Parallel Processing Framework on a P2P System Using Map and Reduce Primitives

Kyungyong Lee*, Tae Woong Choi†, Arijit Ganguly‡, David I. Wolinsky*, P.Oscar Boykin*, Renato Figueiredo*
*ACIS Lab. Department of ECE. University of Florida
E-mail: {klee, davidiw, boykin, renato}@acis.ufl.edu
†Samsung SDS, Seoul, South Korea
E-mail: taewoong.choi@samsung.com
‡Amazon Web Service. Amazon.com
E-mail: arijit@amazon.com

***Abstract*—This paper presents a parallel processing framework for structured Peer-To-Peer (P2P) networks. A parallel processing task is expressed using Map and Reduce primitives inspired by functional programming models. The Map and Reduce tasks are distributed to a subset of nodes within a P2P network for execution by using a self-organizing multicast tree. The distribution latency cost of multicast method is $O(log(N))$, where N is a number of target nodes for task processing. Each node getting a task performs the Map task, and the task result is summarized and aggregated in a distributed fashion at each node of the multicast tree during the Reduce task. We have implemented this framework on the Brunet P2P system, and the system currently supports predefined Map and Reduce tasks or tasks inserted through Remote Procedure Call (RPC) invocations. A simulation result demonstrates the scalability and efficiency of our parallel processing framework. An experiment result on PlanetLab which performs a distributed K-Means clustering to gather statistics of connection latencies among P2P nodes shows the applicability of our system in applications such as monitoring overlay networks.**

***Keywords*-Parallel processing, Map, Reduce, P2P, Monitoring, Distributed data mining**

I. Introduction

In recent years, Peer-To-Peer (P2P) systems have received considerable attention from industry and academia. Different from the traditional client-server architecture, each peer in a P2P system participates in a virtual overlay network while acting as a client and a server. Without a central server, in a P2P network, each peer is responsible for providing and retrieving information and services to and from other peers in the overlay network. In spite of the growing popularity of P2P networks, use cases of P2P systems are limited to file-sharing applications (e.g., Gnutella, KaZaa, and BitTorrent) and VoIP solutions (e.g., Skype).

Map and Reduce primitives are popular paradigms in functional programming languages. LISP defines Map as a function which applies to successive sets of input data. Reduce is defined as a function that combines input elements of sequence or aggregates results from those elements. Erlang and Python use Map and Reduce functions similarly to LISP. Hadoop [1] and Google-MapReduce [2] also use Map and Reduce concepts for their large data processing jobs. Different from the others, Hadoop and Google MapReduce apply Map and Reduce primitives to distributed computing environments freeing users from parallel job distributions and handling failed nodes.

In this paper we present a decentralized parallel processing framework which uses Map and Reduce primitives on a structured P2P network for applications such as network status monitoring, resource discovery, and distributed data mining. Without a central broker node, our system relies on a self-organizing multicast tree for an efficient task distribution. Map functions are performed at each node in a multicast tree with input data. In parallel to Map task execution, the task is propagated to child nodes in a multicast tree to reach all leaf nodes. Once leaf nodes have computed their Map functions, the results are communicated up the tree. As the results are propagated up the tree, aggregation and summarization happen at each intermediate node. Those nodes execute Reduce function over the results obtained from their child nodes and local Map function.

We have implemented our parallel processing framework on a structured P2P framework, Brunet. Brunet implements Symphony [3], a 1-D Kleinberg small-world architecture [4], and we use the Brunet P2P overlay for connection management, routing, and task distribution. In order to provide an interface to define Map and Reduce tasks, the system provides not only basic Map (e.g., count function) and Reduce (e.g., add and array concatenation) tasks, but an XML-RPC interface to register user-defined Map and Reduce functions.

The major contributions of this work are as following:

- A novel P2P parallel processing framework that uses multicast trees and Map and Reduce primitives
- Implementation and real-world deployment of our system show applicability and feasibility

The rest of this paper is organized as follows. Section II introduces Map and Reduce primitives on functional programming languages. Section III presents an architecture of our parallel processing framework. Section IV talks about use case examples of our system. Section V highlights differences between server-client based MapReduce system and our decentralized Map and Reduce based parallel processing system. Section VI evaluates our system through simulation

and a PlanetLab experiment. Section VII discusses related works. Section VIII concludes this paper.

II. Map and Reduce Primitives

Map and Reduce functions were introduced in LISP functional programming language in 1958. Currently many other languages, such as Python, Heskell, and Erlang, adopt Map and Reduce functions.

