Efficient Distributed All-Pairs Algorithms: Management Using Optimal Cyclic Quorums

Cory James Kleinheksel, *Member, IEEE* and Arun K. Somani, *Fellow, IEEE*

Abstract—All Pairs problems occur in many research fields. The all-pairs problem requires all data elements to be paired with all other data elements. With the advent of new data intensive big data applications and increase in data size, methods to reduce memory foot print and distribute to work equally across compute nodes are needed. In this paper, we propose cyclic quorum sets for all-pairs algorithm computations to reduce memory foot print. We show that the cyclic quorum sets have a unique all-pairs property that allows for minimal data replication. The cyclic quorums set based computing requires only $N/\sqrt{P}$ size memory, up to 50 percent smaller than the dual $N/\sqrt{P}$ array implementations proposed earlier, and significantly smaller than solutions requiring all data in each node. Computation can be distributed efficiently and more importantly and are communication-less after initial data distribution, which is a huge advantage in minimizing computation time. Scaling from 16 to 512 cores (1 to 32 compute nodes), our application experiments on a real dataset demonstrated scalability with greater than 150x (super-linear) speedup with less than 1/4th the memory usage per node in our experiments.

Index Terms—Distributed all-pair computations, cyclic quorum set and all pair property, communication-less computation, memory foot print efficiency

1 Introduction: The Problem

Virtually all research and business fields have big data problems. Many are seeing orders of magnitude increases in data generated. Algorithms have the difficult challenge of keeping up with this pace. Infrastructure like internet backbones and data centers continue to evolve, but meet challenges on all sides; resources, bandwidth, and fault tolerance are all major constraints. This has led to phrases like, "swimming in sensors, drowning in data" [1]. As a solution, some have turned to the cloud computing or specialized high performance computing to meet their computing needs.

Our work addresses some of the challenges facing a specific type of big data interaction. The interaction considered requires all elements (nodes or data) to interact with all other elements in the set. This interaction can generally be referred to as an "all-pairs" interaction.

To illustrate this we use a popular "handshake" problem [2] as an example. The problem goes like this: P people attend a party and a popular greeting is to shake hands, how many handshakes take place? The answer of $\binom{P}{2} = \frac{P(P-1)}{2}$ can be derived. The simple hand shake example could be considered a symmetric interaction

$$e_i \leftrightarrow e_j, i < j. \tag{1}$$

More formally there is set E_N, where there are N elements indexed from 0 to $(N-1)$

$$E_N = \{e_0, e_1, \ldots, e_{N-1}\}. \tag{2}$$

The elements can be simple like a single item data structure or they can be complex structures with many fields. Fields are not restricted to a single data type either, as many bigdata problems can rely on heterogeneous datasets.

The all-pairs interaction considers all possible pairs of elements, $\binom{N}{2}$ symmetrically listed below

$$\{(e_0, e_1), (e_0, e_2), \ldots, (e_1, e_{N-1}), \ldots, (e_{N-2}, e_{N-1})\}. \tag{3}$$

The computational complexity of this general algorithmic form is not daunting as $\binom{N}{2} = \frac{N(N-1)}{2} = O(N^2)$ computation tasks. In fact, even for pair computations that do not have the commutative property, the complexity is unchanged. In general, polynomial $O(N^2)$ computations are considered computationally scalable.

When performing an all-pairs data interaction on the big data scale sizes, while the computational complexity theoretically is manageable, the data management becomes complex. The problem definition inherently requires access to the entire dataset, such that every data element can be paired and processed with every other data element in the set. When the datasets exceed a system's available memory size, this presents a challenge. Minimizing the amount of data replication in a distributed system is a recurring theme in this classification of algorithms. Our research addresses scalable distributed memory efficient computation for the general all-pairs interactions. We also avoid redundant computation that may arise during the distribution of the workload.

Distributed computing is used to share the computational workload. Since our problem definition requires

• *The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011.*
E-mail: Cory.J.Kleinheksel@gmail.com, arun@iastate.edu.

Manuscript received 4 Mar. 2016; revised 3 May 2017; accepted 19 May 2017. Date of publication 23 May 2017; date of current version 12 Jan. 2018.
(Corresponding author: Cory James Kleinheksel.)
Recommended for acceptance by D. Padua.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2707417

every data element to interact with the entire dataset, one way to accomplish this is to distribute all datasets to all computing nodes and assign a load-balanced workload to each node. This requires high memory footprint at all nodes, a significant limitation. Alternatively, a dataset can be divided and distributed to computation nodes. Then the computation and communication among nodes is organized to accomplish the desired computation. Earlier work, see Fig. 2, uses this strategy to distribute $\frac{N}{\sqrt{P}}$ data elements to each node. Then each node receives additional $\frac{N}{\sqrt{P}}$ data elements using an appropriate shift and communication strategy (see Section 2.3). With $\frac{2N}{\sqrt{P}}$ data elements present at each computation node, and additional communication step(s), all pair interaction computations can be performed.

Our technique also manages data replication and distribution in such a way that all interactions are spread across compute nodes. Similarities stop there though. Our algorithms have all nodes share the work load in a balanced manner, avoid duplicate computation, and require no communication step after initial data distribution. Our computation management also is used to reduce memory resource requirements per node and enable big data scalability.

