2017

Efficient Parallel All-Pairs Computation Framework: using Computation - Communication Overlap

Venkata Kasi Viswanath Yeleswarapu
Iowa State University

Efficient parallel All-Pairs computation framework: Using computation - communication overlap

by

Venkata Kasi Viswanath Yeleswarapu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Secure and Reliable Computing)

Program of Study Committee:

Arun K. Somani, Major Professor

Yong Guan

Lu Ruan

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to dedicate this thesis to my parents Sri. Yeleswarapu Sri Rama Chandra Vara Prasad and Srimati. Yeleswarapu Radha. I would also like to thank my friends and family for their loving guidance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my immense gratitude to Dr. Arun K Somani, for his guidance, support and patience throughout my research. I thank him for instilling confidence in me, through his encouragement, insights and belief in me. I would like to thank Dr. Cory J. Kleinheksel of Iowa State University, who worked with me during the initial stages of my research and made valuable insights to my work. I would like to thank my sister Sailaja and my friend Lakshmi Sindhura for their virtual presence, love and care throughout the course of my master's degree. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Yong Guan and Dr. Lu Ruan.

ABSTRACT

The advent of parallel computing systems enabled the users with huge computation power to efficiently process huge work loads. Most of the recent applications, which are data intensive, require parallel computing power to complete the job efficiently. To facilitate efficient computing there is a necessity for simplified abstraction of the parallel computing systems.

We propose one such parallel computation abstraction, designed to solve All-Pairs problems which fits the needs of several data intensive applications. All-Pairs problems require each data element to be paired with every other data element. This framework aims to address recurring problems of scalability, distributing equal work load to all nodes and reducing memory footprint. Our framework reduces memory footprint of All-Pairs problems, by reducing memory requirement from $N/\sqrt{P}$ to $3N/P$. A bio-informatics application is implemented to demonstrate the scalability (ranging up to 512 cores), redundancy management and speed up performance of the framework(super-linear speed up).

CHAPTER 1. THE PROBLEM: MOTIVATION

All research and business fields are witnessing increase of data being generated in large volumes. Algorithms are challenged to keep up with the pace in data intensive applications. To cater the needs of these applications, high performance computing and cloud computing are utilized. These parallel computing environments increase productivity of applications.

Distributed computing expertise is confined to a relatively small section in the Computational Science and Engineering community. Researchers in fields like Biology, Mathematics and Bio-Informatics have the need for distributed computing, but might lack the expertise. Inadvertent poor choices can lead to misuse of shared resources. Developing a framework to solve a class of problems and deliver optimal performance is a solution to aid the usage of distributed computing. Such a framework acts as an abstraction to hide complications and ease the usage.

Our work proposes one such framework to solve All-Pairs problems, a class of problems in some research and business fields. All-Pairs problems can be illustrated with a popular "handshake" problem (1), in which N people attend a meeting and every person shakes hand with every other person in the meeting. This is a symmetric (commutative) interaction where $\frac{N(N-1)}{2}$ handshakes take place. All-Pairs problems require every data element in a dataset to interact with every other data element.

Consider the following dataset with N (say) data elements indexed from 0 to N-1,

$$E_N = \{e_0, e_1, ...e_{N-1}\}$$

An interaction between two elements can be computation between two elements, which can be symmetric or asymmetric.

$$e_i \leftrightarrow e_j, i < j$$

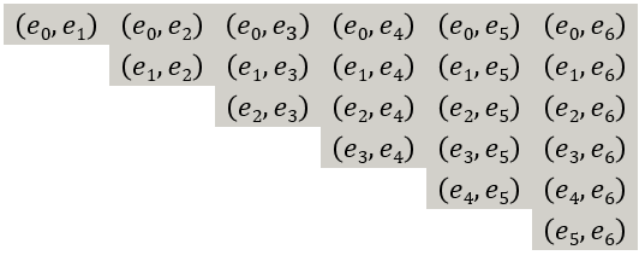

Figure 1.1 All-Pairs interactions for a set of elements

These elements can be primitive data elements, data structures like graphs, tries, files etc. and complex structures with many fields like JSON objects, tweets, images, etc. Multiple All-Pairs applications involve using these complex data structures.

More formally, All-Pairs problems is defined over two data sets where every data element has to interact with every other data element in the data sets. A generalized All-Pairs problem statement can be formally stated as follows:

$$\mathbf{M = F\ (A \odot B)}$$

where A, B are data sets

F is the function used for computation

M is the output matrix Here M[i, j] is obtained from interaction between A[i] and B[j]

For instance, consider a data set of seven elements as given below:

$$E_N = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$$

All-Pairs algorithm which pairs every data element in the set with every other data element gives the interactions as shown in Figure 1.1. It must be noted that interactions between data elements $\{e_0, e_1\} : (e_0, e_1)$ and (e_1, e_0) is considered to be the same due to commutative nature in interaction. Thus it can be noted that, for a data set of N elements, $\binom{N}{2}$ interactions take place.

The computational complexity of the All-Pairs problems is defined by the number of interactions between elements. For 'N' number of elements, number of symmetric interactions is given by $\frac{N(N-1)}{2}$ and number of asymmetric interactions are given by $N\,(N-1)$. Thus the computational complexity is given by $O\left(N^2\right)$.

For large values of N, data management and memory management becomes complex. All-Pairs problems inherently requires access to entire dataset, such that every element to be interacted with every other element. Challenges arise when the data sets or the intermediate data produced in the applications, exceeds the available memory size of the system. Many data intensive All-Pairs applications in bio-informatics (2) and metagenomics (12) with large data set (say N data elements), require multiple copies of intermediate data of N^2 or sometimes N^3 dimensions. For example, consider data intensive applications in biometrics establishing correlation among faces in a large data set size, obviously maintaining all data elements in memory exceeds available memory size of the system. A typical All-Pairs application in biometrics, Face Recognition Grand Challenge (4) (5) consists of comparing a set of 4010 images each of 1.25 MB size. These applications maintain a similarity matrix, where each element in the matrix represents comparison of two images which forms a matrix with 16,080,100 elements. Future biometrics applications needs All-Pairs computation over 60,000 iris images, which is 200 times bigger than the currently existing problem.

1.1 Approach

Minimizing required number of data elements in main memory of a node is managed in distributed computing by using data replication. Selected data elements are replicated on different nodes and moved around to complete interactions between all elements on a distributed manner. Nodes can communicate with each other to fetch required data segments or store results to complete the computations. Outputs from different nodes can be consolidated or forwarded to other applications in multiple ways. Minimizing replicated data to avoid redundancy in memory and time consumption has been a recurring theme in this research domain. Our research aims to address the time and memory challenges involved in solving the problem using efficient memory management techniques.

