Efficient Execution of MapReduce Applications on Irregular NoC Topology

Abraham Addisie*, Meghan Cowan†, Milind Furia‡, Helen Hagos§
{abrahad*, cowanmeg†, furia‡, hahagos§}@umich.edu

***Abstract*—With the exponential growth of data, data-intensive applications are becoming common in CMPs. Such applications are mostly written using MapReduce programming model, because it provides simple programming interface. The underlying interconnect in CMPs is network-on-chip(NoC). Due to extreme device scaling, link failure in NoCs is becoming a common scenario. Link failures leads to an irregular network topology. MapReduce frameworks like Phoenix++ are written based on an assumption of a robust regular network topology. For communication bound applications, irregular network topology leads to inefficient execution of MapReduce applications. Both the mapping phase and reducing phase of MapReduce applications provide opportunity for load balancing based on the connectivity of a node to the rest of the system. In this project, we proposed and evaluated novel load balancing algorithm in both phases of MapReduce. We found that depending on the communication to computation ratio and the number of unique keys of workloads, the execution time and network latency can be significantly improved.**

I. Introduction

MapReduce is a programming model which provides programmers with a simple abstraction to implement a wide range of data-intensive applications. MapReduce programmers write an application using two simple functions. The functions are called map and reduce functions. The map function process the input data and emits intermediate key-value pairs. The reduce function aggregates each set of intermediate key-values pairs associated with the same key. The MapReduce framework then automatically handles the partitioning of the input data, the scheduling of the map and reduce tasks to the available processors in the system, and the exchange of intermediate key-value pairs among the reducers. The MapReduce model is currently one of the most popular programming paradigms for big-data applications, and it constitutes the backbone of many important problems within the areas of image processing, artificial intelligence, web search engines, genome sequencing, among many others. However, the exponential increase in the amount of data generated is compounding the challenges of big data applications, as we strive to find efficient ways to sort through this data, analyze it and meaningfully make use of it. Thus, developing frameworks that support the high-performance execution of MapReduce-based applications is a critical step towards overcoming these massive big-data challenges.

The original implementation of MapReduce, first introduced by Google [4], was developed to run on computer clusters. However, MapReduce is also increasingly being adopted on various other computing infrastructures, such as chip-multiprocessors (CMPs). The growing number of cores in CMP designs provides a highly-parallel computing platform that can be leveraged to run MapReduce applications in parallel and achieve fast execution speeds.

The continuous shrinking of transistors has made ensuring reliability a challenge. As transistors become more fragile, wear-out becomes common. Since MapReduce applications split workload among cores, communication between nodes plays a significant role. Permanent link failures in the NoC may be a hindrance to communication and hence cause the entire chip to malfunction. Applications running on irregular topology will be subject to increased communication latency due to congestion that can potentially slow down execution time.

Our paper identifies and analyzes sources of inefficiencies in both the map and reduce phases caused by execution on an irregular topology. Specifically, we categorize each phase of an application as being either communication or computation bound and note that a communication bound phases generates read requests at a higher rate creating network congestion. Our analysis show that the reduce phase is more sensitive to the network topology due to the all to all exchange of key value pairs that occurs during shuffling, while the mapping phase depends more on application characteristics as more processing tends to be done during that phase.

To combat these inefficiencies we developed a novel load balancing algorithm to distribute work among cores according to their connectivity in the network. Our algorithm takes advantage of the fact that nodes with greater hop counts have most if not all of their communication go through a single link in the network, and assigns those nodes less work to avoid congestion on the potentially bottlenecked link.

Our solution addresses the effects of hard faults with regards to performance of MapReduce applications. Our algorithm only applies to connected topologies. We do not consider any irregular topologies that cause the network to become disconnected.

Contributions. Specifically, we provide the following contributions:

1) We propose a novel load balancing algorithm for irregular network topologies.
2) By load balancing only the mapping phase, we achieved over 5% improvement in execution time and over 30% improvement in average latency for communication bound workloads.
3) By load balancing only the reducing phase, we achieved over 25% improvement in execution time for workloads

with large number of unique key-value pairs.