Quorum systems are commonly used for coordination and mutual exclusion in distributed systems [3], [4]. We utilize quorum theory to manage data distribution in a systematic manner to achieve all-to-all interactions. We associate all computing associated with each quorum to one process, assign that process it to one computing node, and provide data elements associated with that quorum to that node. We utilize the slow quorum growth rate compared to number of nodes to scale all-pairs algorithms. The size of data elements associated with a quorum is $\frac{N}{\sqrt{P}}$ elements per node. There is no further communication step needed during the computation. This being much improvement over those that require all N elements per node and up to 50 percent improvement over those that have used replication techniques to reduce memory requirements to two arrays of size $\frac{N}{\sqrt{P}}$ elements per node.

Moreover, we are the first to show that cyclic quorum sets have an "all-pairs" property [5]. For $P = 4, \ldots, 111$, processes our work uses the optimal cyclic quorums from [4]. The cyclic quorum sets can be generated by finding the the base quorum, which unfortunately has to be found by an exhaustive search. These cyclic quorums are optimal in memory and computation for all Singer difference sets [6] and near-optimal for all others.

Furthermore, for the near-optimal difference sets, we developed a decentralized, load balanced, and communication-less management technique to identify and avoid all redundant computations in non-Singer difference sets. This adds greater flexibility for users to utilize all of their local or cloud HPC resources.

1.1 Contributions

We specifically make the following contribution.

- We provide the proof that cyclic quorums sets have an "all-pairs" property.
- We verified optimal correctness of previously published cyclic quorums sets for $N = 1, \ldots, 111$.
- We develop new application of cyclical quorums to facilitate scaling all-pairs data interactions in distributed computing.
- We used cyclic quorums provable lower bound in size to limit all-pairs data replication, which lead to reduction in memory footprint for each computing node by a factor of two with respect to the best known algorithm. Moreover, our algorithm requires no intermediate communication.
- We show that cyclic quorums load balance all-pairs computations and scale well.
- We develop a distributed, communication-less management technique to remove inherent small redundancy in computation to achieve efficient all-pairs computations on any $P \geq 4$ processes.

The rest of the paper is organized as follows. Section 2 discusses several application domains that encounter the all-pairs problem. We formalize the problem and discuss a few relevant approaches developed in literature. Section 3 defines quorum sets, and more specifically cyclical quorum sets. Section 4 provides a definition of the all-pairs property and a proof that cyclic quorums satisfy the property and help manage load balancing. We experiment with this approach and Section 5 present the results of our experiment with a bioinformatics all-pairs application to show the scalability and memory efficiency of our all-pairs quorum set solutions. We also note that except for some values of P, other quorum sets have some inherent redundancy (some interactions are computed more than once). Therefore, we develop a distributed, computation management technique to avoid such redundant computation without communication in Section 6. Section 7 provides concluding remarks.

2 All-Pairs Problem

The all-pairs problem (or "handshake" problem) occurs in many different fields and occurs in a broad classification of algorithms. For example, the bioinformatics field notably has seen an increase in data. Led by several advances in science including Next Generation Sequencing (NGS) technology, the use of data to drive biological and medical discoveries has become a prominent research method. However, the size of data and computation time can easily eclipse local resources. Hence, scaling algorithms to larger datasets and for utilizing more resources has been a reoccurring theme in bioinformatics. The authors in [7] surveyed publicly available bio and health related systems. Referencing a gene data against every other gene data is commonly performed in such applications.

In databases, this all-pairs or "handshake" problem manifests as a self-join without a join condition, forcing all tuples to interact with all other tuples.

In physics, the n-body problem predicts the position and motion of n bodies by calculating the total force every body has on every other body.

In biometrics applications, a similarity matrix can be formed using a set of images compared with itself using facial recognition [8].

In metagenomics, finding a protein's likeness to every other protein is a crucial part of forming the complex graphs used in protein clustering, which has led to new discoveries of protein functions [9].

In file data-duplication detection problem, all files needs to be compared all other files.

$$\begin{array}{rrrrrr}
(d_0,d_1) & (d_0,d_2) & (d_0,d_3) & (d_0,d_4) & (d_0,d_5) & (d_0,d_6) \\
 & (d_1,d_2) & (d_1,d_3) & (d_1,d_4) & (d_1,d_5) & (d_1,d_6) \\
 & & (d_2,d_3) & (d_2,d_4) & (d_2,d_5) & (d_2,d_6) \\
 & & & (d_3,d_4) & (d_3,d_5) & (d_3,d_6) \\
 & & & & (d_4,d_5) & (d_4,d_6) \\
 & & & & & (d_5,d_6)
\end{array}$$

Fig. 1. All-pairs of seven data elements.

2.1 General All-Pairs Problem Definition

On the surface the problem is very straight forward as shown in Fig. 1. Given a set of data elements (seven in our example), an algorithm pairs all elements with all other elements. Notice that it is not necessary to explicitly form a (d_1, d_0) pair because the pair can be formed by the (d_0, d_1) pair already present.

The pseudocode for a general all-pairs algorithm would look like the following:

```
**Given**: Array $D$
**for** $i \leftarrow 0$ **to** $length(D) - 2$ **do**
  **for** $j \leftarrow i + 1$ **to** $length(D) - 1$ **do**
    Perform work on pair $(i, j)$
  **end for**
**end for**
```

Stated more formally:

$$\text{Set of } N \text{ elements } D = \{d_0, d_1, \ldots, d_{N-1}\} \tag{4}$$

$$Pair(d_i, d_j), \text{ where } 0 \leq i < N - 1 \text{ and } i < j < N. \tag{5}$$

Eq. (4) enumerates the N data elements being paired, while Eq. (5) performs all data pairings resulting in $\binom{N}{2} = \frac{N(N-1)}{2}$ element pairings.