Map function is usually applied to successive sets of an input argument for recursive operation, and it is defined as:

$$\textbf{map}\ \textit{result\_type function} : \textit{input} \ \rightarrow\ \textit{result} \qquad (1)$$

In (1) *result_type* defines a type of result. The *function* is called sequentially for all the elements of the *input*. For example,

$$\textbf{map}\ 'Array'\ 'ABS()' : \{-10, 3, -4, -1\} \rightarrow \{10, 3, 4, 1\}$$

The input Map function is defined as *ABS()*, which returns an absolute value of an input integer. As this Map function is performed, an array of absolute values from input data is returned as a result.

The Reduce function usually combines all elements of input sequences using a binary operation and is defined as:

$$\textbf{reduce}\ \textit{function input\_sequence} \ \rightarrow\ \textit{result} \qquad (2)$$

In (2) *function* is a designated operation which combines the elements of *input sequence* to create *result*. For instance,

$$\textbf{reduce}\ '*'\ \ (4, 3, 6) \ \rightarrow\ 72$$

In this example, a binary operation multiply is defined as a Reduce function for the input sequence *(4, 3, 6)*. First, the input sequence is processed as (* 4 3) = 12. Then the next combining is processed as (* 12 6) = 72.

Map and Reduce functions provide powerful primitives for programming of parallel applications. Building upon these concepts, for instance, Google MapReduce [2] and Hadoop [1] are widely used for large data processing (e.g., count of URL access frequency, reverse web-link graph, and distributed grep) in a distributed computing environment.

III. Architecture

In this section, we describe our Map and Reduce style parallel processing framework architecture, whose components are composed of *Underlying P2P network, Task Distribution,* and *Map and Reduce Tasks* modules.

A. P2P Network Module

The P2P network is responsible for handling node joins and departures, connection management with neighboring nodes, routing messages, and XML-RPC interface for an user interface. The current version of our parallel processing framework is developed on top of Brunet [5], which implements Symphony, a 1-d Kleinberg's small-world network [4]. However, our system can be deployed on the structured P2P, such as Chord [6], CAN [7], or Pastry [8].

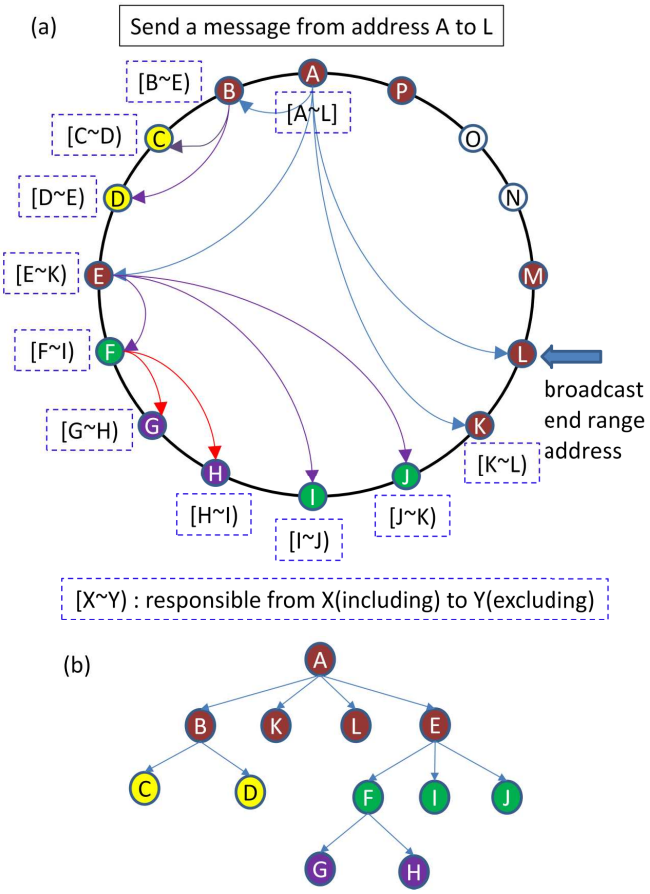

Figure 1: Bounded-broadcast task propagation (a) Node A distributes a task from node A to L. Node A sends the task to its neighboring nodes, B, E, K, and L after setting broadcast region appropriately. The task recipient nodes send the task to neighbors within the allocated region recursively. (b) Generated bounded-broadcast tree after distributing the task.

B. Task Distribution

In a distributed computing environment, task distributions or job scheduling are vital procedures for an efficient resource usage. Most compute power sharing frameworks rely on a master node, which maintains the entire view of resources currently deployed in the system. This master node-client approach often imposes significant administrative costs and scalability constraints: the server needs to be closely managed, and the failure of the master node can render all its resources unusable. In order to overcome these shortcomings, our decentralized parallel processing framework does not rely on a master node for task distribution. Instead, a bounded-broadcast [9] that uses a self-organizing multicast tree is in charge of task distribution.