We reduce the memory footprint by distributing data and overlapping communication time among nodes with computation time. Our approach also employs round robin neighbor communication approach between nodes. Each node communicates with its neighbor in an asynchronous fashion to facilitate the computation - communication overlap. Data replication and

distribution is managed in such a way that computation interactions are evenly distributed between all nodes. This framework achieves balanced load distribution, elimination of duplicate computation and un-supervised computation-communication overlap.

We evaluate our approach using a real world, data intensive bio-informatics application, PCIT (2), for finding associations between genes in co-expression networks using correlation and information theory approaches. PCIT application is a quintessential data intensive All-Pairs application where, each gene has to interact with every other gene to form correlation matrices to determine associations between genes. Multiple implementations of PCIT application were developed to resolve limitations like scalability, high time and memory usage. Previous PCIT implementations were not scalable due to high memory footprint. Also, for large datasets PCIT application consumed days together to complete the computation. We prove the effectiveness of our work by scaling the PCIT application for large data sets with smaller memory footprint and time consumption. Proposed framework completes the computation for more than 33,331 genes with optimized time and memory consumption. We evaluate the framework by analyzing speedup performance and memory usage. We compare the results with best known results to evaluate the effectiveness of this framework.

1.2 Contributions

- We develop a parallel computing abstraction and framework to compute All-Pairs applications. This framework aids non-computational distributed computing researchers to utilize this computing power in their research fields.
- We address the recurring problem of high memory footprint in solving All-Pairs problems. Our work reduces memory requirement reduced from $N/\sqrt{P}$ to $3N/P$ for more than 9 processes ($P \geq 9$).
- We demonstrate high scalability with super linear speed up. We experimented scalability till 512 cores (32 nodes).
- We demonstrate the use of communication and computation overlap in achieving the speed up.

- We propose a simpler and easier data distribution principle.
- We distributed balanced load over all compute nodes and eliminated redundancy completely to achieve efficient All-Pairs computations with minimized memory footprint and time consumption (For more than 9 processes, $P \geq 9$).

CHAPTER 2. COMPUTATIONAL FRAMEWORKS

Research in developing higher level abstractions for parallel programming is a recurring theme and can be classified into three major types.

- Parallel programming languages abstractions
- Frameworks abstractions
- Abstractions for a specific class of applications

2.1 Parallel Programming Languages Abstractions

One of the ways to deliver higher level abstraction for users is creating a parallel programming language. A parallel programming language is generally developed as an extension to an already established sequential language with parallelizing abilities. Most of the parallel programming languages developed were extensions of sequential languages like FORTRAN, C, C++ and Java. For example, Charm++ (6) is a parallel programming language that is an extension of C++ and Python.

2.2 Frameworks Abstractions

Frameworks are programming structures where the user can enter the sequential code and the structure develops the parallelized machine code or intermediate code. Frameworks facilitate a huge ease of use and commercial value. An example framework is FraSPA (7) developed in a user guided manner. This framework synthesizes parallel applications from existing sequential applications and middleware components for multiple platforms and diverse domains. It is based on design patterns and generative programming techniques.

2.3 Abstractions for a Specific Class of Applications

These abstractions are developed for a class of applications, where a user enters sequential code for a computation and uses a framework to parallelize the job on multiple nodes. For example, All-Pairs and wavefront problems (5) are examples of applications which can use higher level abstractions for parallel programming. An implementation can be done by Dynamic Linking or Static Linking and by developing required libraries. These abstractions do not parallelize the sequential code given by the user, but use the parallel code for data distribution, synchronization, communication and result collection. The abstraction uses a give global view of data, communication transparency and control over the data. These abstractions offer high performance as they are tailored for a set of applications and offer the following advantages:

- Hides pitfalls of distributed computing
- Improves code maintenance and reuse
- Reduces the number of lines of code to be written manually
- Hides the notion of parallel programming from the users purview
- Increases productivity
- Offers time efficiency
- Is easier to use

The work presented in this paper, belongs to the third category. We develop a framework to present higher level abstraction for All-Pairs problems which fits the needs of several data intensive applications.

CHAPTER 3. PREVIOUS RESEARCH

3.1 Significance and Applications

3.1.1 Significance

All-Pairs problems fit the needs of many research and engineering fields. Also, many problems in science and engineering can be reduced to All-Pairs problems. Early filtering and data clustering techniques (8) can be applied to reduce jobs to smaller jobs and apply them to All-Pairs problems. All-Pairs problems can be used in research fields for two purposes.

- To understand the behavior of an algorithm based on datasets.
- To find the co-variance of two datasets based on the algorithm.

3.1.2 Applications

Scalability issues for All-Pairs problems have increased with rapid increase in data generation and data set size. For example, applications in bio-informatics and health related systems (9) needed to reference a set of genes to every other gene set. Several advances in these fields have led to massive increase in data generation. These applications are quintessential examples for the Big data All-Pairs problems, which are aimed by this work.

All-Pairs problems occurs naturally in many research areas. In Physics, n-body motion problem (10), (11) predicts the individual motion of celestial objects, where each object gravitationally interacts with every other object. In metagenomics, complex graphs formed from protein clustering are used to identify protein functions. These graphs are formed by determining the likeness of a protein to every other protein.

In data mining fields all-Pairs problem is used to understand the behavior of algorithms (5). Similarly, researchers need to test multiple algorithms on a given data set to identify which algorithm delivers better performance. Frameworks to solve these applications deliver quick and reliable results with an efficient time and memory consumption.

3.2 Previous Work

Approaches have been developed on distributed clusters, multicore CPU's, FPGA, Intel' multi core MIC using openMP and MPI to perform such computations.

3.2.1 High Memory Footprint

A framework (5) was designed to solve All-Pairs algorithms showing improved performance of applications in biometric and data mining fields. This work differentiates and shows the performance improvement of Active storage over demand paging. Active storage is the process of storing all data elements in the memory whereas demand paging is the process of querying for data over network from FAT node. Data elements are distributed to all nodes using spanning tree algorithm and performance in solving All-Pairs problems is compared between active storage and demand paging schemes. Although active storage delivers high performance with smaller turn around times, it requires all memory elements to be stored in memory and demands high memory footprint. Such applications with large data sets, high memory footprint can eclipse local resources, stressing the necessity to relax the requirement of having all elements in memory.

3.2.2 Memory footprint management

Memory footprint at a node is the amount of memory required to perform the computation at the node. Memory footprint management has been addressed in many contexts.

N-body problems in molecular dynamics have a atomic decomposition problem which requires every atom communicating with every other element. Authors in (13) proposed a method to balance the load and perform communication between all atoms to perform force decompo-

sition. For N elements and P processors, this method distributes two arrays of $N/\sqrt{P}$ elements to each processor.