2.2 Solution Approaches

Multiple approaches were proposed in accelerating the execution of many of the important all-pairs applications using multicore CPUs, FPGAs, GPUs, Intel's many-core MIC, and distributed clusters.

2.2.1 All Elements in Memory

In [10] the authors provide a generalized framework to solve these all-pair classification of algorithms and show performance improvements for biometrics and data mining applications in a distributed system. This generalized framework showed that efficiently distributing all of the input data to all of the nodes prior to execution resulted in faster turnaround times than reading from the disk on demand. Every element interacting with every other element leads to a natural result of having all elements present in memory, resulting in high memory footprint. While dealing with data intensive applications, this high memory footprint can eclipse the local resources, which motivates the necessity in relaxing the requirement of having all elements in the memory.

2.2.2 Relaxing the All Elements Present Requirement

An N-body problem has a natural all-pairs decomposition called atom-decomposition [11] that is based on equal distribution of N element responsibilities to P parallel processes. To address load imbalances and the need to communicate all data to all nodes, the authors proposed a method to perform force-decomposition which still requires input data replication, but reduced it to two arrays of size $\frac{N}{\sqrt{P}}$ elements per process. The authors in [12] showed that data replication in the system can be variable (c); and when $c = \sqrt{P}$, a lower bound on communication is achieved. When $c = 1$, their solution behaved similar to atom-decomposition, although requiring only 2 arrays of $\frac{N}{P}$ elements per process. When $c = \sqrt{P}$, their solution behaved similar to force-decomposition and required 2 arrays of size $\frac{N}{\sqrt{P}}$ elements per process.

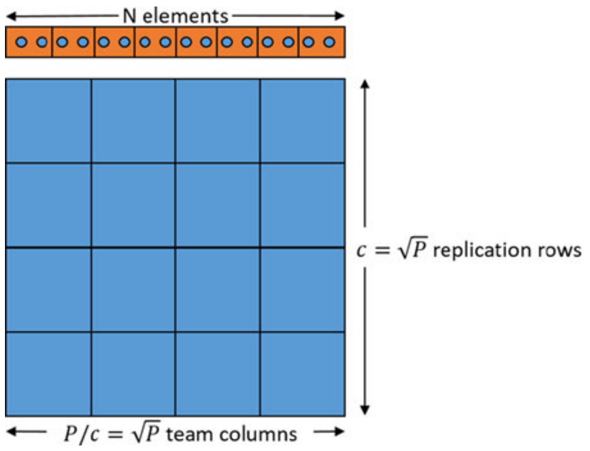

N elements
$c = \sqrt{P}$ replication rows
$P/c = \sqrt{P}$ team columns

Fig. 2. Driscoll et al. [12] communication optimal n-body algorithm's data replication and distribution. Optimality achieved when $c = \sqrt{P}$, resulting in $\sqrt{P}$ teams, $\sqrt{P}$ replication rows, and each processor performing the pairing between two size-$\frac{N}{\sqrt{P}}$ arrays of elements. Note that their algorithm's communication steps are not depicted in this figure.

2.2.3 Memory Management

A different approach was taken for a bioinformatics application seeking to reconstruct gene co-expression networks. The PCIT algorithm [13] is designed to identify significant gene correlations. The authors modified and optimized this algorithm for Intel's multicore Xeon and many-core MIC [14]. The optimization of the PCIT algorithm [14] experienced needing all of the data in memory and created a second optimization strategy with longer runtimes that had a minimal memory usage footprint by swapping data in and out of memory in a structured manner.

2.3 Distributed All-Pairs Problem

The distributed all-pairs problem distributes the $\binom{N}{2}$ element pairings across P processes. Methods to perform this distribution of work and data vary, e.g., [10], [11], [12], [15], [16]. Some implementations [15] give all of D to all processes and each process is responsible for a different portion of the $\binom{N}{2}$ element pairings. Other implementations have mechanisms to generate different data subsets and then compute the global pairing of all of D by pairing individual subsets.

In Fig. 2, Driscoll et al. [12] distribute N data elements evenly across $\frac{P}{c}$ team columns, i.e., forming $\frac{P}{c}$ smaller datasets. To achieve the communication optimal lower bound, they showed that the replication factor should be $c = \sqrt{P}$. Each member in a team column receives a dataset copy of the same $\frac{N}{\sqrt{P}}$ data elements. Then, an additional $\frac{N}{\sqrt{P}}$ data element array is copied and shifted from within their respective replication rows resulting in all nodes holding a pair of datasets (D_i, D_j). Their algorithm does this shift-copy in such a way to generate all of the respective data pairings. The algorithm works the best with $P = c^2$ processes. For $P = c \times r$, the algorithms needs more communication steps.