Bounded-broadcast spreads a message over a sub-region of a P2P network by self-organizing a multicast tree using a Structured P2P network. It is currently implemented on top of Brunet [5]. Each Brunet node maintains two types of connections: (1) a constant number of near connections to

its nearest left and right neighbors on the P2P ring, and (2) approximately $log(N)$, shortcut connections based on the proximity neighbor selection on the P2P ring; such that the routing cost is $O(log(N))$, where N is the number of nodes.

During a broadcast a node is allocated a sub-region of the P2P ring over which to distribute tasks. The node then redistributes the task to its neighboring nodes inside its allocated sub-region while allocating new sub-regions to them. This process continues until the message is disseminated over the entire sub-region. A detailed technical description of the procedure is explained in Algorithm 1.

Algorithm 1 Bounded-Broadcast(Start, End, Message)

Require: $ConnectionList$

1: **for** i in Length(ConnectionList) **do**
2: n_start ← ADDRESS(ConnectionList[i])
3: **if** n_start ∉ [Start, End) **then**
4: continue
5: **end if**
6: n_end ← ADDRESS(ConnectionList[$i+1$])
7: **if** n_end ∉ [Start, End) **then**
8: n_end ← end
9: **end if**
10: msg←(Bounded-Broadcast, n_start, n_end, Message)
11: SEND(ConnectionList[i], msg)
12: **end for**

For example, to broadcast a task over the sub-region [A-L] in Figure 1, a message initiator issues a broadcast command to node A with sub-region information, [A-L]. Node A recognizes B, P, E, K, L, and M as neighbors. Node A then broadcasts the task to its neighbors located within [A-L], by specifying broadcast range as [B-E), [E-K), [K-L), [L] to node B, E, K, and L, respectively. After receiving the message, nodes B, E, K, and L re-broadcast the task only to their neighboring nodes inside the specified sub-region recursively. After distributing the task until the leaf node, a graph like Figure 1 (b) is formed. If a message-initiating node does not lie within the bounded-broadcast region, the message is first routed to a center node inside the bounded-broadcast region by using greedy routing.

Similar to our bounded-broadcast, Vishnevsky et. al [10] present multicast mechanisms on a structured P2P network using a routing table information. In their work, a multicast message is propagated over a subset of region by recursively partitioning allocated sub-region. They explained the algorithm on Chord [6] and Pastry [8]. Although our system is currently deployed on top of Brunet [5], our Map and Reduce based parallel processing framework can be deployed on Chord or Pastry by maintaining a current architecture design by adopting these multicast methods for task distribution.

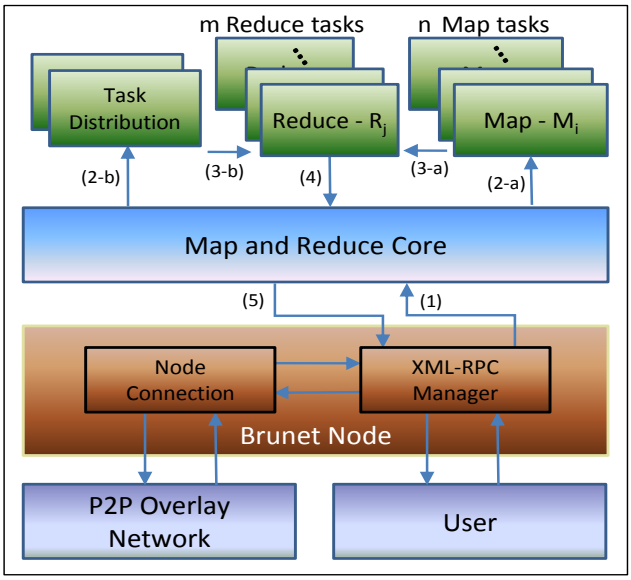

Figure 2: Map/Reduce Parallel Processing Framework Architecture

C. Map and Reduce Tasks

Nodes within a bounded-broadcast range are candidate nodes to execute Map and Reduce tasks. Every node in the range receives a name of Map and Reduce task with input arguments for each task. With the input arguments and tasks, each node executes the Map task and delivers a result of the task to the Reduce task. The Reduce task is performed with the input argument, the Map task result from itself, and the Reduce task results from child nodes. A Reduce task usually summarizes and aggregates those results and creates a new result. The summarized result is delivered to the Reduce task of parent node in the multicast tree for further aggregation. The hierarchical information aggregation method, which applies to the Reduce result accumulation, provides scalability in a distributed information management system [11] [12].

One user-favorable feature of MapReduce framework is the transparency for parallel job execution. Users do not need to concern about resource discovery, job scheduling, and handling a failed worker node. What a user needs to do is defining Map and Reduce functions associated with one's needs. Similar to Hadoop and Google MapReduce, our decentralized MapReduce framework supports user-defined Map and Reduce function by creating a class at runtime or loading a DLL that contains tasks which a user want to execute. All these actions are possible through XML-RPC interface of an underlying P2P framework. Note that users can also use pre-defined basic Map (e.g., count) and Reduce (e.g., add, array concatenation) tasks.