II. Related Work

The MapReduce programming model was originally introduced by Google [4] to provide the efficient execution of data-intensive applications on a cluster of commodity-machines. Hadoop [17] is another cluster-based open-source framework that implements the MapReduce framework. In addition, MapReduce have been ported to different platforms [6,9,12,16,21]

With the adoption of chip-multiprocessor (CMP) architectures, several MapReduce implementations have been proposed to target these systems [13,14,19,20]. In particular, Phoenix++ [19] is an optimized implementation of MapReduce for multi-core systems. It provides a simple programming interface for users, while internally managing the execution of the MapReduce tasks. With the adoption of network on chip interconnect, different works have been proposed to implement MapReduce [5,7].

Irregular NoC topologies are created due to link failure. [11] proposes an algorithm BLINC which allows to quickly perform reconfiguration locally in-case of failures affecting few routers. One approach for load balancing in irregular topology was used by [15] wherein, they transform the network into a B+tree like topology and balance the load on this new topology. Similar techniques have also been studied where an irregular network is mapped onto a regular topology since the underlying network topology affects the load balancing [18].

[10] demonstrates the dynamic load balancing capabilities of the Charm++ and [8] can be used to further improve performance for tightly-coupled applications running in Grid computing environments. However, they require knowledge of the topology before-hand. [2] uses positional scan load-balancing technique in a two-phase dynamic load-balancing technique for "Peer to Peer" computing systems but this is well suited for regular topologies more in general.

III. MapReduce Background

MapReduce provides a simple programming model for data-intensive applications. Users can develop parallel applications by simply writing a map and a reduce function. Whereas, the remaining aspects of parallel applications, including the data scheduling and data exchange stages, are handled by the particular implementation framework. In a typical framework, the application's input data is first partitioned and divided among the cores in the system. Then, each core runs the user-defined map function, which processes the input data and produces a list of intermediate key-value pairs. Once the mapping stage is complete, the intermediate data is sent over the interconnect to the reducer cores, such that all key-value pairs belonging to the same key are allocated to the same reducer. Common MapReduce implementations typically uses distributed hash table to store intermediate key-value pairs. In the final stage, each core executes the user-defined reduce function to aggregate key-value pairs received from all cores. WordCount is a canonical example that illustrates the different stages of MapReduce. The WordCount problem computes the frequency of occurrence of each word in a document, and it is utilized in many important applications, such as search engines and social networks (*e.g.* indexing, identifying trending topics, *etc.*). Figure 1 shows pseudo-code for the map and reduce functions for WordCount. The map function parses the input document based on the user-defined word delimiter. It emits each word as key and 1 as a value, generating intermediate key-value pairs. The reduce function, shown in the right part of Figure 1, collects all key-value pairs from the different cores. It then sums up the values for each word, generating a final list of all unique words and the number of times they occurred in the input document.

```
map(doc):
for each word w in doc
   emitKeyValue(w,1);
```

```
reduce(key,values):
result = 0;
for each v in values
   result += v;
emitFinal(key,result);
```

Fig. 1: **The map and reduce functions for WordCount**. The map function emits a key-value pair for each word in the document. The reduce function aggregates the word counts and emits the final result.

IV. Irregular NoC Topology

Continuing decrease in the feature size, technology scaling down to the nanometer domain, lower power voltages, higher operating frequencies of integrated circuits leads to increasing susceptibility to transient, intermittent and/or permanent faults. A fault is a defect in a system caused by any or all of the below methods:

- process defects, taking the form of missing contact windows, dust settlement, parasitic transistors, oxide breakdown, electrostatic discharge (CDM, HBM) events, etc.;
- aging effects, electro-migration, permanent antenna faults, NBTI (negative bias temperature instability), etc;
- silicon defects, bulk defects (cracks, crystal imperfections), surface impurities, etc; and
- intermittent effects caused by temperature variation, threshold voltage instability; etc.