The general algorithm design approach for these problems takes N data elements and divides them into smaller dataset groups, often a division by the number of nodes P. However, [12] for example used divisions by $\frac{P}{c}$. Eq. (6) enumerates this set of P datasets. Each datasets is a subset of the original dataset D (Eq. (7)) and all elements from D must be present in at least one of the subsets (Eq. (8))

$$\text{Set of } P \text{ datasets } \hat{D} = \{D_0, D_1, \ldots, D_{P-1}\} \tag{6}$$

$$D_i \subseteq D, i \in 0, 1, \ldots, P-1 \tag{7}$$

$$D = \bigcup\nolimits_{i=0}^{P-1} D_i. \tag{8}$$

Eq. (9) describes the global pairing of all datasets D_i and D_j. This is in contrast to Eq. (5), which preforms the pairing of particular data elements d_i and d_j. Additionally, in Eq. (9) the range of index j is altered to allow for the pairing of D_i with itself. This is unnecessary in Eq. (5) because elements in general would not need to be paired with themselves; however once placed in a subset, it is still necessary that elements within the subset be paired with others within the same subset and not simply with only other subsets

$$Pair(D_i, D_j), \text{ where } 0 \leq i < P-1 \text{ and } i \leq j < P. \tag{9}$$

The global pairing work described must take place in a distributed manner, hence local to a process assigned o a computing node to perform the pairing of (D_i, D_j) must be datasets D_i and D_j in order to carry out the computation. The set of datasets available to a particular process is set S_i, where i is the process's id and S_i is a subset of $\hat{D}$ (Eq. (10)). All or a subset of the datasets available can then be paired to complete the distributed all-pairs problem. For example in [12], at any one time a process would only have two datasets available so all one dataset pairings were computed. Whereas in [15], all of the data was available to each process, so processes were assigned only a subset of the possible pairings

$$S_i \subseteq \hat{D}, i \in 0, 1, \ldots, P-1. \tag{10}$$

In the next section, we introduce the quorum sets that we use to reduce memory requirements per node in our work. These sets load balance the work and achieve minimal data replication and communication.

3 Quorum Sets

In distributed algorithms, coordination, mutual exclusion, data replication and consensus implementations generally group P processes (or nodes) into sets called quorums [17]. This organization is to minimize communications in operations like negotiating access to a global resource or reaching a joint, distributed decision.

Quorum's decentralized approach and slow quorum growth rate compared to the system size are two of the reasons that make them a good tool in managing replicated

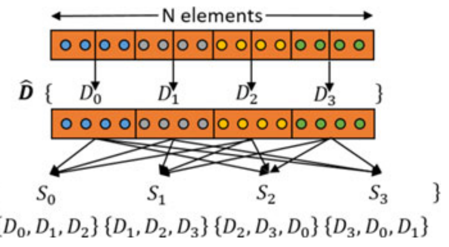

N elements
$\hat{D}$ { D_0 D_1 D_2 D_3 }
Q { S_0 S_1 S_2 S_3 }
$\{D_0, D_1, D_2\}$ $\{D_1, D_2, D_3\}$ $\{D_2, D_3, D_0\}$ $\{D_3, D_0, D_1\}$

Fig. 3. A quorum set example with N elements and four processes. Set $\hat{D}$ divides the N elements into 4 smaller datasets D_0 through D_3. Quorum set Q is then formed from sets (quorums) of these datasets, i.e., S_0 through S_3.

data [17]. Quorums of size $O(\sqrt{P})$ were proven optimal using finite projective planes in [18]. Relaxed difference sets (explained later) were used to create size $O(\sqrt{P})$ cyclic quorum sets in [4]. In a cyclic quorum set, all other sets can be derived using a base set as described later on.

A quorum set minimally has the following three properties for distributed algorithms: i) all quorums must intersect; (ii) each quorum must have equal work (the term "work" here refers to the tasks that must be done by/for the elements in that quorum set); iii) each element have equal responsibility within the quorum set (the term "responsibility" here refers to the number of times an element occurs in that quorums set). Maekawa [18] proposed the second and third properties as additional properties for quorums used for distributed algorithms. These were later re-phrased by Luk and Wong [4] using similar wording. Multiple different groupings of nodes or quorum sets exists. Not every quorum set will result in having these three properties, nor will the quorum sizes be minimal. Authors in [18] proved the lower bound on the size of a quorum set with these three properties. Cyclic quorum sets have these properties and additionally can be generated using a base quorum definition. We propose their use in this paper for efficient all-pairs problem computation and data replication management.

3.1 Quorum Sets for Data

We explain the quorum set for data using an example in Fig. 3. It provides a visual representation of a quorum set for $P = 4$ processes to perform computations on $N = 16$ elements. We divide data elements into $P = 4$ groupings. $\hat{D}$ is a set of four datasets, each containing a portion (exactly 4 in the example, N/P in general) of the N elements. For $P = 4$ processes, the optimal quorum size is three and the base quorum of $\{0, 1, 2\}$ is a solution. Thus for a quorum per process, S_0 through S_3, make up the quorum set Q. Each quorum can be seen to be a subset of the larger $\hat{D}$ and that all datasets in $\hat{D}$ are also present in Q. Additionally each set S_i set intersects with all other S_j sets (at least one dataset is common). For example S_0 shares dataset D_0 with both S_2 and S_3 and shares dataset D_1 with both S_1 and S_3. In this case, sets interaction include more than one dataset.

In general, let Q be a set of dataset subsets (Eq. (11)). Set Q is a quorum set, when Q's subsets covers all of $\hat{D}$ (Eq. (12)) and all subsets also have non-empty intersections (Eq. (13))

$$Q = \{S_0, \ldots, S_{P-1}\} \tag{11}$$

$$\bigcup_{i=0}^{P-1} S_i = \{D_0, \ldots, D_{P-1}\} = \hat{D} \tag{12}$$

$$S_i \cap S_j \neq \emptyset, \forall i, j \in 0, 1, \ldots, P-1. \tag{13}$$

The lower bounds for the maximum individual quorum size (i.e., $|S_i|$) in a minimum set is k, where Eq. (14) holds and $(k-1)$ is a power of a prime, proved through equivalence to finding a finite projective plane [18]. Solving for k in Eq. (14) results in a quorum size of approximately $k \approx \sqrt{P}$ datasets

$$P \leq k(k-1) + 1. \tag{14}$$

Additionally, it is desirable that each quorum S_i in the quorum set be of the same size (Eq. (15)), such that there is equal work. It also is desirable that each dataset D_i be contained in the same number of quorums (Eq. (16)), for equal responsibility. Both of these properties can be seen to hold by inspection in Fig. 3. All S_0 through S_3 contain the same $k = 3$ number of datasets (i.e., equal work) and counting the occurrences of each of the datasets D_0 through D_3 reveals that each occur $k = 3$ times in Q

$$|S_i| = k, \forall i \in 0, 1, \ldots, P-1 \tag{15}$$

$$D_i \text{ is contained in } k\ {S_j}'\text{s}, \forall i \in 0, 1, \ldots, P-1. \tag{16}$$