In our decentralized parallel processing framework, in brief, task distributions are processed at the Bounded-Broadcast module, and the tasks are determined by a user

according to one's need. Using *Task Distribution, Map*, and *Reduce* modules, our system works as shown in Figure 2.

1) A task is delivered to the Map/Reduce Core module.
2) The Core module executes a Map task M_i among n Map tasks (2-a). In parallel to executing the Map task locally, it redistributes the task using Bounded-Broadcast to the allocated sub-region (2-b).
3) Local Map task result (3-a) and child nodes' Reduce results (3-b) are delivered to the Reduce module.
4) A Reduce task R_j among m Reduce tasks is executed using results from the local Map task and child nodes. The Reduce result is returned to Core module.
5) The Reduce task result is delivered to the Reduce task of parent node using underlying XML-RPC module.

IV. Use Cases on a P2P Network

In this section, we present our parallel processing framework use case examples while showing how to define Map and Reduce functions according to a task purpose. Increasing order of Reduce task complexity, we present from a simple Map and Reduce functions to more complicated functions.

A. Counting Number of Nodes in a P2P Pool

Although a P2P system provides a self-organizing and scalable overlay network service, a lack of global status of a network limits system administrators' control over the network, and it makes system debugging challenging. This is a factor that contributes to the hesitation in adopting a P2P as an underlying overlay network for some applications [13].

Our parallel processing framework provides a scalable approach to monitor a structured P2P network. For example, consider how to measure the number of nodes in a P2P pool using our framework. At the Map task, each node creates a *<key, value>* pair whose key is *"count"* and value is an integer, 1. The Map task returns a result (*<"count", 1>* pair) to the local Reduce task. The Reduce task adds *count* values returned from the Map task of itself and child nodes. The accumulated values are returned to their parent nodes until the result reaches the root node in the multicast tree.

Figure 3 shows the overall flow of counting the number of nodes using our Map and Reduce framework. The dotted line shows task distribution using bounded-broadcast, and the thick line shows passing an aggregated Reduce result.

With regards to the job scheduling, every node receiving the task has to perform Map and Reduce tasks. Thus, nodes within a bounded-broadcast region will perform a task. By setting the broadcast region as the entire network address space, a task-initiating node can get an accurate number of nodes in a pool. A task-initiating node can reduce task distribution range and infer a number of nodes based on the task distribution range and the number of nodes within the range. This inference is possible based on the fact that node addresses of structured P2P networks are randomly selected, and nodes in a pool are uniformly distributed [5] [6] [8].

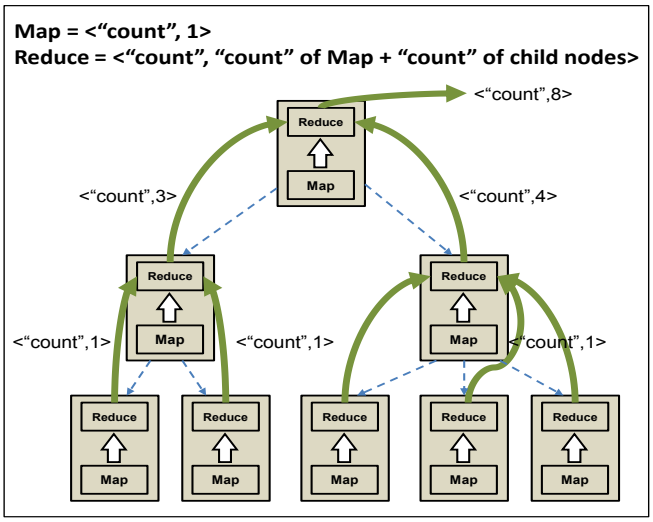

Figure 3: Counting the number of nodes in a P2P pool: Map task returns an entry <"count", 1>. Reduce task sums the "count" value from local Map task result and Reduce results from child nodes.

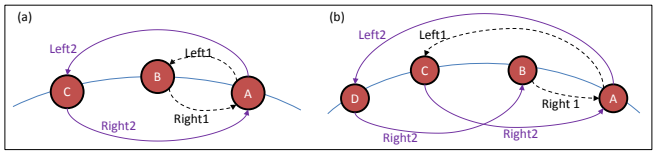

Figure 4: P2P network consistency check: (a) Node A maintains a stable connection: (NodeA.Right1 == NodeA.Right1.Left1) and (NodeA.Right2 == NodeA.Right2.Left2) (b) Node A maintains inconsistent connection: Node A is ignorant of Node B

