertices	# Edges	References
Small Size Social graphs	Zachary Karate Club	Friendships of a karate club	34	78	[83]
	Dolphin Social Network	Doubtful Sound Community	62	159	[84]
Large Social Graphs	Wikipedia(EW)	Graph of the English Part of Wikipedia	4.206M	77.66M	[85, 86]
	LiveJournal (LJ)	LiveJournal Social Network	3.997M	34.68M	[87]
	Youtube (YT)	Youtube social network	1.135M	2.987M	[87]
	Amazon (AZ)	Amazon product co-purchasing network	0.335M	0.925M	[87]
	ND-Web (ND)	University of Notre Dame web-pages network	0.325M	1.497M	[88]
	DBLP (DB)	DBLP collaboration network	0.317M	1.049M	[87]
	UK-2005 (UK)	Web crawl of English sites in 2005	39.46M	936.4M	[85, 86]
Large Synthetic Graphs	R-MAT	R-MAT generated graphs	Scale from 25 to 32		[89]

Our parallel algorithm is entirely written in C and uses Pthreads. The communication runtime can leverage either the generic MPI protocol or the special SPI communication protocol on Blue Gene/Q. We test our algorithm on both a small size cluster and the 48-rack Blue Gene/Q system, Mira, from Argonne Leadership Computing Facility (ALCF). The cluster has 16 nodes and each node is equipped with two 2.67GHz hex-core Intel Xeon X5650 CPUs and 24GB memory.

5.3.1 Benchmarks and Performance Metrics

In our evaluation, we use an extensive set of graphs including both *real world social graphs* and *synthetic graphs* as listed in Table 5.4. The real world graphs covers both small

and large size social graphs. The small social graphs include Zachary Karate Club Graph and Dolphin Social Network. The large social graphs include friendship network, collaboration network and shopping network such as Wikipedia, LiveJournal, Youtube, Amazon, ND-Web, DBLP, and UK-2005. The size of these graphs varies from hundreds of thousands of vertices with several million of edges to hundreds of millions vertices with to nearly 1 billion edges. These real world graphs not only have good community structures but also tend to be both *small world* and *scale free* with low effective diameter and power law degree distribution [90, 91, 92]. The third type of graphs are synthetic graphs. We generate these graphs using the massive random graph generator R-MAT [89]. The graphs generated by R-MAT usually have very low modularity and do not have good community structure.

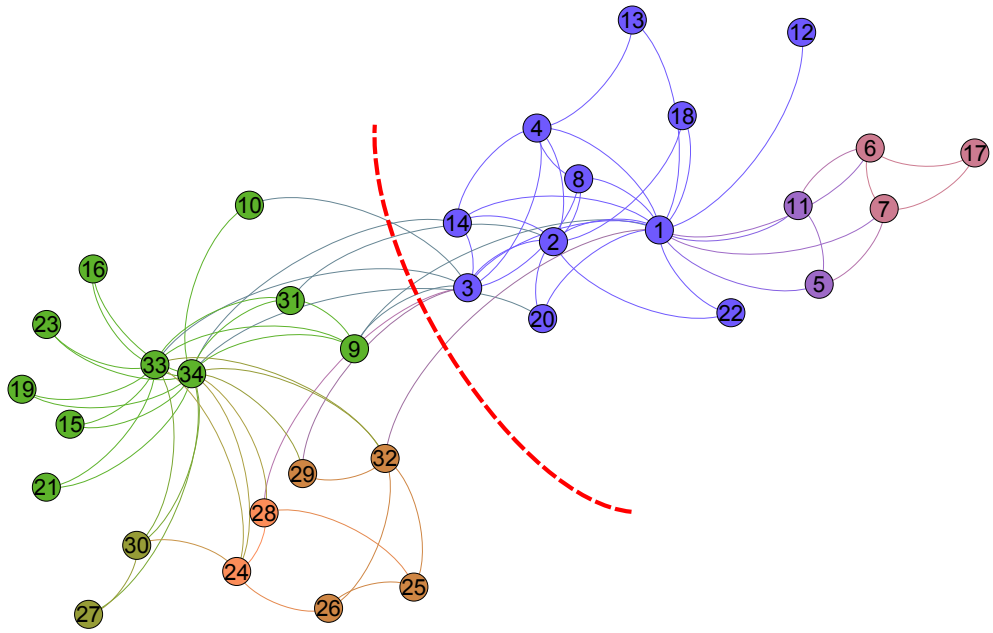
To quantify the quality and scalability of our parallel algorithm, we use several metrics including *Modularity*, *Evolution Ratio*, and *Traversed Edges Per Second (TEPS)*. We also use *Number of Hashed Entries*, *Average Bin Length*, *Maximum Bin Length* and *message rate* to analyze the internal features of our algorithm.