Driscoll et al. (14) proposed an approach to solve All-Pairs problems with variable data replication and relaxing the requirement of having all the elements in memory. They show that data replication in the system can be a variable (c) and distributing data set into P/c subsets (for P processors), one for each processor. Thus forming P/c smaller subsets to reduce the memory footprint. Also, the authors show that for achieving lower bound on communication between processors to solve the All-Pairs problems, the replication factor should be $c = \sqrt{P}$. Thus, every processor receives a dataset of $N/\sqrt{P}$ elements. An additional set of $N/\sqrt{P}$ elements is communicated between different processors to solve the All-Pairs problem. Thus every processor holds two sets of $N/\sqrt{P}$ elements. The additional data set is shifted and copied between processors to generate all the required pairings between all data elements. The algorithm works best for $P = c^2$ processors.

Authors in (10) proposed an approach to solve All-Pairs problems with reduced memory footprint upto 50%. They distribute data elements in a mathematical form called cyclic quorums. Authors prove that cyclic quorums have All-Pairs property and cyclic quorums can be used to solve All-Pairs problems. The basic idea is as follows. Let us assume the number of nodes in the system is N. Let N nodes be denoted by set P=$\{P_0, P_1, P_2, P_3,..., P_{N-1}\}$. Let the data be divided into N subsets of data set (called quorums), denoted by S=$\{S_0, S_1, S_2, S_3,..., S_{N-1}\}$ and each subset in S, S_i follows certain properties of quorums and contains data segments in a cyclic manner. The subset is said to be a quorum if it follows the following properties:

- Element e_i is contained in the subset S_i, for all i ϵ 1,2,3.. N
- Non-empty intersection property. $S_i \cap S_j \neq \Phi$ for all i, j ϵ 1,2,3.. N
- Equal work property: $|S_i|$=k, for all i ϵ 1,2,3.. N, k<N
- Equal responsibly property: Element e_i is contained in k S_j 's for all i ϵ 1,2,3.. N

The above properties can be applied for N data segments to generate quorums and solve All-Pairs problems. For example, let us consider a dataset for 7 elements given by:

$$E_N = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$$

Let us assume that we solve this problem with seven nodes. Then seven quorums namely, $\{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}$ are generated as shown in Table 3.2.2. All-Pairs interactions generated from these subsets are shown in the table and it can be noticed that all data elements are paired with every other data element. Every quorum set is given to its respective processor and no further communication is necessary to complete the All-Pairs interactions. Table 3.2.2 can be referred for understanding the interactions that take place at every node.

Table 3.1 Computations performed at every node using quorums approach

Node	Quorums	Data element interactions
Node 1	e_0, e_1, e_3	$\{e_0, e_1\}, \{e_0, e_3\}, \{e_1, e_3\}$
Node 2	e_1, e_2, e_4	$\{e_1, e_2\}, \{e_2, e_4\}, \{e_1, e_4\}$
Node 3	e_2, e_3, e_5	$\{e_2, e_3\}, \{e_3, e_5\}, \{e_2, e_5\}$
Node 4	e_3, e_4, e_6	$\{e_3, e_4\}, \{e_4, e_6\}, \{e_3, e_6\}$
Node 5	e_4, e_5, e_0	$\{e_4, e_5\}, \{e_5, e_0\}, \{e_4, e_0\}$
Node 6	e_5, e_6, e_1	$\{e_5, e_6\}, \{e_6, e_1\}, \{e_5, e_1\}$
Node 7	e_6, e_0, e_2	$\{e_6, e_0\}, \{e_0, e_1\}, \{e_6, e_1\}$

The above work aims at solving All-Pairs algorithms for large scale data sets with reduced memory footprint. However, this method requires a brute force search to find the quorum sets for each node. Once quorums are identified the rest of the problem is straight forward. Initial data distribution has to follow the quorums which also induces computation overhead. This approach is taken as benchmark for betterment here in solving All-Pairs problems.

3.3 Challenges

All-Pairs problems might appear simple on the outset but might pose multiple challenges during implementation.

3.3.1 Number of Compute Nodes

It is a misconception to assume that assigning more number of processors to do the same work will give better results. Authors in (5) considered a small experiment, where running an

All-Pairs problems with 250 compute nodes gives worse performance than serial implementation of the problem. This shows that choosing optimal number of compute nodes is necessary to get optimal performance. In any parallel or distributed computing systems, additional nodes reflects additional overhead in exchange of information and data. In All-Pairs problems in particular, data needs to be transferred among nodes to complete all pairings with reduced memory footprint. Thus as number of nodes increase, the communication load also increases resulting in worse performance.

3.3.2 Data Distribution

Datasets have to be distributed to each node after choosing number of nodes. As mentioned previously, active storage delivers high throughput but results in high memory consumption. Demand paging reduces memory consumption but results in worse throughput. To achieve higher throughput with reduced memory footprint, subsets of data should be distributed to all nodes in an efficient manner.

3.3.3 Resource Limitations

Several unexpected limitations can occur while using distributed computing systems. Moreover, most of the clusters used for research purposes are shared among multiple users. Issues with processing, communication, storage and memory are often observed. For example, while writing outputs to files, the number of files open may eclipse the maximum limit. Also, issues while handling output and faults while recording outputs are quite often. A framework to handle all these tasks would reduce errors and makes problem solving easier.

CHAPTER 4. APPROACH

Our approach aims at reducing memory foot print by using communication between nodes. Reducing the number of elements required to store at every node, reduces data replication and reduces memory footprint. We propose to use a simple round robin neighbor approach to communicate between nodes to transfer required data to solve the problem while minimizing the need for elements. By introducing communication - computation overlap, we aim to solve the All-Pairs problems with $\sqrt{P}$ (where P is number of processors) less data elements than the quorums approach discussed in earlier chapter. The reduction in data replication and necessity can be explained by comparing our work with previous works. Table 4.1 provides a comparison and depicts the improvement in data replication of our work over previous works.