B. P2P Pool Connection Consistency Check

In structured P2P systems, it is important to maintain consistent connection pointers among nodes to ensure correct routing. In ring-structured P2P networks (i.e., Chord, Pastry, and Brunet), a consistent connection can be defined with respect to the pointer of predecessor and successor nodes. Specifically in Brunet, for every node, we can check if its immediate left near connection node identifies it as immediate right connection node. The consistent connection in Brunet is shown in Figure 4 (a). The same rule applies for the second left and right neighboring nodes. Figure 4 (b) shows an inconsistent connection at node A. In the figure, Node A is ignorant of Node B, so Node A points Node C as its left-most neighbor and Node D as its second left neighbor. By checking connection table of Node C and D, Node A can notice that it is maintaining inconsistent connections. These steps can be applied for every node in a pool to determine overall connection consistency of a P2P pool. Based on this definition, we can measure a P2P network connection consistency using our parallel processing framework.

Algorithm 2 and 3 show Map and Reduce tasks which count the number of nodes in a pool and check connection consistency. Algorithm 2 line 1-2 initialize variables to keep *consistency* and *count* value. At line 4, the Map task gets connection list of *Left1, Left2, Right1, Right2* nodes. At line

Algorithm 2 Map()

Require: $ConnectionList$

```
1: dict["s_score"] ← 0.0
2: dict["n_count"] ← 1
3: for s in Array("Left1","Left2","Right1","Right2") do
4:    c_table[s] ← GetConnectionTable(ConnectionList[s])
5:    if c_table[s].[s̃^1] is self_address then
6:       dict["s_score"] ← dict["s_score"] + 0.25
7:    end if
8: end for
9: SEND(Local Node's Reduce Task, dict)
```

Algorithm 3 Reduce(reduce_result[], map_result)

```
1: n_count ← map_result["n_count"]
2: s_score ← map_result["s_score"]
3: for i in Length(reduce_result) do
4:    count ← reduce_result[i].["n_count"]
5:    score ← reduce_result[i].["s_score"]
6:    n_count ← n_count + count
7:    s_score ← s_score + score
8: end for
9: dict["n_count"] ← n_count
10: dict["s_score"] ← s_score
11: SEND(Parent Node's Reduce task, dict)
```

5 and 6, it checks whether it is pointing correct left or right neighbor. If it is, *s_score* is increased by 0.25. Finally the dictionary is returned to the Reduce task of local node.

Algorithm 3 line 1 and 2 initialize *n_count* and *s_score* values with the Map task result from itself. Those two values keep the number of nodes and consistency score, respectively. Line 3 - 7 iterate over Reduce results returned from child nodes. At the step, node counts and consistency scores are accumulated. Those summed values are delivered to the Reduce task of the parent node.

C. Distributed Data-Mining in a P2P Network

Data mining is a process of extracting patterns or useful models from large datasets. A survey of parallel and distributed data mining algorithms is presented at [14]. Many of those algorithms are applicable in our decentralized parallel processing framework by carefully defining Map and Reduce function. Among them, we are going to cover K-means clustering algorithms [15] and will show how it can be applied to our framework. K-mean clustering algorithm is aim at partitioning N datasets into K disjoint clusters minimizing the sum of squared error function:

$$\sum_{j=1}^{k}\sum_{i=1}^{n}|x_i - c_j|^2$$

x_i means the value of each data point, and c_j means centroid of each cluster. To achieve the minimal squared error between data point and centroid in a cluster, the algorithm first places K centroid points which represent each cluster. Each data point joins a cluster which has the closest centroid value. After all data points join clusters, new centroids are calculated within a cluster. This step is repeated until centroids do not change.

We apply K-means clustering algorithm to our Map and Reduce style parallel processing framework to get a statistic of connection latency among nodes in a P2P pool. At the task, the Map function is defined as measuring communication latency between neighboring nodes (i.e., structured near and shortcut connections). The latency information is aggregated at the Reduce task using K-Means clustering algorithm. In Section VI-B, we present a statistic of clustered shortcut and near connection latency among P2P nodes on PlanetLab using K-means clustering algorithm. Note that other distributed data mining algorithms (e.g., classification, association, and regression) can be also applied at Reduce phase.

D. P2P-Grid Computing Resource Discovery

A resource discovery is an important process for a large scale distributed grid computing to achieve an efficient resource provisioning and a fair resource usage. Most of grid computing middlewares [16] [17] rely on a central server for a resource discovery while suffering from the shortcomings of server-client based approach. Our decentralized parallel processing framework can provide a new approach for resource discovery while overcoming those shortcomings of centralized approach.

We can define Map task as a matchmaking process that checks whether a current resource status satisfies a job requirement or not. The matching result is returned to the local Reduce task. The Reduce task orders matchmaking results from the local Map task and child nodes' Reduce task. The ordering can be done using various methods (e.g., Ordering based on *Rank* value in Condor [16] or waiting queue size at each host). The ordered matchmaking result will be delivered to the Reduce task of parent node for further aggregation and summarization.