5.3.2 Community Quality Analysis

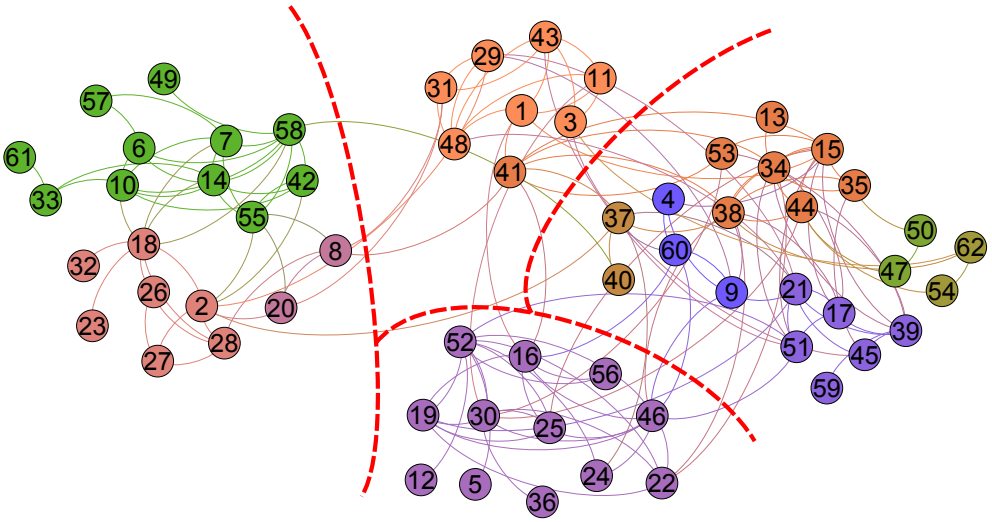
We compared our parallel algorithm to sequential algorithm on both modularity and evolution ratio. The *evolution ratio* is defined as the percentage of the detected communities divided by the size of the original graph. These tests are conducted on the small size cluster. For both parallel algorithm and sequential algorithm, we run the experiments multiple times and the results are consistent.

5.3.2.1 Real World Small Graphs

We use two small social graphs Zachary Karate Club [83] and Dolphin [84] social network to examine the accuracy of our parallel algorithm. Figure 5.23 shows the resulting graphs with our algorithm. The red dashed line splits the ground truth communities in these two graphs, which have been detected by many other algorithm [93, 94, 17]. The vertices with



(a) Zachary Karate Club Graph

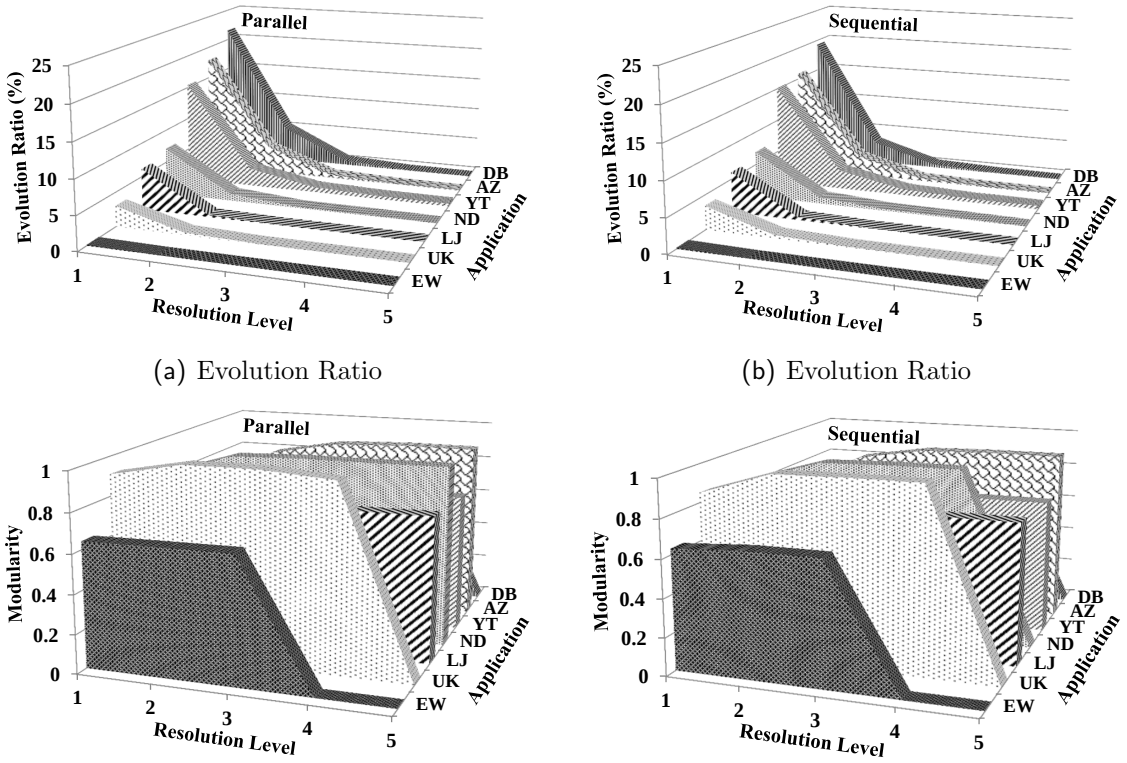


(b) Dolphin Social Network

Figure 5.23: Result on Small Graphs

different colors represent the communities detected by our parallel algorithm. It is clear that our parallel algorithm is able to detect the ground truth communities for both graphs. More importantly, it is able to extract fine-grained community structures. The resulting modularity is 0.41 and 0.51 for Karate Club graph and Dolphin social network, respectively.

These modularity numbers are similar to the results (0.41 and 0.495 respectively for the two) reported in the literature [93, 94, 17].