Table 4.1 Comparing previous works

Approach	**Data elements requirement**	**Communi-cation**	**Comments**
Moretti's Framework	N	No	High Memory footprint
Driscoll's Work	$2N/\sqrt{P}$	Yes	Replication of High memory footprint
Cyclic Quorums	$N/\sqrt{P}$	No	Exhaustive Complex initial data distribution
Computation Communication Overlap	$3N/P$	Yes	Ease of use, Low memory footprint ($P \geq 9$)

To explain the difference made by reducing the data elements consider Figure 4.1. The graph represents the number of data elements required for each approach as the number of processes increases. It can be seen that our approach needs considerably smaller number of

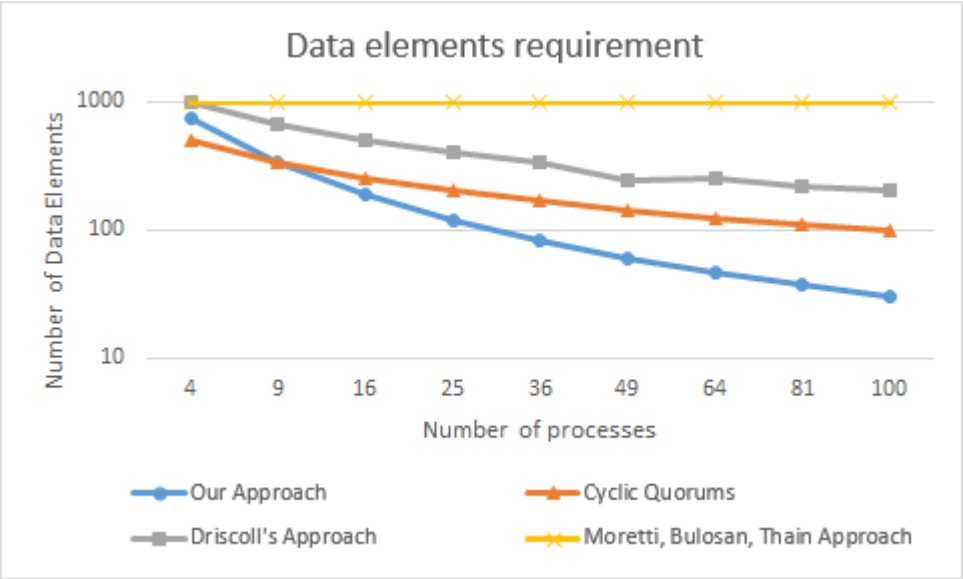

Figure 4.1 Comparing data elements requirement for previous works

data elements as number of processors increases. This difference grows larger with increase in both dataset size and number of processes.

4.1 How it Works

In our approach, each node accommodates three segments of the partitioned data. Every node communicates with every other node in a round robin fashion to complete the computation with a maximum of 3 data segments in the memory at any time instant. The communication time is overlapped with computation time to minimize the delay caused due to transfer of data segments between the nodes. Asynchronous communication routines are used to achieve computation-communication overlap.

Every node needs to accommodate a maximum of three segments. The computation is carried out in the following steps.

- Initially each node receives one of N/P data segments.
- In the first cycle, every node performs the computation on data segment available to it also transferring the data segment available to it to the next node.

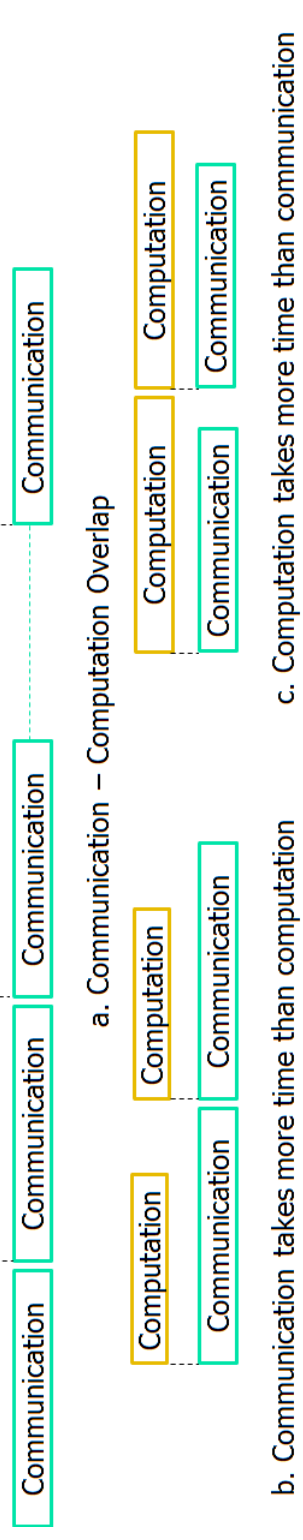

Figure 4.2 Figure a. shows the phenomenon of computation - communication overlap. Figure b. shows the scenario when communication takes more time than computation time. Figure c. shows the scenario when communication takes less time than computation time.

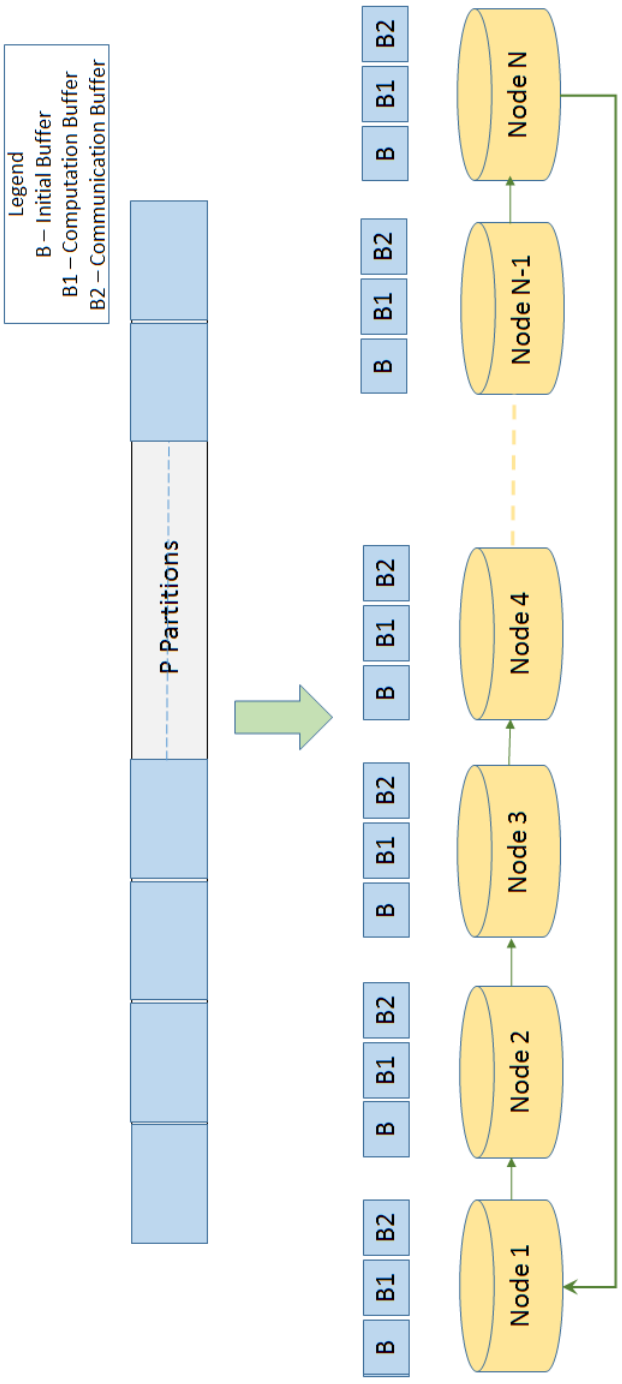

Figure 4.3 Round-robin Neighbor approach where every node gets 3 data segments

- During the next cycle, every node performs computation between data segment initially allocated to the node and data segment received during the previous cycle. At the same time, the newly received data segment is communicated to the next node in round robin neighbor fashion.