As we have shown in this section, users can define a Map function as processing locally available datasets. In Reduce function, many aggregation methods, such as sum, average, sorting, and even distributed data mining algorithms, can be defined in accordance with users' need.

V. Comparison with Hadoop/Google MapReduce

The goal of Hadoop [1] and Google MapReduce [2] is sharing computing power in order to process large dataset in a cluster environment. Our decentralized Map and Reduce parallel processing framework, on the other hand, targets not only sharing computing power but for monitoring and managing nodes in a highly distributed environment.

Hadoop and Google MapReduce run a central manager which is responsible for assigning Map and Reduce tasks

and dealing with a worker failure. On the contrary, our framework has no central manager node. Instead, a self-organizing bounded-broadcast tree is responsible for committing Map and Reduce tasks.

For efficient task processing, Hadoop MapReduce detects a lagging node based on a progress score. If a node's progress score is below a threshold value, which is decided based on the average Map and Reduce task execution time, the node is marked as a straggler. The node's job is reassigned by a central manager. The Late scheduler [18] detects lagging nodes that will finish the farthest into the future based on *Remaining Job Portion/Progress Rate*, which considers both how fast the node is processing the task and how much amounts of work remain. In Hadoop and Late algorithm a central manager is obligated to monitor Map and Reduce task processing nodes. In our system, XML-RPC timeout from an underlying P2P system will distinguish lagging nodes. If a parent node detects a RPC timeout from one of its child nodes, the parent node would prune the retarding node from a bounded-broadcast tree.

VI. Evaluation

In this section, we evaluate our Map and Reduce Parallel Processing Framework on a P2P system in a decentralized and heterogeneous environment using a simulation and real-world deployment on PlanetLab [19]. We implemented an event-driven simulator to evaluate our system in a controllable manner. The simulator models not only our parallel processing framework but also Brunet [5] routing, XML-RPC, and node management shown in Figure 2 by reusing the Brunet code base that has been extensively verified and deployed in realistic infrastructures such as PlanetLab. In order to allow experiments with large networks on a controlled environment, it uses simulated event-driven times. We used the King data set [20] to set network latency between nodes and Archer [21] in order to run simulations on a decentralized and distributed computing resources efficiently. After running 10 simulations with different parameters, we calculate an average value.

A. Simulation

Figure 5 shows latency to complete counting number of nodes and checking P2P network consistency which was introduced in Section IV-B using our parallel processing framework. In order to compare our system with a sequential system, we also performed the same task using sequential method. In a sequential method, a master node asks N number of nodes in a pool to get connection table one after another. Based on the returned connection tables, the master node calculates consistency value. Our parallel processing task is performed once a minute at a random node, and sequential task is performed once in an hour. In case of our Map and Reduce based parallel crawler, it took 8.7 seconds to scan 4000 nodes. Otherwise, sequential crawler took 27 minutes to scan 1000 nodes, and it could not complete the task within an hour when the number of node is 2000, 3000, and 4000 nodes. In order to show the latency increasing pattern, we added $O(LOG(N))$ pattern. Though the latency increasing pattern is slightly larger than $O(LOG(N))$, we can see that it almost follows the $O(LOG(N))$ pattern. On the other hand, sequential task shows linearly increasing latency pattern. Table I presents bandwidth consumption per one task. Both approaches show linearly increasing bandwidth consumption when the number of scanned nodes increases. The sequential task shows more bandwidth consumption than our approach, because it takes more than one routing hops from a master node to a target node while querying connection table. In our parallel approach, in contrast, each querying action takes one hop, because every node gets connection table from its nearest neighbors.

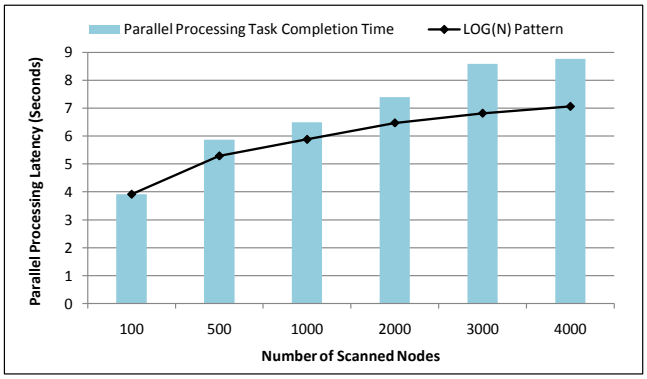

Figure 5: Latency to count number of nodes and check consistency: Blue bar shows our Map and Reduce parallel processing latency. $LOG(N)$ increasing pattern (solid line) is added for reference.