(a) Evolution Ratio

(b) Evolution Ratio

(c) Modularity

(d) Modularity

Figure 5.24: Convergence and Detection Quality with Social Networks

5.3.2.2 Real World large Social Graphs

We also use real world large social graphs to examine the quality of our algorithm in terms of both *evolution ratio* and *modularity*. The results are shown in Figure 5.24 and Figure 5.25. Figure 5.24 compares the convergence property of our algorithm to the sequential Louvain algorithm, where the X-axis is the resolution level, the Y-axis corresponds to the applications. In (a) and (b), the Z-axis is the *evolution ratio* and, in (c) and (d), the Z-axis is the modularity. Figure 5.25 compares the corresponding modularity for these applications on the first level. Several important observations can be made from these figures. First, the social graph size is reduced significantly at the first level and different

social graphs show different graph evolution trend. For LiveJournal, ND-Web, Wikipedia, and UK-2005, more than 94% vertices are merged into communities at the first level, while for other networks, only 80% of vertices are merged. Second, both algorithms are able to unfold the hierarchical community structures up to five levels. This means that the parallel algorithm is able to capture the same important behavior as the sequential algorithm. Third, our parallel algorithm is able to achieve similar convergence property in both *modularity* and *evolution ratio*. Note that the first level output is normally the most informative. Figure 5.25 shows that the modularity achieved by the parallel algorithm has little difference from the sequential algorithm. Finally, with the resolution on the hierarchical community structure of the social graphs, the graphs evolve in a way that increases the corresponding modularity. This evolution is meaningful only if the modularity does not decrease. Because good modularity implies the good community structure. Our parallel algorithm is able to preserve this feature and achieve high modularity. Taken together, these experimental results on real world social graphs adequately demonstrate the high quality and good convergence of our parallel algorithm.

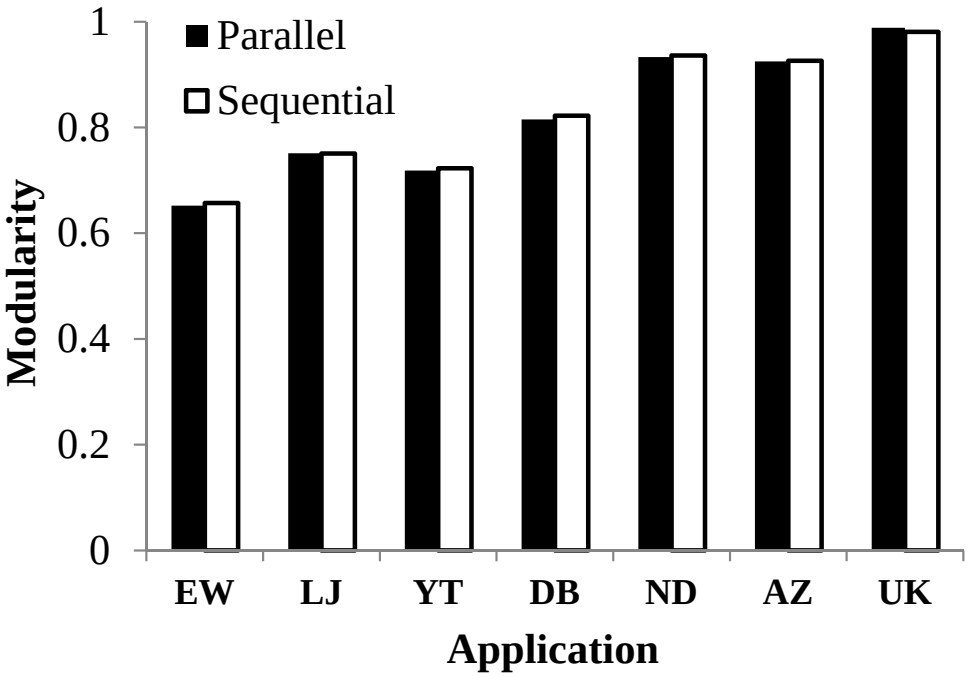
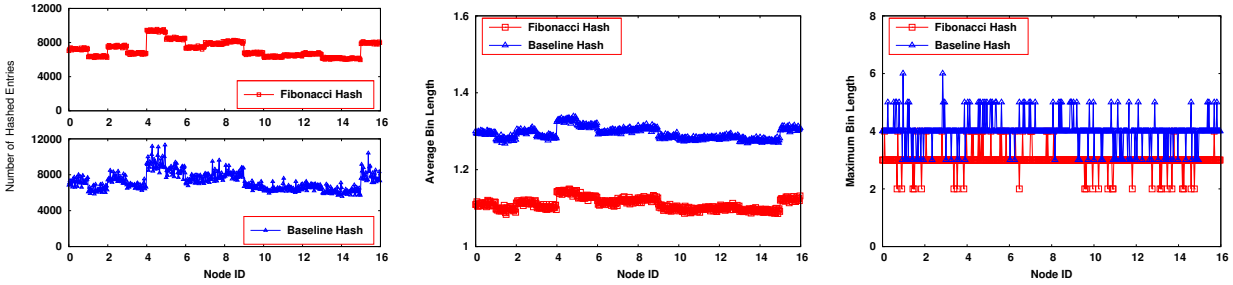


Figure 5.25: Modularity Comparison on Social Networks

5.3.3 Hash Behavior Analysis



(a) Hashed Entries (b) Average Bin Length (c) Maximum Bin Length

Figure 5.26: Profiling Comparison on Hash Policies

Our algorithm make use of a hashing scheme to partition data instead of the traditional sorting based technique. It is important to understand the detailed behavior of hash functions. To this end, we analyze our algorithm in terms of the impact of hash functions on load balancing and the expansion factor which are important for parallel graph algorithms over distributed memory systems.

5.3.3.1 Hash-based Load Balancing

Our parallel algorithm uses two hash operations, namely sequential scan and insert. To take advantage of this hash based implementation, a carefully design on the hash table is needed for good performance. The first consideration is load balancing. The hash table is created on each compute node and will be managed by all the worker threads on that node. We have examined a set of hash functions including *concatenated hash*, *linear congruential hash*, and *bitwise hash*. Finally we choose Fibonacci hash, which achieves the best load balancing.

Figure 5.26 shows the comparison on different hash policies, where we examined the hash quality through simulation tests over 16 compute nodes with 32 threads concurrently running on each node. The synthetic graphs generated by R-MAT are used as input. We have tested multiple times with various graphs and only show one set of result here. To be concise, we use *Baseline hash* to denote the best of hash policies other than Fibonacci hash.