- For every following cycle, every node performs computation on the segment it receives in the previous cycle and transfers the data segment on which the computation is performed to the next node.

- Thus by the end of every cycle, every node will have a new data segment on which the computation has to be performed. This way we overlap computation and communication and minimize the time elapse in completing the computation.

- After every cycle, the data segment which is most recently used for computation in previous cycle is rewritten with another segment during communication. This process is depicted in algorithmic form for better understanding in Algorithm 1.

The concept of nodes communicating in a round robin- neighbor fashion can be better explained with an example. Let us consider a system which has four nodes. Let us assume that the data segment with 8 data elements and stated as follows: D=$\{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$
Let the nodes be denoted as P=$\{P_1, P_2, P_3, P_4\}$
Each node gets data partitions equal to

$$\frac{Total\ Number\ of\ data\ partitions}{Number\ of\ nodes}$$

Here, every node gets 2 data partitions. The nodes P_1, P_2, P_3, P_4 get the data segments D_1, D_2, D_3, D_4 respectively. Each data segment is allocated with the following partitions. D_1=$\{e_0, e_1\}$, D_2=$\{e_2, e_3\}$, D_3=$\{e_4, e_5\}$, D_4=$\{e_6, e_7\}$. Consider Table 4.2 how the data segments are communicated between nodes in round robin neighbor approach. Lets assume every node has three buffers. One for initial data one for communication buffer and one for computation buffer. Every node stores its initial data in initial buffer. The data segment that is communicated between nodes is stored in the communication buffer. Since, the communication and computation occurs simultaneously, we need an additional buffer to store the communicated

buffer and start communication in the next cycle. Computation buffer is used for swapping data from communication buffer and performing the computation. The communication in next cycle rewrites the communication buffer with new data segment.

Table 4.2 Data segments occupying the memory of nodes in each time cycle

Cycle	Node1	Node2	Node3	Node4
0	D_1	D_2	D_3	D_4
1	D_1, D_4	D_2, D_1	D_3, D_2	D_4, D_3
2	D_1, D_4, D_3	D_2, D_1, D_4	D_3, D_2, D_1	D_4, D_3, D_2
3	D_1, D_2, D_3	D_2, D_3, D_4	D_3, D_4, D_1	D_4, D_1, D_2

Table 4.1 explains the interactions that take place between different data segments at each node. The computation gets completed by the end of four cycles, when the round robin communication cycle gets completed. When we assume commutative nature between interactions in All-Pairs problems, we can observe that few redundant interactions are performed. These redundant computations can be avoided to achieve more performance. We have tested for redundant interactions for different set of nodes and made few observations. We have observed that few cycles can cause redundant interactions and the cycles differ with number of nodes.

- For odd number of processes, redundant interactions happen after $N/2 + 1$ cycles.
- For even number of processes, redundant interactions happen after $N/2$ cycles. In $(N/2+1)^{th}$ cycle, half of the interactions are redundant.

To encounter these redundant interactions and improve performance, for odd number of processes we perform just $N/2 + 1$ cycles to complete the All-Pairs problem. For even number of processes, we perform first $N/2$ cycles and in the $(N/2+1)^{th}$ cycle, only the first half of processes stay active and perform computations while the other half stay idle. This approach removes redundant interactions to improve performance in solving All-Pairs problems. Table 4.1 explains how data segments are communicated between nodes and communication-computation overlap is performed. Assume n referred in the table to be an odd number for better understanding.

```
Data: Two Input Data Sets
Result: All-Pairs problem solved in round-robin neigbor approach
Initialization of MPI processes;
P is number of processes;
my_id is the id of each process;
while data set remaining do
    determine the size of the data blocks each process should receive;
    communicate the size of the data blocks each process is receiving;
    distribute initial data in equal amounts to all processes;
end
Perform computation on the individual elements (if necessary);
while computation to be performed is not completed do
    Perform computation on data blocks available on the node;
    Perform Aynchronous communication;
    source = ((my_id - 1) + P) % P ;
    destination = ((my_id + 1) + P) % P ;
    if my_id == source then
        Asynchronously send data block to destination node
    end
    if my_id == destination then
        Asynchronously receive data block from neighboring source
    end
    Loop : if Computation in progress is completed then
        if receiving data block from neighbor node is completed then
            Perform Computation on the two blocks, go back to the beginning of loop;
        else
            Wait until the block is received
        end
    else
        continue the computation, go back to Loop
    end
end
```

Algorithm 1: Solving All-Pairs problems with Round Robin Neighbor Approach

Table 4.3 Computation at each node in each time cycle

Cycle	Node1	Node2	Node3	Node4
0	D_1	D_2	D_3	D_4
1	D_1, D_4	D_2, D_1	D_3, D_2	D_4, D_3
2	D_1, D_3	D_2, D_4	$\boldsymbol{D_3, D_1}$	$\boldsymbol{D_4, D_2}$
3	$\boldsymbol{D_1, D_2}$	$\boldsymbol{D_2, D_3}$	$\boldsymbol{D_3, D_4}$	$\boldsymbol{D_4, D_1}$

4.2 Computation - Communication Overlap

Figure 4.2 a. represents communication and computation overlap phenomenon. In Figure 4.2 b. we show the scenario when communication takes more time than computation. In this scenario, computation in the next cycle has to wait until the communication is completed. This causes delay in starting computation and worsens performance. In Figure 4.2 c. we show the scenario when computation takes more time than communication. In this scenario, computation in next cycle is started as soon as the previous computation is completed since the processor does not have to wait for any communication to get its data for performing the computation.

4.3 Modeling the System

It may be difficult to predict the performance of a distributed computing system due to a variety of reasons like programming style, frequency of input/output and dependencies on network. Using a framework which performs similarly for all workloads, one can predict the performance of the system. Factors like input data size, network bandwidth and job size determine the run time of the job. Understanding the working of a framework will certainly help in modeling the jobs and drawing best outputs in utilizing the framework. Output of the framework depends on choosing job size, number of processors to complete the job and bandwidth of the network involved. Since the framework overlaps computation time with communication time, time taken to complete each round of computation is maximum of computation and communication times. This is given as follows:

Table 4.4 Communications and Computations performed at each node at each time stamp.