Table I: Bandwidth Usage Per One task

# Nodes	Crawling BW Consumption	
	Parallel Processing	Sequential Processing
100	35 KBytes	41 KBytes
500	218 KBytes	278 KBytes
1000	430 KBytes	623 KBytes
2000	946 KBytes	1,472 KBytes
3000	1,310 KBytes	2,237 KBytes
4000	1,861 KBytes	3,234 KBytes

By optimizing a sequential method, we can decrease the latency and bandwidth consumption of the sequential approach. However, we want to emphasize that our Map and Reduce based parallel processing framework can perform tasks in parallel without imposing overheads for managing parallel jobs. In addition, with the aid of bounded-broadcast, our approach can distribute tasks and aggregate results with the $O(LOG(N))$ latency cost.

B. Experiment on PlanetLab

In this section, we present an experiment result that was conducted on PlanetLab with about 430 nodes deployed globally. Figure 6 shows a clustered statistic of connection latency among nodes in a P2P network. The clustered

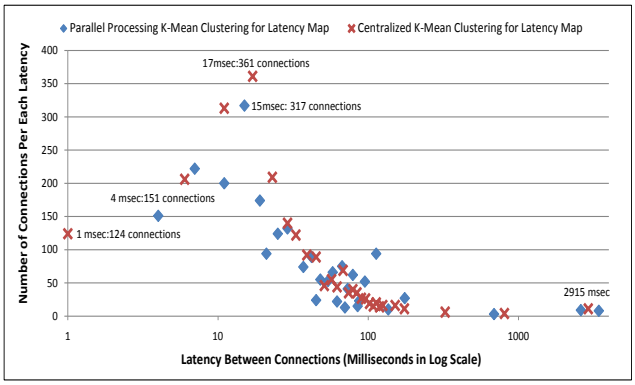

Figure 6: Clustered connection latency on PlanetLab. The blue diamond mark shows a clustered latency map which uses our decentralized parallel processing framework. The X mark shows centralized clustered information.

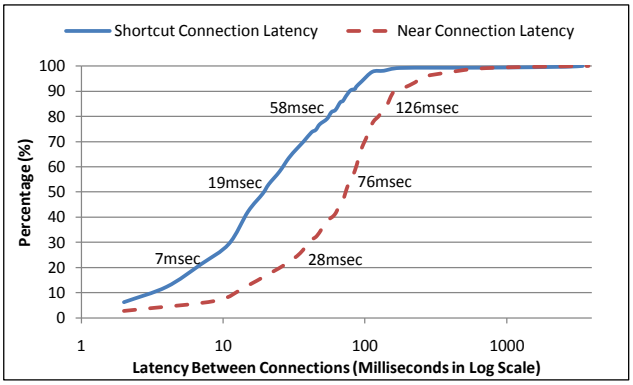

Figure 7: Cumulative distribution of clustered connection latency of shortcut and near connections.

latency information is created using our parallel processing framework with K-means clustering algorithm as explained in Section IV-C. The blue diamond mark shows a distributed and parallel K-means clustered connection latency created by our framework. X mark shows a centralized K-means clustered connection latency to compare the performance with our system. At the centralized method, every node reports its latency values to one central node, and the central node creates a clustered connection latency using a K-means clustering method. The horizontal axis shows the latency between connections in milliseconds. Note that the horizontal axis value is presented in log scale. The vertical axis shows the number of connections for each latency value. For example, a point whose horizontal axis value (i.e., latency) is 15 msec and the vertical axis value (i.e., the number of connections) is 317 means that there are 317 connections whose clustered latency is 15 msec. From the figure, we can see that the clustered connection latency distribution of our Map and Reduce parallel processing framework is similar to that of centralized K-means clustered distribution.

Figure 7 shows Cumulative Distribution Function (CDF) of clustered structured near and shortcut connection latency of Brunet deployed on PlanetLab. The clustering is performed using our parallel processing framework. As we can see from the figure, structured shortcut connections show shorter latencies than near connections. The reason is Brunet establishes shortcut connections using proximity metrics based on Vivaldi coordinates. Otherwise, structured near connections are formed to near left and right neighbors based on the P2P address. From this experiment, we can confirm that the proximity-based shortcut connection algorithm in Brunet is working correctly. As shown here, our parallel processing framework can provide a P2P system monitoring capability in an efficient and scalable manner.

For the distributed K-mean clustering experiment, 30 clusters are created using a midpoint initial centroid value selection method. We noticed that different number of clusters and initial centroid selection methods show different clustering performance. Choosing the optimal number of clusters and initial K-means centroid values on a hierarchically distributed tree structure remain as our future work.

VII. Related Work

Binzenhofer et. al [13] presents a Chord-based P2P system monitoring system. They use a snapshot algorithm to infer the status of P2P pool at a given time window. The algorithm determines a begin and end address of subregion for scanning. Nodes within the region pass a token to gather statistics for pool status. Due to its sequential token-passing, its communication latency increases as $O(N)$, where N is the number of scanned nodes. Different from our parallel processing system, this system supports only pre-defined pool monitoring functions.