The workload of the threads is showing in Figure 5.26(a). Clearly, Fibonacci hash balances the workload very well across all compute nodes. The other hash functions cannot balance the workloads. Different compute nodes usually have different number of hashed edges. This is due to the vertex based partition.

Figure 5.26(b) shows the comparison on the average length of the hashed bins and Figure 5.26(c) shows the comparison on the maximum length of the hashed bins. Fibonacci hash achieves an average bin length of 1.12 and a maximum bin length of 3. But the *Baseline hash* has an average bin length of 1.35 and a maximum length of 6. From these simulation analysis, we demonstrate that Fibonacci hash provides superior quality over other hash policies.

5.3.3.2 Impact of the Expansion Factor

The expansion factor is an important parameter for the hash table and tightly related with memory consumption. A large expansion factor reduces the chance of collision with more memory holding the buckets. A small expansion factor increases the chance of the collision because there is less room to hold buckets. We study the relationship between the expansion factor and the chance of the collision by simulation. We use RMAT-generated synthetic graphs as input and simulate 16 compute nodes with 32 threads per node. We set the expansion factor as 1X, 2X, 4X, and 8X of the size of total hashed edges per node. Figure 5.27 shows the statistic of average bin length of each thread. With the 1X expansion factor, the corresponding average bin length is over 1.1, which implies lower hit ratio for the insert operation. The average bin length is nearly 1 when the expansion factor reaches 8X, which leads to high hit ratio. This experiment provides insights on how to configure the expansion factor in the real parallel environment. When memory is available, high expansion factors should be chosen for good performance. In the rest of experiments, we choose the expansion factor as 4X because of its tradeoff on the average bin length and 50% less memory consumption compared to an expansion factor of 8X.

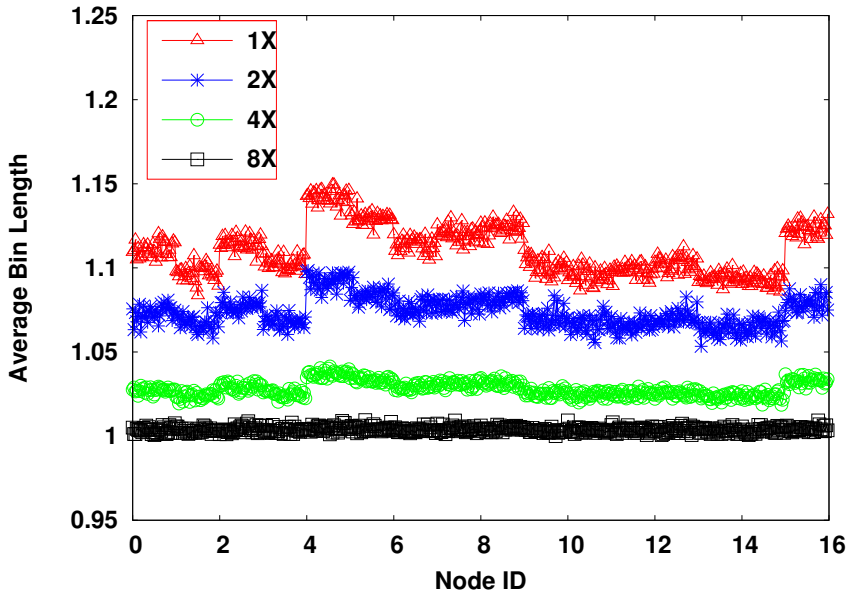


Figure 5.27: Impact of Expansion Factor

5.3.4 Message Rate by the Runtime

We examine the actual message rate supported by our communication runtime on Blue Gene/Q. The test is conducted on 128 nodes using the synthetic RMAT graphs with problem sizes from 26 to 28. Figure 5.28 shows the message rate and the corresponding execution time for a single node with various problem sizes. The message unit is 16 bytes. We vary the number of threads per node from 4 to 64. The execution time is measured as a combination of the time for scanning all the entries the hash table on sending side, the time on the network, and the time of inserting the received entries in the hash table on the receiving side. We collect the measurements across all nodes and observe a balanced distribution of the execution time. This also indicates that our load balancing technique is effective. We present the average message rate from all the nodes. With a fixed problem scale, the message rate increase with an increasing number of threads per node leads to the increasing of the message rate, which is proportional to the *number of nodes*. This is a strong evidence of the efficiency of our communication runtime. Since more threads will lead to more contention on the receiving side due to the concurrent updates in the hash table. Our runtime is able to alleviate such contention by using *Compare-and-Swap* based implementation. Also note

that the message rate decreases with an increasing problem size. This is because, with bigger problems, the number of edges on each node increases, which require more time for the scan and hash operations. In theory, the message rate will never decrease to zero with the increasing problem size, because the message rate still depends on the speed of hash operations.