To explain further, permanent faults are caused by irreversible changes in the chip. It can occur during the manufacturing processes. It can also occur during the usage of the circuit, when the circuit is old and starts to wear out. Once the permanent fault occurs, those will remain in the chip for its entire lifetime and the faulty results can be regenerated.

Intermittent faults occur occasionally in spurts. They repeat themselves now and then and are not as continuous as permanent errors. These faults can be caused by aging of the hardware, threshold voltage instability, temperature variation and such faults can lead to permanent faults too. These faults are very hard to detect and occur in the presence of some specific environmental condition or input.

The above two classes of faults are the very reason why we introduce our project, problem and the solution of it. These two types of faults can cause broken links in a design as shown in figure 2. Figure 2 shows the irregular NoC topology with the faulty mesh network that we are using in our experiments with

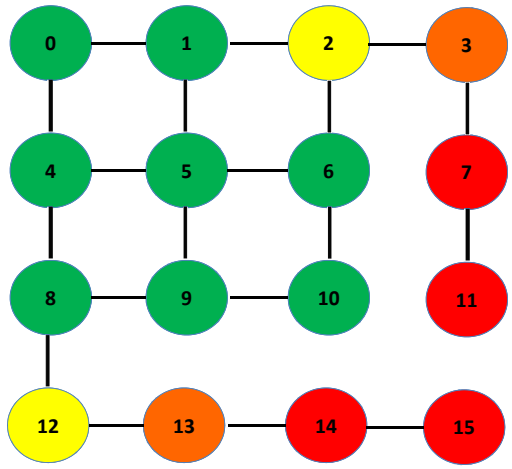

Fig. 2: **Irregular NoC Topology**. Permanent link failure leads to irregular topology. The figure shows an irregular topology where the link between 6 and 7, 10 and 11, 9 and 13, 10 and 14, 11 and 15 are broken. Because of this, node 11 and 15 become the least connected nodes, followed by node 7, 13, and 14.

5 broken links called the Mesh_5Edge network. Because of these faults, cores 15 and 11 have the least connection, which are marked as red. Similarly, cores 14, 7 and 13 have lost fewer links.

V. Load Balancing Algorithm

Our load-balancing algorithm works by assigning nodes with the highest average hop count, the least amount of work and vice-versa. The intuition behind our algorithm is that a broken link in the NoC creates a bottleneck by forcing nodes to communicate through less optimal paths with higher hop counts.Nodes with significantly higher average hop counts tend to have most if not all of their communication go through a single link in the network, creating a potential bottleneck in the network that lead to congestion.

For instance in Figure 2 the link between nodes 8 and 12, link (8, 12), is at high risk for congestion as all packets from nodes 12-15 must go through it. Compared to other nodes, nodes 12-15 experience a larger increase in average hop count because a greater percentage of their routing paths are reconfigured to less optimal paths. Therefore, using average hop count as a guide, our load-balancing algorithm indirectly reduces traffic on bottle necked links by assigning nodes most affected by a bottleneck link less work.

Below is the formula used to calculate the relative workload each node is assigned where hops(i, j) is the minimum number of hops to travel from node i to node j, and N is the number of nodes in the network. The numerator calculates the average number of hops between any two nodes in the network, or the network average hop count, while the denominator calculates the average number of hops from node X to any other node, or node X's average hop count. To get the exact percentage each node receives, the relative workloads need to be normalized.

$$NodeXload \propto \frac{\frac{\sum_{i=0}^{N-1}\sum_{j=i}^{N-1} dist(i,j)}{\binom{N}{2}}}{\frac{\sum_{i=0}^{N-1} dist(X,i)}{N-1}} \quad (1)$$

In Table 1 we show the average hop count and workload distribution our algorithm computes for the topology in Figure 2. It can be seen that nodes 12-15 which lie on the bottlneck link (8, 12) have high average hop counts and are assigned smaller tasks than well connected nodes such as nodes 1 and 5.