3.2 Cyclic Quorum Sets

Cyclic quorum sets are based on cyclic block design and cyclic difference sets. However, searching for optimal sets requires an exhaustive search [4]. Cyclic quorum sets are unique in that once the first quorum (Eq. (17)) is defined the remaining quorums in the set can be generated via incrementing the dataset indices (modulus to the number of datasets, as shown in Eq. (18)). For simplicity, assume $D_0 \in S_0$ without loss of generality (any one-to-one remapping of dataset indices can satisfy this assumption). This cyclic property can be seen in Fig. 3. The datasets contained in S_1 all have indices one greater than the datasets (modulus P) found in S_0. The same applied to S_2 an S_3

$$S_0 = \{D_0, \ldots, D_j\} \tag{17}$$

$$S_i = \{D_{0+i}, \ldots, D_{j+i}\}, \forall i \in 0, 1, \ldots, P-1. \tag{18}$$

For our work, we used the $P = 4, \ldots, 111$ optimal cyclic quorums from [4]. In the next section, we define and show a proof (our work in [5]) that cyclic quorum sets have an all-pairs property that makes them ideal for distributed all-pairs problems.

4 All-Pairs Property for Quorum Sets

Cyclical quorums were introduced in the previous section as having a small size ($|S_i| = k \approx \sqrt{P}$) and equal work/responsibility properties. However, it is not immediately evident how these small, equable cyclic quorum sets can support the pairing of all $\{D_0, \ldots, D_{P-1}\}$ distributed datasets to solve the general all-pairs problem (i.e., Eq. (9)). In this section we define the all-pairs property for quorum sets and provide a proof that cyclical quorum sets satisfy this property.

4.1 All-Pairs Property

As all-pairs algorithms scale using multiple processes and distribute the $\binom{N}{2}$ work, it remains necessary that all pairs of elements are present in that node memory which is executing that process in order to efficiently perform the pairing. In Section 2.3, we described how methods of distributing the work and data vary, e.g., [10], [11], [12], [15]. These all shared the same two basic attributes.

1) Each dataset pair is formed in some process (Eq. (9)).
2) Each process has a subset of the global data (Eq. (10)).

Some implementations [15] distributed all of D to all processes guaranteeing all element pairs can be formed. Other implementations have mechanisms to generate data subsets and permute them in a defined, predictable way to perform the global element pairing. Fig. 2 showed a high level example of the distribution developed by Driscoll et al. [12], where N data elements were evenly distributed and a shift copy permutation was used to generate all of the respective dataset pairings.

To define the all-pairs property for our distributed system, we first assign dataset S_i to process i (see Eq. (10)). The following constraint must hold for the all-pairs property to be satisfied

$$\exists S_i \ni (D_j, D_k)\ \forall j, k \in 0, 1, \ldots, P-1, \text{ where } S_i \in Q. \tag{19}$$

Eq. (19) states that for every pairing of datasets in $\hat{D}$ there exists at least one process's set S_i that contains the pair. Distributed systems with this all-pairs property can be used to satisfy Eq. (9) that defined the work necessary to compute the distributed all-pairs problem.

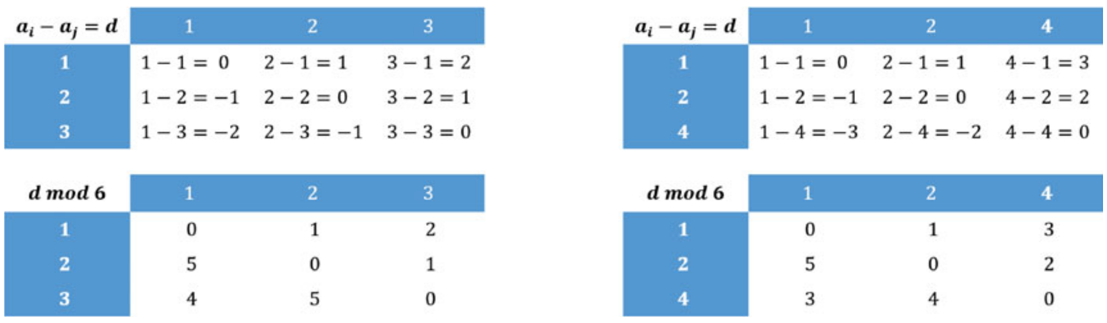
4.2 Cyclic Quorums have the All-Pairs Property

Our method utilizes cyclic quorums to satisfy the all-pairs property with minimal node resources and communication. Each process i is assigned a quorum S_i of datasets. Definition 1 and Theorems 2 and 3 proven by Luk et al. [4] establish the relationship between cyclic quorum sets and relaxed difference sets. We use this relationship as part of our proof in Theorem 4 that cyclic quorums satisfy the all-pairs property [5].