Cycle	Node 0	Node 1	Node 2	——	Node n-1
0	Initial data distribution D_0	Initial data distribution D_1	Initial data distribution D_2	——	Initial data distribution D_{n-1}
1	Computation on D_0, Send D_0 to Node 1	Computation on D_1, Send D_1 to Node 2	Computation on D_2, Send D_2 to Node 3	——	Computation on D_{n-1}, Send D_{n-1} to Node 0
2	Computation on D_0-D_{n-1}, Send D_{n-1} to Node 1	Computation on D_1-D_0, Send D_0 to Node 2	Computation on D_2-D_1, Send D_1 to Node 3	——	Computation on D_{n-1}-D_{n-2}, Send D_{n-2} to Node 0
3	Computation on D_0-D_{n-2}, Send D_{n-2} to Node 1	Computation on D_1-D_{n-1}, Send D_{n-1} to Node 2	Computation on D_2-D_0, Send D_0 to Node 3	——	Computation on D_{n-1}-D_{n-3}, Send D_{n-3} to Node 0
-	———-	———-	———-	——	———-
$(n)/2$	Computation on D_0-$D_{(n-1)/2}$	Computation on D_1-$D_{(n-3)/2}$	Computation on D_2-$D_{(n-5)/2}$	——	Computation on D_{n-1}-D_0

$$T_{EachRound} = Max[T_{Communication}, T_{Computation}]$$

Here, $T_{EachRound}$ is the time taken to complete one round of computation and communication. $T_{Communication}$ is time taken to complete communication of data between two consecutive nodes. $T_{Computation}$ is the time taken to complete the computation on one node. Communication time is determined by the amount of data sent between two nodes. Thus, time taken for communication is data size over bandwidth of the network. Communication time is given as follows:

$$T_{Communication} = \frac{(n+m)s}{B}$$

where n and m are number of elements in each data set. s is the size of each element in the data set. B is the Bandwidth of network being utilized.

Computation time is an estimate of the time taken to perform All-Pairs operation over two datasets. Computation time is given by number of computations performed per processor multiplied by time taken for each computation. For two data sets with m and n elements respectively, the number of interactions is given by $m * n$. Time taken for $m * n$ computations is given by:

$$T_{Computation} = \frac{mn}{P} * (t + D)$$

Here, t is the time taken to complete a function call. D is any delay associated in dispatching job or writing output to files. Since these $m * n$ computations are equally distributed over P processors, computation time is calculated for the same $(m * n)/P$ interactions. If it takes i iterations to complete the job, then total time taken is i times time taken for each round. This is given as follows:

$$T_{Total} = i * T_{EachRound}$$

We construct an example set to determine how the given equations can be used for modeling the system to maximize the performance from the framework. We choose a value for P, the

number of processors based on the other factors. The factors to be taken into consideration are data sets size, time taken to complete a computation and bandwidth of the network. In Table 4.5, we set some sample values to justify how number of processes should be chosen based on the factors mentioned above. Three scenarios for different bandwidths, size of data and time taken to complete an interaction are considered to determine number of processes.

Table 4.5 Modeling System

n, m	t (ms)	s	B	**p**
4,000	1	1	100	200
4,000	1	1	**10**	20
1,000	1	1	100	50
1,000	1	**10**	100	5
100	1	1	100	5
100	**100**	1	100	50

4.4 Implementation

We implement our framework in such a way that the details of the parallel programming are hidden from the user. The user is expected to include a header file which contains parallel programming code and user has to write a piece of sequential code for the interaction between two data elements in a prescribed function. Data distribution, communication between nodes, load distribution and redundancy check are done by the code embedded in the header. When the code is executed, the framework code in the header file utilizes the function written by the end user to perform interactions between elements.

CHAPTER 5. EVALUATION

In this chapter, we evaluate the results of applying our approach over a real world data intensive application. We implemented the PCIT application (2) in such a way that it can be set as a job to our framework. Data distribution and computation is performed as defined in Chapter 3.

5.1 Test Setup

We conducted our experiments on a High Performance Computing (HPC) system called CyEnce at Iowa State University. This cluster, supported by NSF MRI program is accessible to researchers, faculty, principal and co-investigators on a shared basis. Every node has dual Intel Xeon E5 8-core processors and 128GB of memory. Our experiments are run on 1 to 32 nodes (16 to 512 cores) and each node has 40 Gb IB interconnect. Although, each node has a maximum memory of 128 GB, we limit our maximum memory consumption to 60 GB. This limit is placed to make our environment similar to most of the rented cluster environments like Amazon Web Services.

Three real and six simulated data sets were used for testing our approach. The simulated data sets are produced for certain number of genes and conditions based on the information given in (2). Real datasets include readings taken from cattle, mice and rice samples. Details of the datasets used are given in Table 5.1. Real datasets are distinguished with an asterisk (*) mark.

Number of genes are the primary factor to determine the computation complexity in All-Pairs problems, so our simulated datasets are generated with an increasing number of genes. The input columns correspond to the number of test subject conditions (2), (10) i.e number of

samples used. The simulated datasets are generated with rows and columns similar to real data sets.

We use MPI threads to perform parallelism in our code. The number of HPC nodes used for each observation are varied to make comparisons with previous works and identify the variance in behavior of the framework. Although general choice is 4, 8, 16 32, etc., we chose 4, 7, 8, 13, 16, etc. since cyclic quorums approach delivers different performance for singer difference sets (10) and we choose this approach as benchmark for comparing our performance.

5.2 PCIT Application

PCIT application is a bio-informatics application used for identifying meaningful gene-gene associations in a co-expression network (2). Authors in (2) combined partial correlation coefficients with information theory approach to find gene associations. The PCIT algorithm is used for reconstructing co-expression networks and correlation matrices to identify novel biological regulators. This correlation matrices are built by comparing every gene with every other gene forming an $O(N^2)$ computation. Then, the partial correlation coefficients in the matrix are subjected to guilt-by-association heuristic to analyze if a gene is correlated to any other gene using purely data. Except for the cyclic quorums approach this application is solved by placing all data elements in memory.

PCIT application similar to several data intensive applications suffers resource limitations. For instance, while dealing with a 16,000 gene data set on a single process, a correlation matrix with 256,000,000 entries is generated which occupies approximately 1.9 GB memory. When all the data elements are placed in the memory and considering the intermediate data generated, for large data sets the memory consumption can easily eclipse local resources. Thus PCIT application is a quintessential data intensive All-Pairs application which we are aiming to solve.

Table 5.1 Input Datasets Utilized in PCIT Experiments

Type	Rows	Columns
***Cattle**	27364	5
Simulated	33331	5
Simulated	39298	5
***Mice**	45265	5
Simulated	51232	1893
***Rice**	57194	1893
Simulated	63166	1893
Simulated	69133	1893
Simulated	75000	1893

5.3 Results

We conducted the experiments to observe the execution time and memory foot print of the application. We divide our observations for smaller data sets and larger data sets. We aim to prove that our approach delivers good performance for any size of data set.