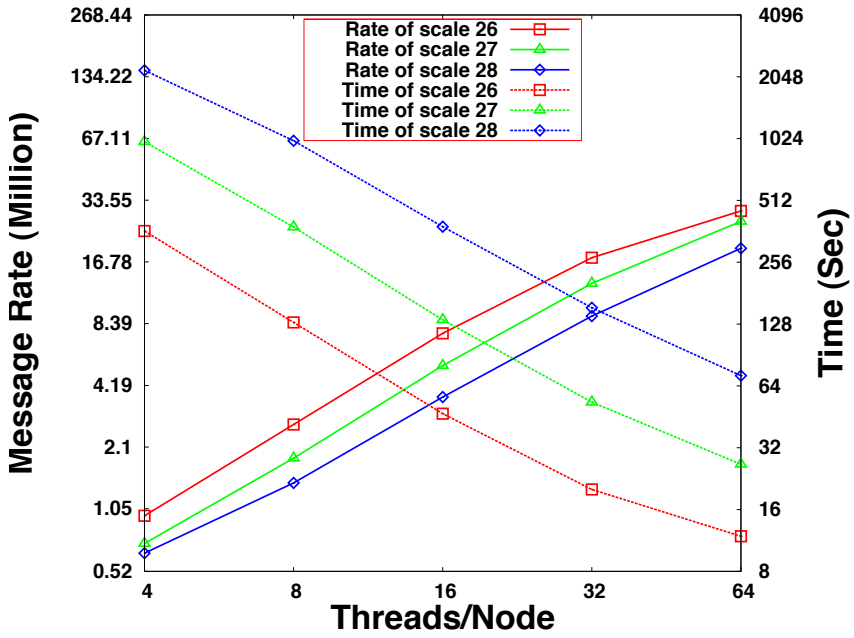
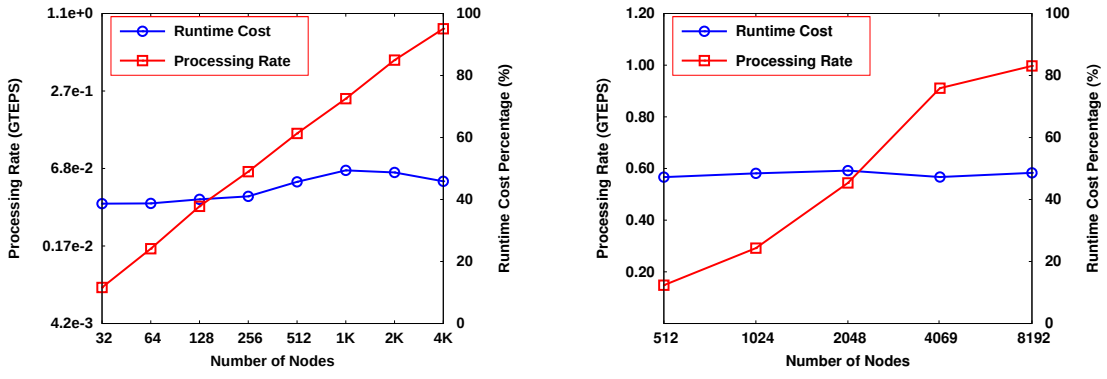


Figure 5.28: Message Rate

5.3.5 Scalability Analysis



(a) Weak Scaling (b) Strong Scaling

Figure 5.29: Scalability Test

We also evaluate the scalability of our parallel algorithm using RMAT-generated graphs. Traversed Edges Per Second (TEPS) is often used as a performance metric in Graph 500, which is calculated by the input edges and the execution time. With our parallel algorithm, the graph is shrinking and the first level generates most of informative community structures. This is different from Breadth First Search (BFS) in Graph 500, Thus we calculate TEPS by dividing the input edges by the execution time it takes to finish the first level.

We evaluate the weak scaling trend of our algorithm, for which we fix the graph with 2^{20} vertices and 2^{24} undirected edges per node and increase the number of nodes from 32 to 4096. Figure 5.29(a) shows the performance trend of weak scaling. Our algorithm achieves very good scalability with an increasing system size. The processing rate is proportional to the *number of nodes*. We also evaluate the strong scaling trend. We fix the problem size to be 30 and increase the number of nodes from 512 to 8192. As shown in Figure 5.29 (b), our algorithm demonstrates very good scalability with an increasing number of nodes. Note that these numbers are much smaller than that reported in Graph 500. The reason is that the community detection algorithm is different from the BFS and the computation complexity and communication are different. In addition, we are able to run a problem scale 32 on 8192 Blue Gene/Q nodes with a processing rate close to 2 GTEPS. Furthermore, we include the percentage of the runtime cost during the execution for all test cases. As shown in the figure, for both cases, the runtime cost is less than 50% of the total execution time. These results indicate that our communication runtime provides efficient communication service for our algorithm. Taken together, the scaling results show that our parallel algorithm is able to achieve good scalability for massive synthetic graphs.

Chapter 6

Conclusion

With the rapid increase of the computational power and exponential growth of the digital universe, scientific applications and data analytics still face challenges in terms of the scalability and efficiency. This dissertation investigates the opportunities for scalable and efficient fast computation and data processing. To this end, we have described HiCOO as a hierarchical Cooperation architecture for scalable communication in Global Address Space programming models. HiCOO formulates a cooperative communication architecture with inter-node cooperation amongst multiple nodes (a.k.a multinode) and hierarchical cooperation among multinodes that are arranged in different virtual topologies. With HiCOO, we have systematically studied the resource management and contention issues in a GAS run-time system, ARMCI, on the petascale Jaguar Cray XT5 system at ORNL. We use several different virtual topologies to represent the management of communication resources in ARMCI as directed graphs, and substantiate it with two new virtual topologies, MFCG and CFCG, as well as a canonical topology Hypercube. Our extensive evaluation of all three virtual topologies demonstrates that MFCG is the best choice for HiCOO in accomplishing scalable memory usage and contention attenuation. While addressing the challenges of resource scalability and network contention, equally important is the need to maintain the performance of GAS programming models. We show that HiCOO improves ARMCI's resilience to network contention caused by transient and irregular communication patterns. At the same time, it can maintain or improve the performance of scientific applications. In addition, we have proposed virtual shuffling as a new strategy to enable efficient data movement for MapReduce programming model. Accordingly, we have designed and implemented virtual shuffling as a combination of three techniques including a three-level segment

table, on-demand merging, and dynamic and balanced merging subtrees. Our experimental results show that virtual shuffling significantly relieves the disk I/O contention problem and speeds up data movement in MapReduce programs. It also significantly reduces the power consumption. Finally, we have designed a parallel version of the Louvain algorithm for fast community detection over distributed memory systems. Along with this algorithm, we have introduced a novel hierarchical hashing scheme to organize vertices, communities, and their adjacency list, and partition their representation data across all processing nodes. Our parallel detection algorithm preserves the same convergence modularity and community properties of the original Louvain algorithm. It can scale to support graphs with up to 4 billion vertices/128 billion edges on 8,192 nodes (524,288 threads) of Blue Gene/Q.