Node Id	Avg. Hop Count	% Workload
0	3.20	7.02
1	3.07	7.33
2	3.33	6.74
3	4.00	5.62
4	2.93	7.66
5	2.80	8.02
6	3.07	7.32
7	4.80	4.69
8	3.07	7.32
9	2.93	7.66
10	3.20	7.02
11	5.73	3.92
12	3.60	6.24
13	4.27	5.27
14	5.07	4.43
15	6.00	3.74

TABLE I: **Load Balancing A** Average hop count and assigned percentage of workload for each node in Figure 2. Workload size is calculated using the equation 1.

VI. Inefficiency in Mapping Phase

MapReduce's mapping phase begins with the scheduler dividing the input data into map tasks. Instead of statically assigning each worker an equal amount of data, the scheduler creates many map tasks per worker. Workers are assigned map tasks one at a time, and request new tasks from the scheduler as they finish until all tasks are completed. A worker processes a map task by reading a small chunk of data, computing intermediate key value pairs, and then locally storing the key value pair. Depending on the application, the amount of processing per byte of data can cause mapping to be either communication or computation bound. For example, during mapping linear regression executes 3 multiplies and 5 writes for every byte of input data, while histogram only executes 2 adds and 3 writes for every 3 bytes of data.

MapReduce's current division and scheduling of map tasks is optimized for computation heavy map phases. It attempts to equalize the time spent processing data among the workers by allowing fast workers to receive more tasks than slow workers. For computation bound map tasks the latency of reading from memory is negligible compared to the time spent computing and there is less concern for network congestion as memory reads are occurring at a lower frequency.

For communication bound mapping, the amount of processing per byte of input data is relatively low. This makes communication bound mapping more sensitive to the network topology for two reasons. First, the rate at which memory reads are generated is much higher. Therefore, relative to amount of time computing, memory latency is a significant factor in execution time, especially for more isolated nodes. Second, input data is processed at a much higher rate, creating the risk for congestion. In a chip multiprocessor, memory controllers are distributed across all nodes. If the data a worker needs to

read is not controlled by its local memory controller it makes a remote memory request. The latency of a remote memory requests depends on the network topology as requests must traverse the network to reach their destination. An irregular topology creates a greater risk for congestion as bottlenecked links create a single common path for many memory requests. Congestion slows down memory requests for all nodes in the network creating a global problem.

Our proposed topology based load balancing algorithm would assign each worker 1 map tasks whose size is determined by the worker's overall connectivity to the rest of the network. Workers executing on nodes most affected by bottlenecked links will be assigned a smaller chunk of data and make less memory requests. This reduces the reduces the risk of congestion in the network and allows for potential an decrease in execution time over the baseline division of labor.

VII. Inefficiency in Reducing Phase

Each mapper produces key-value pairs in its local hash table. A larger proportion of the hash table is resident in the local cache of the corresponding mapper. During the reduce phase of MapReduce, each reducer core pulls key-value pairs from the hash table distributed across cores. In the current implementation of Phoenix++, the hash table is equally partitioned per each reducer, leading to a balanced load in healthy network. However, in an irregular topology, the connectivity of each cores is different. Some of the reducer cores can be bottleneck. Different factors affect the amount of effect an irregular topology creates, in the performance of MapReduce applications. One of the factor is the number of unique key-value pairs that a workload has. A workload with large number of unique key-value pairs takes large fraction of time in the reduce phase. If this workload has high communication to computation ratio, then the irregular topology can cause a significant degradation in the performance.