Flock of Condor [22], WaveGrid [23], and PastryGrid [24] presents a grid computing middleware for resource sharing which runs on top of P2P network. PastryGrid [24] is built on Pastry [8], and a job scheduling is performed by sequentially traversing nodes in a pool until all queued jobs are assigned. In WaveGrid [23], a timezone-aware overlay network is formed to consider time zone information for a job scheduling. Flock of Condor [22] connects several Condor [16] pool header nodes using Pastry [8] P2P framework. Head nodes within the P2P pool share their hosts and queued job information for efficient resource scheduling. BonjourGrid [25] built a compute power sharing framework while using publish/subscribe-based multicast mechanism. The scheduling cost for sequential traversing (e.g., $O(N)$, where N is the number of nodes for job scheduling in a pool) is much more expensive than our parallel processing framework, which is $O(logN)$, where N is the number of nodes for a job assignment in a pool. Without a structured

multicast mechanism, flooding a scheduling message to the network does not guarantee complete message transmission. In addition, these approaches are inappropriate for tasks whose results correlate to each other.

VIII. Conclusion and Future Work

This paper presents a parallel processing framework which runs on a structured P2P network. The system relies on a self-organizing tree based broadcast method, bounded-broadcast, for parallel task distribution without the need of a central head node. With the aid of Map and Reduce primitives from functional programming languages, we could clearly define tasks suitable for parallel processing as shown in the use case scenarios. In our implementation using Brunet, users can easily define and insert a task by using underlying XML-RPC interface or use pre-defined tasks. Using a simulator which reuses Brunet routing and node management source code, we demonstrated that our system provides a scalable and efficient parallel processing framework. PlanetLab deployment and the clustered connection latency information using K-means clustering method show the feasibility of our system as a real-world application with various application scenarios. Future work includes optimizing user-transparent scheduling method and heuristics to handle faulty nodes in a bounded-broadcast tree.

References

[1] A. S. Foundation, http://hadoop.apache.org/.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] G. S. Manku, M. Bawa, P. Raghavan, and V. Inc, "Symphony: Distributed hashing in a small world," in *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003, pp. 127–140.

[4] J. M. Kleinberg, "Navigation in a small world," *Nature*, vol. 406, August 2000. [Online]. Available: http://dx.doi.org/10.1038/35022643

[5] P. Boykin and et al., "A symphony conducted by brunet," 2007.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 149–160.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 161–172.

[8] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," 2001.

[9] T. W. Choi and P. O. Boykin, "Deetoo: Scalable unstructured search built on a structured overlay," in *Seventh International Workshop on Hot Topics in Peer-to-Peer Systems*, 2010.

[10] V. Vishnevsky, A. Safonov, M. Yakimov, E. Shim, and A. D. Gelman, "Scalable blind search and broadcasting over distributed hash tables," *Comput. Commun.*, vol. 31, pp. 292–303, February 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1332143.1332433

[11] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, no. 2, 2003.

[12] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 379–390, 2004.

[13] A. Binzenhofer, G. Kunzmann, and R. Henjes, "A scalable algorithm to monitor chord-based p2p systems at runtime," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 8 pp.

[14] M. J. Zaki, "Parallel and distributed data mining: An introduction," in *Large-Scale Parallel Data Mining*, ser. LNCS/LNAI State-of-the-Art Survey. Springer-Verlag, Heidelberg, Germany, 2000, vol. 1759.

[15] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[16] J. Basney and M. Livny, *Deploying a High Throughput Computing Cluster*, R. Buyya, Ed. Prentice Hall PTR, 1999.

[17] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Fifth IEEE/ACM International Workshop on In GRID*, 2004.

[18] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-99, Aug 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-99.html

[19] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, 2003.

[20] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: estimating latency between arbitrary internet end hosts," *SIGCOMM Comput. Commun. Rev.*, 2002.

[21] R. J. Figueiredo and et al., "Archer: A community distributed computing infrastructure for computer architecture research and education," in *Collaborative Computing: Networking, Applications and Worksharing*, vol. 10, 2009, pp. 70–84.

[22] A. R. Butt, R. Zhang, and Y. C. Hu, "A self-organizing flock of condors," *J. Parallel Distrib. Comput.*, vol. 66, pp. 145–161, January 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2005.06.022

[23] D. Zhou and V. Lo, "Wavegrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 10, 2006.

[24] C. C. Abbes, H. and M. Jemni, "A decentralized and fault-tolerant desktop grid system for distributed applications," in *Concurrency and Computation: Practice and Experience, 22*, 2010, p. 261?277.

[25] H. Abbes, C. Cerin, and M. Jemni, "Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–8, 2009.