Chapter 7

Future Work

In this Chapter, we discuss some related areas that would be natural extensions to this work. There are many interesting open problems to be explored, which we plan to investigate in our future research.

For Global Address Space programming models, the majority of our research on the runtime is performed on Cray XT platforms which uses Seastar network, To gain general acceptance, it is important for these techniques to be examined and evaluated on other types of supercomputers and physical network, e.g., 5-D torus on Blue Gene/Q [95]. Furthermore, the focus of the runtime is ARMCI and it is meaningful to explore the benefits of our techniques to other GAS runtime systems such as GASNet [96] for generic study. Besides, it is also worth to investigating large-scale applications which can leverage more memory at the application level for better performance. Last but not least, it is interesting to investigate the benefits of virtual topologies in the context of PGAS languages such as UPC [4] and Co-Array Fortran [5].

In terms of MapReduce programming model, our research mainly focus on the infiniband and 10Gigabit Ethernet. Since the high-speed computer network are tended to be widely adopted in data center. It is very meaningful to study the applicability of virtual shuffling over different network protocols such as 40 Gigabit Ethernet, 100 Gigabit Ethernet, and Sockets Direct Protocol (SDP). The characteristic of the performance may provide more valuable insights for the performance factors. On the second place, our study of the virtual shuffling focuses on the benchmarks mainly for simple data analytics. Thus it is worth investigating virtual shuffling for more commercial and scientific workloads. Furthermore,

it is meaningful to examine the virtual shuffling on large-scale commercial cloud computing systems such as EC2 from Amazon.

For parallel community detection algorithm, we have introduced a novel hierarchical hashing scheme to organize vertices, and their adjacency list, and partition their representation data across all processing nodes. It is worth applying our technique for other graph computation kernels such as breadth first search, single source shortest path, and connectivity. On the second place, it is meaningful to examine the quality of our parallel community algorithm. The third direction is to further abstract and optimize our portable runtime system and provide a more generic interface which can serve the communication for many parallel graph algorithms.

Bibliography

[1] IDC, http://www.emc.com/leadership/digital-universe/iview/executive-summary-a-universe-of.htm.

[2] Big Data, http://www.idc.com/getdoc.jsp?containerId=233485.

[3] Top 500 supercomputing sites, http://www.top500.org/.

[4] Upc specifications, v1.2, http://www.gwu.edu/∼upc/publications/LBNL-59208.pdf.

[5] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, A multi-platform co-array fortran compiler, 2004, pp. 29–40.

[6] Global arrays toolkit, http://www.emsl.pnl.gov/docs/global.

[7] Report on experimental language X10, http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf (2008).

[8] A. Shet, V. Tipparaju, R. Harrison, Asynchronous programming in upc: A case study and potential for improvement, in: Workshop on Asynchrony in the PGAS Programming Model Collocated with ICS 2009, 2009.

[9] J. Nieplocha, V. Tipparaju, M. Krishnan, D. K. Panda, High Performance Remote Memory Access Communication: The Armci Approach, International Journal of High Performance Computing Applications 20 (2) (2006) 233–253.
URL `http://hpc.sagepub.com/cgi/content/abstract/20/2/233`

[10] V. Tipparaju, E. Apra, W. Yu, J. S. Vetter, Enabling a highly-scalable global address space model for petascale computing, in: Computing Frontiers '09, 2010.

[11] Petabyte, http://www.tech-faq.com/petabyte.html.

[12] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Sixth Symp. on Operating System Design and Implementation (OSDI) (2004) 137–150.

[13] Apache Hadoop Project, http://hadoop.apache.org/.

[14] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10.

[15] S. Fortunato, Community detection in graphs, Physics Reports 486 (3-5) (2010) 75 – 174.

[16] V. Blondel, J. Guillaume, R. Lambiotte, E. Mech, Fast unfolding of communities in large networks, J. Stat. Mech (2008) P10008.

[17] A. Clauset, M. E. Newman, C. Moore, Finding community structure in very large networks., Phys Rev E Stat Nonlin Soft Matter Phys 70 (6 Pt 2) (2004) 066111, automatic medline import.

[18] L. Danon, A. Díaz-Guilera, A. Arenas, The effect of size heterogeneity on community identification in complex networks, Journal of Statistical Mechanics 2006 (11) (2006) P11010.

[19] H. Du, M. W. Feldman, S. Li, X. Jin, An algorithm for detecting community structure of social networks based on prior knowledge and modularity: Research articles, Complex. 12 (3) (2007) 53–60.

[20] A. Noack, R. Rotta, Multi-level algorithms for modularity clustering, in: Proceedings of the 8th International Symposium on Experimental Algorithms, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 257–268.

[21] C. P. Massen, J. P. K. Doye, Identifying communities within energy landscapesarXiv:cond-mat/0412469.

[22] A. Medus, G. Acuña, C. O. Dorso, Detection of community structures in networks via global optimization, Physica A: Statistical Mechanics and its Applications 358 (2-4) (2005) 593–604.

[23] J. Duch, A. Arenas, Community detection in complex networks using extremal optimization, Physical Review E 72 (2005) 027104.

[24] S. Lehmann, L. K. Hansen, Deterministic modularity optimization, European Physical Journal B.