Our load balancing algorithm, in the reducing phase, divides the hash table containing the intermediate key-value pairs according to the connectivity of cores. A core's connectivity is determined by the average hop count of itself and the rest of the cores in the system. Each core with higher connectivity, takes larger portion of the hash table. We generate a ratio value per core using an algorithm that factors the connectivity of cores. Each core uses its ratio to determine the range of the hash table to read during reducing phase. A less connected node reads small range of hash table entry. A highly connected node reads large range of the hash table. The overall effect is that, when the latency of the core reading from other cores is considered a less connected node is less likely to become a bottleneck. Ideally, the ratio would allow all the cores to finish at the same time. However, the connectivity based on the average hop count of a core based on others is linearly related with the number of cycles taken by the core to complete execution. There are other factors affecting the ratio. For a communication dominant workload, the ratio should be highly dispersed. For a computation dominant workload, the ratio should have less variance across cores. In the latter case, if the ratio is highly dispersed, as the execution time does not depend on the network latency, then the cores can have unproportional amount of work. Each core that are highly connected and as a result got larger range of the hash table, can take significantly larger amount of time than the low connected cores that are assigned with a lower range of hash table. Consequently, the execution time can get worse than the baseline.

VIII. Experimental Evaluation

We performed our experimental evaluation using Gem5 [3]/Garnet [1], cycle accurate, integrated core and network simulator. Gem5 models the individual core, whereas Garnet models the network level characteristics. We modeled a 16-node CMP, which allows us to evaluate our data-intensive applications in reasonable time. The results shown in the following section were based on experiment performed on the irregular topology shown in Figure 2. However, our proposed load balancing algorithm is not restricted to any kind of network topology.

We first ran several workloads from Phoenix++ without any modification on our simulation infrastructure. We collect core-level information such as number of cycles and network-level information such as average latency and link utilization. We performed two kinds of load balancing by implementing our algorithm on both the mapping and reducing phases. We modified the task splitting function for each workload in Phoenix++. In addition, we modified the reducing phase for each workload in Phoenix++. We measured the execution time and other network parameters and compared in isolation, these values with our baseline. We discussed our experimental results in the following section. Table II summarizes the Gem5 and Garnet configurations we used for our simulations.

Gem5 configuration	Garnet configuration
clock frequency = 1GHz	topology = 4*4 mesh
L1 D and I cache size = 16KB	number of input buffers = 2
L1 D and I cache lat. = 16KB	number of vcs = 2
L2 cache size = 128KB	routing algorithm = minimal
L2 cache latency = 12ns	
Protocol = MESI_Two_Level	

TABLE II: **Experimental setup.** We performed cycle-accurate core and network simulations on Gem5 and Garnet, simulations, respectively. We run the modified Phoenix++ workloads on these simulation platforms and got core and network level information.

The workloads we investigate, listed in Table III, are based on the Phoenix++ framework: *wc*, *hist*, and *lr* are the WordCount, Histogram, and Linear Regression workloads, respectively. For *hist*, we used a smaller version (5MB) of the data sets provided with Phoenix++. For *wc*, to evaluate the mapping phase, we used a smaller version of data sets (1MB) provided with Phoenix++. For evaluating the reducing phase, we used a data sets that contain unique words to stress test the all to all exchange of key-value pairs during this phase. For *lr*, we used randomly generated data sets. The size of the data sets was kept 5MB. The size of our data sets were smaller than those provided by Phoenix++, because simulation time was unreasonably high for those provided by Phoenix++.

A. Performance Evaluation - Mapping Phase

Figure 3 shows the execution time of different workloads for the baseline and our load balanced solution. hist achieved over

name	description	# of keys
hist	finds the histogram of RGB values in an image.	768
lr	performs linear regression.	5
wc	counts the frequency of words in a document.	varies

TABLE III: **Phoenix++ workloads.** We selected these workloads because they represent a wide range of data-intensive applications. hist is commonly used for image analysis; lr in artificial intelligence; and wc in search engine and document processing. The number of unique keys for *wc* depends on the number of unique words in the inputs.