Definition 1. *Set $A = \{a_0, \ldots, a_k\}$ modulus $P, a_i \in 0, \ldots, P-1$ is a relaxed (P, k)-difference set if for every $d \neq 0$ modulus P, $\exists (a_i, a_j), a_i, a_j \in A$ such that $a_i - a_j = d$ modulus P.*

Definition 1 [4] defines a relaxed difference set as a set of integers whose values are greater than or equal 0 and less than P. It has a restriction that every integer from 0 to (P-1) must also be able to form from the difference of some pair of integers in the set (using modulus when necessary). Fig. 4 illustrates this definition through two examples. The example on the left forms all of the differences for $A = \{1, 2, 3\}$ on the top and then performs the modulus $P = 6$ on the bottom. This is not a valid difference set because $d\ mod\ 6 = 3$ is missing. The example on the right, however, is a valid difference set because the differences for $A = \{1, 2, 4\}$ modulus $P = 6$ form all integers $0, \ldots, (P-1)$.

Theorem 2. *The cyclic quorum set Q defined by set $S_i = \{a_0 + i, \ldots, a_k + i\}$ modulus $P, i \in 0, \ldots, P-1$ is a relaxed (P, k)-difference set $A = \{a_0, \ldots, a_k\}$ modulus $P, a_i \in 0, \ldots, (P-1)$.*



$a_i - a_j = d$	1	2	3
1	$1-1=0$	$2-1=1$	$3-1=2$
2	$1-2=-1$	$2-2=0$	$3-2=1$
3	$1-3=-2$	$2-3=-1$	$3-3=0$

d mod 6	1	2	3
1	0	1	2
2	5	0	1
3	4	5	0

(a)

$a_i - a_j = d$	1	2	4
1	$1-1=0$	$2-1=1$	$4-1=3$
2	$1-2=-1$	$2-2=0$	$4-2=2$
4	$1-4=-3$	$2-4=-2$	$4-4=0$

d mod 6	1	2	4
1	0	1	3
2	5	0	2
4	3	4	0

(b)

Fig. 4. Defining a relaxed (P, k)-difference set. For a given set $A = \{a_0, \ldots, a_k\}$ and integer P, all integers $0, \ldots, (P-1)$ must be formed from the differences modulus P of integer pairs from set A. Figure (a) on the left shows an invalid difference set corresponding to $A = \{1, 2, 3\}$ and $P = 6$ because no pair of integer differences modulus 6 formed $d \; mod \; 6 = 3$. However Figure (b) on the right with $A = \{1, 2, 4\}$ and $P = 6$ is a valid difference set because all integer differences modulus 6 were formed, i.e., $0, \ldots, 5$ are all present.

The intuition for Theorem 2 [4] relies on the quorum set's intersection property, $S_i \cap S_j \neq \emptyset, \forall i, j$ (Eq. (13)).

Proof. By contradiction, assume that set A was not a relaxed difference set, then there would be a value $d \neq 0$ that no difference $(a_i - a_j)$ modulus $P, a_i, a_j \in A$.

Given that every quorum intersects in the set Q (Eq. (13)), there must be a shared item in S_0 and S_d, where $d \in 0, \ldots, (P-1)$. Eq. (20) assumes the shared item is at indices i and j, respectively, hence the shared item $s_{0,i}$ and $s_{d,j}$, where $s_{0,i} \in S_i$ and $s_{d,j} \in S_j$, are differenced on the left-hand side. Using the cyclic quorum set definition, the values for the items are substituted on the right side. Eq. (21) uses the quorum intersection to simplify the left side to 0 before rebalancing to show that the assumption that there was no $d = (a_i - a_j)$ modulus P is false, hence set A is a relaxed difference set

$$S_{0,i} - S_{d,j} = (a_i + 0) - (a_j + d) \text{ modulus } P \tag{20}$$

$$a_i - a_j = d \text{ modulus } P. \tag{21}$$

□

Theorem 3. *The relaxed (P, k)-difference set $A = \{a_0, \ldots, a_k\}$ modulus P is a cyclic quorum set Q defined by set $S_i = \{a_0 + i, \ldots, a_k + i\}$ modulus $P, i \in 0, \ldots, P-1$ and $a_i \in 0, \ldots, (P-1)$.*

The intuition for Theorem 3 [4] again relies on the quorum set's intersection property, $S_i \cap S_j \neq \emptyset, \forall i, j$ (Eq. (13)).

Proof. By contradiction, assume that there were quorums S_x and S_y that did not intersect, i.e., they violated Eq. (13) and hence set Q was not a quorum set.

Quorums S_x and S_y both have elements at indices i and j respectively, where $i, j \in 0, \ldots, k$. Differencing these two elements results in Eq. (22), where on the right the values for the elements are substituted using the cyclic quorum set definition. To show that $S_{x,i} = S_{y,j}$ for some combination of indices i and j, we set the left side to zero and rebalance for Eq. (23). We are given that A is a relaxed difference set, so all differences $d \; mod \; P$ can be formed from elements in A. Hence, there must be some combination of indices i and j, where $i, j \in 0, \ldots, k$ that result in $a_i - a_j = y - x \; mod \; P = d \; mod \; P$. This result confirms that $S_{x,i} = S_{y,j}$ for some i and j and shows that our assumption that S_x and S_y did not intersect was false, hence set Q is a quorum set

$$S_{x,i} - S_{y,j} = (a_i + x) - (a_j + y) \text{ modulus } P \tag{22}$$

$$a_i - a_j = y - x \text{ modulus } P. \tag{23}$$

□

Theorem 4. *The cyclic quorum set Q defined by set $S_i = \{a_0 + i, \ldots, a_k + i\}$ modulus $P, i \in 0, \ldots, P-1$ satisfies the all-pairs property.*

Theorem 4 (developed by us in [5]) uses the relationship between all differences occurring in a difference set (Definition 1) and that cyclic quorums are based on difference sets (Theorems 2 and 3) to prove that cyclic quorums satisfy the all-pairs property.