[25] M. E. J. Newman, Modularity and community structure in networks, Tech. Rep. physics/0602124 (Feb 2006).

[26] J. Ruan, W. Zhang, An efficient spectral algorithm for network community discovery and its applications to biological and social networks, in: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining, IEEE Computer Society, Washington, DC, USA, 2007, pp. 643–648.

[27] S. White, P. Smyth, A spectral clustering approach to finding communities in graphs (2005).

[28] W. Yu, X. Que, V. Tipparaju, J. S. Vetter, Hicoo: Hierarchical cooperation for scalable communication in global address space programming models on cray xt systems, J. Parallel Distrib. Comput. 72 (11) (2012) 1481–1492.

[29] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–16.

[30] U. Gargi, W. Lu, V. Mirrokni, S. Yoon, Large-scale community detection on youtube for topic discovery and exploration.

[31] A. Lumsdaine, D. Gregor, B. Hendrickson, J. W. Berry, Challenges in parallel graph processing, Parallel Processing Letters 17 (1) (2007) 5–20.

[32] K. Munagala, A. Ranade, I/o-complexity of graph algorithms, in: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, SODA '99, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999, pp. 687–694.

[33] M. E. J. Newman, Analysis of weighted networks, Phys. Rev. E 70 (5) (2004) 056131.

[34] M. Newman, Fast algorithm for detecting community structure in networks, Physical Review E 69.

[35] U. Brandes, D. Delling, M. Gaertler, R. Goerke, M. Hoefer, Z. Nikoloski, D. Wagner, Maximizing modularity is hard (2006).

[36] A. Arenas, J. Duch, A. Fernandez, S. Gómez, Size reduction of complex networks preserving modularity, CoRR.

[37] R. Brightwell, R. Riesen, A. B. Maccabe, Design, implementation, and performance of mpi on portals 3.0, The International Journal of High Performance Computing Applications 17 (1).

[38] W. Huang, M. J. Koop, D. K. Panda, Efficient one-copy MPI shared memory communication in Virtual Machines, in: Proceedings of the International Conference on Cluster Computing, 2008.

[39] X.-H. Sun, Y. Chen, Reevaluating amdahl's law in the multicore era, J. Parallel Distrib. Comput. 70 (2) (2010) 183–188.

[40] D. Bonachea, P. Hargrove, W. M., K. Yelick, Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt, in: CUG '09: Cray User Group Meeting, 2009.

[41] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Apra, Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, International Journal of High Performance Computing Applications 20 (2) (2006) 203–231.
URL `http://hpc.sagepub.com/cgi/content/abstract/20/2/203`

[42] J. Nieplocha, E. Apra, J. Ju, V. Tipparaju, One-sided communication on clusters with myrinet, Cluster Computing 6 (2) (2003) 115–124.

[43] J. Nieplocha, V. Tipparaju, M. Krishnan, Optimizing strided remote memory access operations on the quadrics qsnetii network interconnect, High Performance Computing and Grid in Asia Pacific Region, International Conference on 0 (2005) 28–35.

[44] V. Tipparaju, A. Kot, J. Nieplocha, M. Bruggencate, N. Chrisochoides, Evaluation of remote memory access communication on the cray xt3, 2007, pp. 1–7.

[45] F. T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, 1st Edition, MKP, 1991.

[46] J. Duato, S. Yalamanchili, L. Ni, Interconnection Networks: An Engineering Approach, The IEEE Computer Society Press, 1997.

[47] W. J. Dally, Performance analysis of k-ary n-cube interconnection networks, IEEE Trans. Comput. 39 (6) (1990) 775–785.

[48] D. K. Panda, Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms, in: HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture, IEEE Computer Society, Washington, DC, USA, 1995, p. 200.

[49] F. Petrini, M. Vanneschi, k -ary n -trees: High performance networks for massively parallel architectures, Parallel Processing Symposium, International 0 (1997) 87.

[50] W. J. Dally, C. L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, IEEE Trans. Comput. 36 (5) (1987) 547–553.

[51] J. Duato, A theory of deadlock-free adaptive multicast routing in wormhole networks, IEEE Trans. Parallel Distrib. Syst. 6 (9) (1995) 976–987.

[52] X. Lin, L. M. Ni, Deadlock-free multicast wormhole routing in multicomputer networks, SIGARCH Comput. Archit. News 19 (3) (1991) 116–125.

[53] S. Sur, M. J. Koop, D. K. Panda, High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis, in: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, New York, NY, USA, 2006, p. 105.

[54] M. J. Koop, T. Jones, D. K. Panda, Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach, in: CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2007, pp. 495–504.

[55] W.-Y. Chen, C. Iancu, K. Yelick, Communication optimizations for fine-grained upc applications, in: PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 2005, pp. 267–278.

[56] S.-g. Kim, H. Han, H. Jung, H. Eom, H. Y. Yeom, Harnessing input redundancy in a MapReduce framework, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 362–366.

[57] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears, MapReduce Online, in: 7th USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2010, pp. 312–328.

[58] J. Leverich, C. Kozyrakis, On the energy (in)efficiency of hadoop clusters, ACM SIGOPS Operating Systems Review 44 (1) (2010) 61–65.

[59] W. Lang, J. M. Patel, Energy management for mapreduce clusters, PVLDB 3 (1) (2010) 129–139.

[60] Y. Chen, A. S. Ganapathi, A. Fox, R. H. Katz, D. A. Patterson, Statistical workloads for energy efficient mapreduce, Tech. Rep. UCB/EECS-2010-6, EECS Department, University of California, Berkeley (Jan 2010).

[61] Y. Chen, A. Ganapathi, R. H. Katz, To compress or not to compress - compute vs. io tradeoffs for mapreduce energy efficiency, in: Green Networking, 2010, pp. 23–28.