5.3.1 Smaller datasets

We consider the datasets which can be run on a single node as smaller data sets. We have made few observations on how our framework performs on smaller datasets like 1000, 4000, 8000 genes for smaller number of processes upto 8. The graphs for those observations are plotted as shown in Figure 5.1. In Figure 5.1, we show the speed up of parallel implementation when compared to regular serial implementation of PCIT algorithm. In

As explained in Chapter 4, our approach's performance depends on number of computing nodes. If the computation time of a process is lesser than time taken to communicate data between nodes, then framework delivers worse performance. In figure 5.1, consider the fall of the curve representing 1000 genes data set. When the number of processes executing 1000 genes is increased from 3 to 4, the computation time is lesser than communication time leading to worse performance. For other curves in the graphs, speed up curve increases since computation time is greater than communication time.

We test our approach with PCIT algorithm implementation mentioned in (2) to find our approach performance on smaller data sets. As explained previously, our approach does not

perform better than implementation explained in (2), for all number of processes. But for a considerable computation load on processes, our approach gives better performance.

For example, PCIT implementation for 6400 genes in (2) takes 17.3 seconds where as our approach takes 16.2 seconds to complete for 4 processes.

5.3.2 Larger datasets

We perform experiments on our framework with large datasets which could not be solved previously (2). We aim to complete computations on these large datasets with smaller execution time and memory foot print.

As shown in Table 5.1, we have used nine different data sets for testing our approach for PCIT application. It should be observed that in Table 5.1, 5.2 we mean each node as 16 individual processes as each node contains 16 different cores. In Table 5.2, we compared the memory footprint casted by our approach with memory footprint of quorums approach. It can be seen that memory usage by our approach is considerably smaller than that of quorums approach. The reason for this is the reduction in number of data elements allocated for each process. For instance, Table 5.2 shows the memory footprint for a data set of 75,000 genes with 1,893 conditions in every gene over multiple nodes. While working on a single node with quorums approach, every process has to account for 18,750 genes(N/P), which will lead to maintaining a correlation matrix of size 2.61 GB approximately. On the other hand while working with our approach on a single node, each process has to account for 4688 genes which is $\sqrt{P}$ times smaller. This shows the effect on lower memory footprint.

As mentioned in previous chapters, all data intensive applications are run on shared clusters where resources are shared between multiple users. Most of the researchers run their applications on these shared clusters or rent cloud computing resources. Let us consider amazon web services (15), (16) which set an upper limit of 60 GB memory usage per node. We consider the same upper limit for memory consumption for smooth performance. In Table 5.2, we can see that for 75000 genes and 1893 conditions, quorums approach crosses 60 GB upper limit for less than 13 nodes. We highlighted the observations in bold which crossed the 60 GB upper limit.

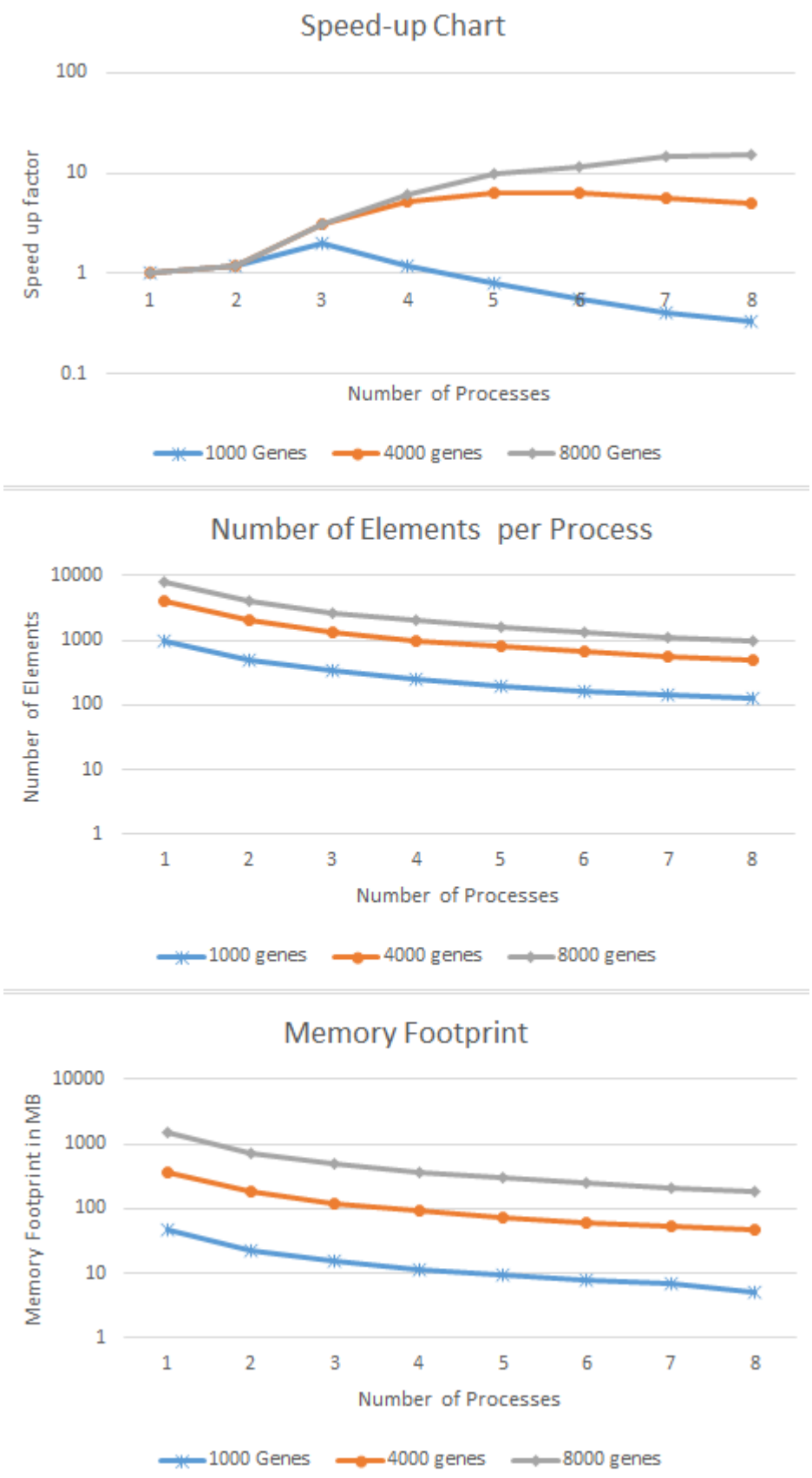

Figure 5.1 Variation of Speedup factor, Memory footprint and Number of Data Elements for implementation of PCIT application on our approach

Table 5.2 Memory Used Per Node (GB)

#Nodes	Our work	Quorums
1	42.263	**189.669**
4	34.97	**142.506**
7	17.636	**81.88**
8	16.752	**95.537**
13	7.438	59.091
16	5.0231	59.996
31	2.811	37.563
32	2.177	42.316

We performed the experiments on several datasets including simulated and real datasets as mentioned in Table 5.1 on varied HPC computing nodes. The execution time for different data sets are shown in Table 5.3. Our run time executions are faster than circular quorums approach mentioned in (10) for more than one node (16 cores). We could perform PCIT application for large data sets with smaller number of nodes (like 75,000 genes with 7 nodes) which was not possible in (10). We observe that for first four data sets, the execution time has worsened when the computing nodes are increased to 31 and 32. This is because the phenomenon explained in Chapter 4, where the computation time is lesser than communication time.