Proof. By contradiction, assume that the all-pairs property is not satisfied. Then there must exist a pair of integers $(a_x, a_y), a_x, a_y \in 0, \ldots, P-1$ that are not present together in any quorum $S_i \in Q$.

Integer pair (a_x, a_y) have the following differences:

$$(a_x - a_y) \text{ modulus } P \text{ and } (a_y - a_x) \text{ modulus } P. \tag{24}$$

From Theorem 2, we know that $A = \{a_0, \ldots, a_k\}$ is a relaxed difference set and that all differences $d \neq 0$ modulus P exist. So, the differences formed by (a_x, a_y) are also formed at least once from the difference set A. Assume that integers $(a_i, a_j), a_i, a_j \in A$ form those specific differences

$$(a_i - a_j) \text{ modulus } P \text{ and } (a_j - a_i) \text{ modulus } P. \tag{25}$$

Using the cyclic quorum set definition and distributive property of modular arithmetic, Eq. (26) shows all S_i cyclic quorums form the same differences. So the differences in Eq. (24), formed by the integer pair (a_x, a_y) not present together in any quorum, are formed in every cyclic quorum S_i

$$\begin{aligned} (a_i - a_j) \text{ mod } P &= (a_i + m) - (a_j + m) \text{ mod } P \\ &= S_{m,i} - S_{m,j} \text{ mod } P \quad \text{and} \\ (a_j - a_i) \text{ mod } P &= (a_j + m) - (a_i + m) \text{ mod } P \\ &= S_{m,j} - S_{m,i} \text{ mod } P, \\ \forall m \in 0, \ldots, P-1. \end{aligned} \tag{26}$$

Eq. (26) reveals that all cyclic quorums have the missing differences in Eq. (24). The cyclic $a_i + m$ modulus P

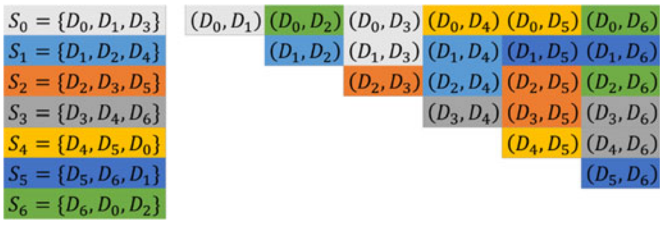

$S_0 = \{D_0, D_1, D_3\}$
$S_1 = \{D_1, D_2, D_4\}$
$S_2 = \{D_2, D_3, D_5\}$
$S_3 = \{D_3, D_4, D_6\}$
$S_4 = \{D_4, D_5, D_0\}$
$S_5 = \{D_5, D_6, D_1\}$
$S_6 = \{D_6, D_0, D_2\}$
(D_0, D_1) (D_0, D_2) (D_0, D_3) (D_0, D_4) (D_0, D_5) (D_0, D_6)
(D_1, D_2) (D_1, D_3) (D_1, D_4) (D_1, D_5) (D_1, D_6)
(D_2, D_3) (D_2, D_4) (D_2, D_5) (D_2, D_6)
(D_3, D_4) (D_3, D_5) (D_3, D_6)
(D_4, D_5) (D_4, D_6)
(D_5, D_6)

Fig. 5. A cyclic quorum set example with seven processes. On the left are the seven quorums and on the right are all of the dataset pairings. The quorums and the corresponding pairs formed are colored. As Theorem 4 states, all pairs have been covered by a quorum set.

definition guarantees there is an m that $a_i + m$ modulus P equals a_x and the difference with a_j will still hold such that $a_j + m$ modulus P equals a_y too

$$a_x = (a_i + m) \text{ modulus } P = S_{m,i}$$
$$a_y = (a_j + m) \text{ modulus } P = S_{m,j}. \tag{27}$$

Eq. (27) show that integers $(a_x, a_y), a_x, a_y \in 0, \ldots, P-1$ are present together in quorum S_m defined by difference set A and integer m modulus P. This contradicts the assumption that the pairs are not present together, hence cyclic quorum sets do satisfy the all-pairs property. □

Fig. 5 is a visualization of cyclic quorum sets having the all-pairs property. The quorums on the left are color in various shades and the corresponding dataset pairs that can be formed are colored that same shade on the right. All dataset pairs on the right are covered. Also note that all quorums performed the same amount of work and were able to cover the same number of pairs, so the data distribution results in load balanced work.

To follow Theorem 4 at a high level with Fig. 5, one can pick any two numbers in $0, \ldots, 6$, e.g., 2 and 6. Making the assumption 2 and 6 are not present together would result in the all-pairs property not being satisfied. Notice that quorum S_0 has datasets with indices corresponding to the difference set $A = \{0, 1, 3\}$. The difference of the two chosen numbers (modulus 7 as necessary), e.g., $2 - 6 = -4 \; mod \; 7 = 3$, is then used to find the pair of numbers in $A = \{0, 1, 3\}$. The difference and modulus 7 of this pair results in the same difference, e.g., $3 - 0 = 3 \; mod \; 7 = 3$. Lastly, beginning with that pair from A just found, e.g., 3 and 0, one can begin incrementing both numbers (modulus 7 as necessary) until finding the original pair of numbers assumed not to be present together, e.g., 2 and 6. Finding the pair of numbers in the same quorum set proves the assumption was false and the all-pairs property holds.