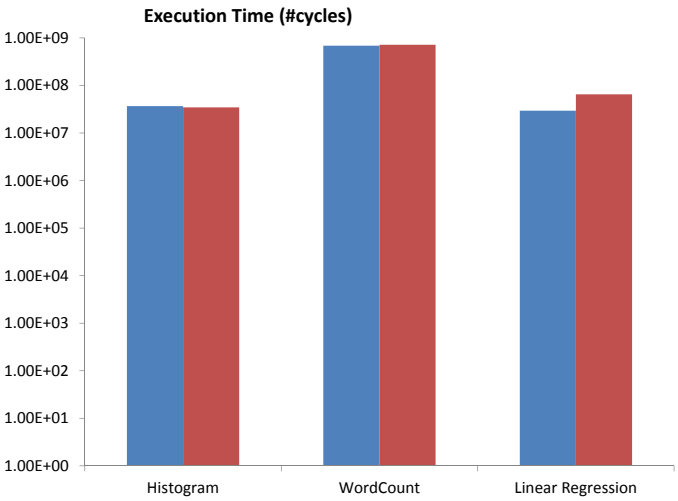

Fig. 3: **Mapping phase - execution time**. We achieved over 5% improvement in execution time for hist. lr shows higher execution time over the baseline, because it is computation bound

5% improvement in execution time, where as the execution time for wc remain the same. The execution time lr get higher in our load balanced solution. hist benefits in terms of execution time because it has the highest communication to computation ratio. The application, hist reads a pixel and generate intermediate key-value pairs without additional operations. wc has a smaller communication to computation ratio as compared with hist. wc performs a couple of comparison per each character read from memory. Because of this, our load balanced algorithm fails to achieve improvement in execution time for wc workload. lr has the smallest communication to computation ratio. In our lr data set, each data point is a single byte. Per each byte read, lr compute 3 multiplications and generate 5 key-value pairs. This makes lr computation bound. Since it is computation bound, our load balancing algorithm gave more work for highly connected cores and make these cores to finish much latter than the baseline. The connectivity of nodes doesn't matter for lr workloads. Users can turn off our load balancing solutions for compute bound applications.

Figure 4 shows the average latency of different workloads for the baseline and our load balanced solution. Average latency includes both network latency and queuing latency. hist achieved over 30% improvement in average latency and wc achieved over 7%. The average latency for lr increased by over 159%. As explained before, we found that lr is computation bound and does not benefit from load balancing.

Figure 5 shows the average link utilization of different workloads for the baseline and our load balanced solution. The

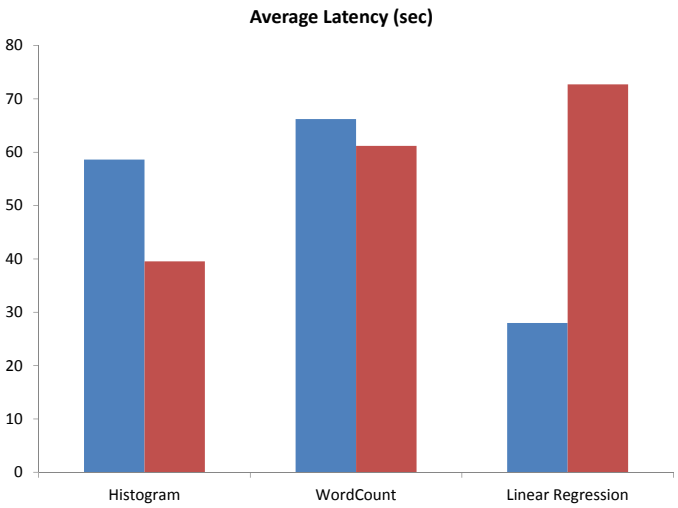

Fig. 4: **Mapping phase - average latency**. We achieved over 30% improvement in average latency for hist and over 7% for hist. lr shows higher average latency over the baseline, because it has a high communication to computation ratio.