[62] E. J. Riedy, H. Meyerhenke, D. Ediger, D. A. Bader, Parallel community detection for massive graphs, in: PPAM (1), 2011, pp. 286–296.

[63] E. L. Martelot, C. Hankin, Fast multi-scale community detection based on local criteria within a multi-threaded algorithm, CoRR.

[64] S. Bhowmick, S. Srinivasan, A template for parallelizing the louvain method, in: A. Mukherjee, M. Choudhury, F. Peruani, N. Ganguly, B. Mitra (Eds.), Dynamics on and of Complex Networks, Volume 2: Applications to Time-Varying Dynamical Systems, Birkhauser Verlag GmbH, 2013.

[65] Y. Zhang, J. Wang, Y. Wang, L. Zhou, Parallel community detection on large networks with propinquity dynamics, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, 2009, pp. 997–1006.

[66] S. Yang, B. Wang, H. Zhao, B. Wu, Efficient dense structure mining using mapreduce, in: Proceedings of the 2009 IEEE International Conference on Data Mining Workshops, IEEE Computer Society, Washington, DC, USA, 2009, pp. 332–337.

[67] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Sixth Symp. on Operating System Design and Implementation (OSDI) (2004) 137–150.

[68] J. Soman, A. Narang, Fast community detection algorithm with gpus and multicore architectures, in: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE Computer Society, Washington, DC, USA, 2011, pp. 568–579.

[69] U. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks., Phys Rev E Stat Nonlin Soft Matter Phys 76 (3 Pt 2) (2007) 036106.

[70] J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney, Statistical properties of community structure in large social and information networks, in: Proceedings of the 17th international conference on World Wide Web, ACM, New York, NY, USA, 2008, pp. 695–704.

[71] W. Yu, X. Que, V. Tipparaju, R. L. Graham, J. S. Vetter, Cooperative server clustering for a scalable gas model on petascale cray xt5 systems, Computer Science - R&D 25 (1-2) (2010) 57–64.

[72] X. Que, W. Yu, V. Tipparaju, J. S. Vetter, B. Wang, Network-friendly one-sided communication through multinode cooperation on petascale cray xt5 systems, in: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 352–361.

[73] W. Yu, V. Tipparaju, X. Que, J. S. Vetter, Virtual topologies for scalable resource management and contention attenuation in a global address space model on the cray xt5, in: ICPP, 2011, pp. 235–244.

[74] C. J. Glass, L. M. Ni, The turn model for adaptive routing, SIGARCH Comput. Archit. News 20 (2) (1992) 278–287.

[75] Y. Wang, X. Que, W. Yu, D. Goldenberg, D. Sehgal, Hadoop Acceleration Through Network Levitated Merge, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11, 2011.

[76] X. Que, Y. Wang, C. Xu, W. Yu, Hierarchical merge for scalable mapreduce, in: Proceedings of the 2012 workshop on Management of big data systems, MBDS '12, ACM, New York, NY, USA, 2012, pp. 1–6.

[77] D. H. Bailey, L. Dagum, E. Barszcz, H. D. Simon, Nas parallel benchmark results, in: Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 386–393.

[78] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, A. T. Wong, High performance computational chemistry: An overview of NWChem a distributed parallel application, Computer Physics Communications 128 (1-2) (2000) 260–283.

[79] T. H. Dunning, K. A. Peterson, D. E. Woon, A. K. Wilson, Quantifying quantum chemistry, in: American Conference on Theoretical Chemistry, 1999, unpublished.

[80] Open Fabrics AllianceHttp://www.openfabrics.org.

[81] F. Ahmad, S. T. Chakradhar, A. Raghunathan, T. N. Vijaykumar, Tarazu: optimizing mapreduce on heterogeneous clusters, in: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'12, ACM, New York, NY, USA, 2012, pp. 61–74.

[82] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Z. 0002, S. Anthony, H. Liu, R. Murthy, Hive - a petabyte scale data warehouse using hadoop, in: ICDE, 2010, pp. 996–1005.

[83] W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33 (1977) 452–473.

[84] D. Lusseau, The emergent properties of a dolphin social network., Proc Biol Sci 270 Suppl 2.

[85] P. Boldi, S. Vigna, The WebGraph framework I: Compression techniques, in: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004), ACM Press, Manhattan, USA, 2004, pp. 595–601.

[86] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: Proceedings of the 20th international conference on World Wide Web, ACM Press, 2011.

[87] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, CoRR.

[88] R. Albert, H. Jeong, A.-L. Barabási, The diameter of the world wide web, CoRR cond-mat/9907038.

[89] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A recursive model for graph mining, in: Proc. 4th SIAM Intl. Conf. on Data Mining (SDM'04), Lake Buena Vista, FL, 2004.

[90] H. Kwak, C. Lee, H. Park, S. Moon, What is Twitter, a social network or a news media?, in: WWW '10: Proceedings of the 19th international conference on World wide web, ACM, New York, NY, USA, 2010, pp. 591–600.

[91] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.

[92] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 12:1–12:10.

[93] K. Wakita, T. Tsurumi, Finding community structure in mega-scale social networks: [extended abstract], in: Proceedings of the 16th international conference on World Wide Web, ACM, New York, NY, USA, 2007, pp. 1275–1276.

[94] P. Pons, M. Latapy, Computing communities in large networks using random walks, J. of Graph Alg. and App. bf 10 (2004) 284–293.

[95] sequoia, https://asc.llnl.gov/publications/Sequoia2012.pdf.

[96] D. Bonachea, C. Bell, P. Hargrove, M. Welcome, GASNet 2: An Alternative High-Performance Communication Interface (Nov. 2004).