In Table 5.3, it can be observed that execution times with single computing nodes are large and more than execution times shown in cyclic quorums approach (10). This is due to the fact that, our framework treats all the processes as independent processes on multiple nodes. Processes that reside on same node and different cores, can directly read data from the shared memory rather than explicitly communicating data from one core to another core. When the framework is run on a single node with 16 cores, we provide data communication among processes rather than accessing required data from shared memory. This communication results in overhead in our execution times. In cyclic quorums approach, processes on the same node access local memory for required data. Since memory access is faster than communication routines, our approach is slower than cyclic quorums approach for single node executions. Our results are dependent on MPI communication routines. MPI communications have cache cancellation nature and sometime provide excessive overhead (17).

Table 5.3 Average Execution Runtimes (Seconds)

#Nodes	*27364	33331	39298	*45265	51232	*57194	63166	69133	75000
1	312.2 ± 1.2	322.9 ± 1.5	526.7 ± 1.2	2508.8 ± 2.5	-	-	-	-	-
4	34.3 ± 0.1	42.4 ± 0.1	83.1 ± 0.5	168.8 ± 1.7	-	-	-	-	-
7	17.7 ± 0.2	22.1 ± 0.1	46.7 ± 0.0	63.6 ± 0.0	271.8 ± 1.2	495.6± 2.5	437.8 ± 2.5	572.6 ± 1.5	716.7 ± 1.7
8	15.3 ± 0.1	17.6 ± 0.1	40.4 ± 0.1	46.5 ± 0.2	260.2 ± 0.1	388.9 ± 0.5	361.4 ± 1.1	434.3 ± 2.1	540.8 ± 1.2
13	12.4 ± 0.1	11.03 ± 0.1	17.0 ± 0.1	27.5 ± 0.0	124.1 ± 0.1	312.7 ± 0.8	302.1 ± 0.1	426.1 ± 1.2	489.1 ± 1.5
16	12.6 ± 0.1	12.3 ± 0.1	15.5 ± 0.1	21.5 ± 0.1	112.0 ± 2.3	284.8 ± 0.1	246.0 ± 0.1	354.1 ± 0.1	382.2 ± 0.1
31	18.4 ± 0.1	17.6 ± 0.1	17.3 ± 0.1	23 ± 0.1	72.7 ± 0.1	182.5 ± 1.5	153.4 ± 0.4	177.6 ± 0.5	216.8 ± 2.7
32	21.4 ± 0.1	19.3 ± 0.2	18.5 ± 0.1	23.4 ± 0.1	65.45 ± 1.2	142.6 ± 1.2	131.0 ± 1.7	152.4 ± 2.4	170.1 ± 1.6

CHAPTER 6. CONCLUSION

In this work, we proposed an approach to overlap computation and communication time to scale algorithms to perform All-Pairs problems on large data sets. We decreased the data elements requirement from $N/\sqrt{P}$ in cyclic quorums approach to $3N/P$ and significantly lesser than approaches which require all data elements in memory. Our approach significantly reduces memory foot print compared to all the All-Pairs implementations.We provided mathematical forms to model the system for a given job time and dataset size. We implemented a bio-informatics application on our approach to demonstrate our scalability, reduced time and memory consumption with real and simulated datasets.

Bibliography

[1] R. Hedegaard, Handshake problem, http://mathworld.wolfram.com/ HandshakeProblem.html, Jan. 2016.

[2] Watson-Haigh, Nathan S., Haja N. Kadarmideen, and Antonio Reverter. "PCIT: an R package for weighted gene co-expression networks based on partial correlation and information theory approaches." Bioinformatics 26.3 (2010): 411-413.

[12] T. Chapman and A. Kalyanaraman, An openmp algorithm and implementation for clustering biological graphs, in Proceedings of the first workshop on Irregular applications: architectures and algorithm. ACM, 2011, pp. 310.

[4] P. Phillips and et al. Overview of the face recognition grand challenge. In IEEE Computer Vision and Pattern Recognition, 2005.

[5] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, All-pairs: An abstraction for data-intensive computing on campus grids, Parallel and Distributed Systems, IEEE Transactions on, vol. 21, no. 1, pp. 3346, 2010.

[6] Kal, Laxmikant V. and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. OOPSLA (1993).

[7] Ritu Arora, Purushotham Bangalore. A Framework for Raising the Level of Abstraction of Explicit Parallelization, ICSE09, May 16 -24, 2009, Vancouver, Canada. IEEE 2009

[8] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In World Wide Web Conference, May 2007.

[9] H. Chae, I. Jung, H. Lee, S. Marru, S.-W. Lee, and S. Kim, Bio and health informatics meets cloud: Biovlab as an example, Health Information Science and Systems, vol. 1, no. 1, p. 6, 2013.

[10] Kleinheksel C.J., Somani A.K. (2016) Scaling Distributed All-Pairs Algorithms. In: Kim K., Joukov N. (eds) Information Science and Applications (ICISA) 2016. Lecture Notes in Electrical Engineering, vol 376. Springer, Singapore

[11] Wikipedia article on N-body motion problems, Wikipedia URL : *https* : *//en.wikipedia.org/wiki/N − bodyproblem*

[12] T. Chapman and A. Kalyanaraman, An openmp algorithm and implementation for clustering biological graphs, in Proceedings of the first workshop on Irregular applications: architectures and algorithm. ACM, 2011, pp. 310.

[13] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of computational physics, vol. 117, no. 1, pp. 119, 1995.

[14] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick, A communication-optimal n-body algorithm for direct interactions, in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013, pp. 10751084.

[15] Amazon Web Services, Aws high performance computing, https://aws.amazon.com/ hpc/, Mar. 2017.

[16] Amazon HPC Resources, Amazon ec2 instance types, https://aws.amazon.com/ec2/ instance-types/, Mar. 2017.

[17] Doerfler, Douglas, and Ron Brightwell. "Measuring MPI send and receive overhead and application availability in high performance network interfaces." European Parallel Virtual Machine/Message Passing Interface Users Group Meeting. Springer Berlin Heidelberg, 2006.