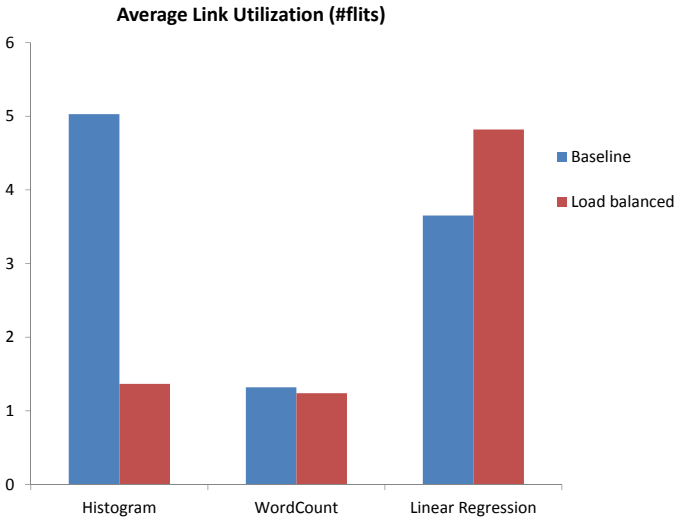

Fig. 5: **Mapping phase - link utilization**. We achieved over 70% improvement in average link utilization for hist and over 6% for hist. lr shows higher average link utilization over the baseline, because it is computation bound.

average link utilization measures the average number of flits that are present in a network link per cycle. hist achieved over 70% improvement in average link utilization and wc achieved over 6%. The average link utilization for lr increased by over 30%. As explained before, we found that lr is computation bound and doesn't benefit from load balancing.

B. Performance Evaluation - Reducing Phase

Figure 6 shows the execution time, the average latency and the average link utilization for wc workload. The values are shown for the baseline and our load balanced solution. Since wc has large number of unique keys, we were able to achieve over 25% improvement in execution time by only load balancing the reducing phase.

Figure 7 shows the execution time, the average latency and the average link utilization for hist workload. The values are

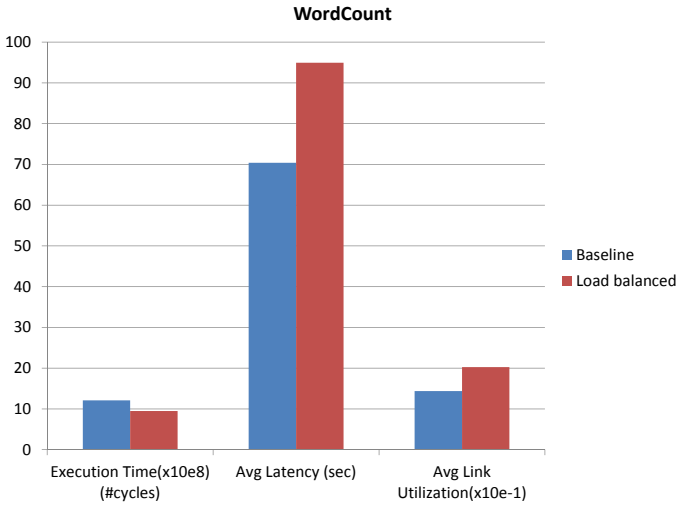

Fig. 6: **Reducing phase - wc evaluation**.

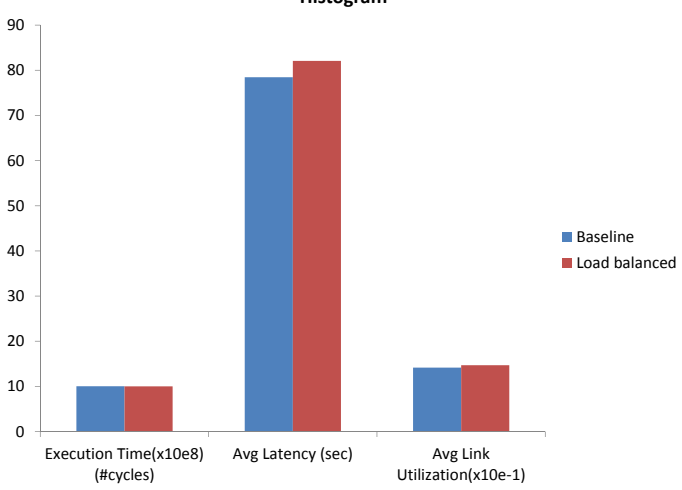

Fig. 7: **Reducing phase - hist evaluation**.

shown for the baseline and our load balanced solution. Since hist has small number of unique keys (768), we achieved 10% improvement in execution time, smaller than that of wc.

IX. Conclusions

We proposed an algorithm capable of load balancing data-intensive applications in a fault induced irregular NoC interconnect. We found experimentally that, for communication dominant applications, our load balancing algorithms improve execution time by more than 5% and average latency by more than 20%, during the mapping phase. During the reducing phase, we improve execution time by more than 25%, for applications with a high number of unique key-value pairs. We basis our load balancing algorithm on a heuristic function that utilized the average hop count to determine the connectivity of each core. Finding the exact mathematical model, which considers the characteristics of applications including computation to communication ratio is an important area of future work.

X. Acknowledgements

We would like to thank Prof. Valeria Bertacco and Mr. Dowoon Lee for their constant support and guidance throughout the course of the project.

References

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009.

[2] M. W. Akhtar, M. Kechadi, et al. On the efficiency of dynamic load balancing on p2p irregular network topologies. In *Parallel and Distributed Computing, 2006. ISPDC'06. The Fifth International Symposium on*, 2006.

[3] N. Binkert *et al*. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[5] K. Duraisamy *et al*. Energy efficient MapReduce with vfi-enabled multicore platforms. In *Proc. DAC*, 2015.

[6] W. Fang, B. He, Q. Luo, and N. Govindaraju. Mars: Accelerating MapReduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4), 2011.

[7] K. Gyftakis, I. Anagnostopoulos, D. Soudris, and D. Reisis. A mapreduce framework implementation for network-on-chip platforms. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, 2014.

[8] C. Huang, O. Lawlor, and L. V. Kale. Adaptive mpi. In *Languages and Compilers for Parallel Computing*. 2004.

[9] F. Ji and X. Ma. Using shared memory to accelerate MapReduce on graphics processing units. In *Proc. IPDPS*, 2011.

[10] G. Koenig, L. V. Kale, et al. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[11] D. Lee, R. Parikh, and V. Bertacco. Brisk and limited-impact noc routing reconfiguration. In *Proceedings of the Conference on Design, Automation & Test in Europe*, 2014.

[12] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. Goh. MrPhi: An optimized MapReduce framework on Intel Xeon Phi coprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 26(11), 2015.

[13] M. Lu *et al*. Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, 2013.

[14] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. CellMR: A framework for supporting MapReduce on asymmetric cell-based clusters. In *Proc. IPDPS*, 2009.

[15] I. K. Savvas, M. Kechadi, et al. Efficient load balancing on irregular network topologies using b+ tree structures. In *Parallel and Distributed Computing, 2007. ISPDC'07. Sixth International Symposium on*, 2007.

[16] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *Proc. of FPGA*, 2010.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of MSST*, 2010.

[18] F. Silla and J. Duato. Is it worth the flexibility provided by irregular topologies in networks of workstations? In *Network-Based Parallel Computing. Communication, Architecture, and Applications*. 1999.

[19] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 2011.

[20] A. Tripathy, A. Patra, S. Mohan, and R. Mahapatra. Distributed collaborative filtering on a single chip cloud computer*. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 140–145. IEEE, 2013.

[21] R. Zheng, K. Liu, H. Jin, Q. Zhang, and X. Feng. Accelerate MapReduce on GPUs with multi-level reduction. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, 2013.