5 Cyclic Quorums Application Results

In this section, we first present the results of our evaluation of the directly applying the cyclic quorum set method. In the next section, we further improve the method to avoid duplicate computation that are inherent in the structure of quorums (as can be seen in Fig. 3). To evaluate the performance and demonstrate the gains, we modified an existing all-pairs application [14] to scale to larger datasets and at the same time be able to utilize more resources. The algorithm implemented the distributed all-pairs problem using the cyclical quorum sets defined in Section 3.2.

TABLE 1
Input Datasets Utilized in PCIT Experiments

Type	Rows	Columns
***Cattle**	27,364	5
Simulated	33,331	5
Simulated	39,298	5
***Mice**	45,265	5
Simulated	51,232	1,893
***Rice**	57,194	1,893
Simulated	63,166	1,893
Simulated	69,133	1,893
Simulated	75.000	1,893

5.1 Bioinformatics PCIT Application

The partial correlation coefficients combined with an information theory approach (PCIT) algorithm was introduced by [13]. The algorithm can be used as a component to the work flow for gene co-expression network reconstruction and for helping to identify novel biological regulators. This technique processes N genes (and candidate genes) by building an $O(N^2)$ matrix and using a guilt-by-association heuristic to analyze gene pair partial correlations to identify using purely data whether a gene expression correlation is or is not meaningful. Thus far the problem has been solved when all data are kept in memory or strategically moved in and out of memory in a single computing (multi-core) node.

5.2 Test Setup

As it true in all research fields, the field of bioinformatics is no stranger to having to scale their algorithms to larger datasets by utilizing more resources that are available in today's cloud systems [7], [19], [20], [21] or high performance computing systems.

We conducted our application experiments using CyEnce, an HPC system at Iowa State University (http://www.hpc.iastate.edu/systems). Every node has dual Intel Xeon E5 8-core processors and 128 GB of memory. Our executions ranged from 16 to 512 cores (1 to 32 nodes). Although nodes in CyEnce include 128 GB memory per node (8 GB/core), we restricted our application's memory usage to only 60 GB per node.

Three real and six simulated input datasets were used in our testing. Table 1 marks the real datasets with an asterisk. Simulated datasets were generated for a particular number of rows and columns using a technique published in [13]. Number of input rows (N) is the primary determinate of the processing complexity of all-pairs algorithms, hence our input sizes were varied with increasing number of gene (and gene candidate) rows. The number of input columns for the simulated datasets were chosen to match that of similar real input datasets. Input columns correspond to the number of test subject conditions, e.g., the number of cattle, mice, or rice test samples.

The single node PCIT algorithm from [14] was run with 16 OpenMP threads on a node by itself. The quorum implementations ran with 4 to 32 nodes (64 to 512 cores) with one MPI process and 16 OpenMP threads per node. The single node performance is used as bench mark to compare multiple node performance for speed up. The number of HPC nodes was varied to demonstrate interesting strengths and

TABLE 2
Memory Sized Required/Used Per Node (GB)

#Nodes	*27,364	33,331	39,298	*45,265	51,232	*57,194	63,166	69,133	75,000
1	25.113	37.257	51.789	**68.708**	**88.736**	**110.495**	**134.680**	**161.233**	**189.669**
4	18.831	27.939	38.837	51.526	**66.725**	**83.064**	**101.225**	**121.159**	**142.506**
7	10.762	15.967	22.193	29.445	38.439	47.812	58.224	**69.651**	**81.888**
8	12.555	18.627	25.893	34.352	44.724	55.647	**67.782**	**81.099**	**95.357**
13	7.727	11.463	15.934	21.140	27.801	34.554	42.054	50.282	59.091
16	7.848	11.643	16.185	21.471	28.224	35.083	42.698	51.054	59.996
31	4.862	7.213	10.025	13.302	17.760	22.035	26.787	31.994	37.563
32	5.495	8.151	11.331	15.033	19.974	24.802	30.156	36.032	42.316

weaknesses in the choice of number of parallel nodes selected. Common choices for number of parallel nodes are powers of 2, i.e., 4, 8, 16, and 32 nodes. However, cyclic quorum sets based on Singer difference sets [6] can have an advantage (more discussion on this in Section 6), so 7, 13, and 31 nodes are also tested.

5.3 Results

For each pair of dataset input and number of HPC nodes, we executed 30 runs of the application in order to establish confidence in the runtime measurements. The two parameters of interest are: memory usage and execution time. The observed results are listed in Tables 2 and 3, respectively. In each table, there is a row for the number of nodes used. The columns in the two tables list the memory size used per compute node and the execution runtimes with 95 percent confidence intervals for each input dataset, respectively.

5.3.1 Memory Usage Performance

The single node instance in Table 2 is the PCIT algorithm from [14] for comparison. As additional nodes are used, our cyclic quorums implementation requires fewer memory resources per node to execute the same dataset. This is because not all of the input, intermediate, or result data are stored at a single node. Instead, the data is distributed in a predictable way using the cyclic quorum design described in Section 3.2. This resulted in up to an 80 percent reduction in memory requirements per node. This was a critical result for our larger real and simulated input datasets where memory used per node ran into the upper 60 GB limit for the single node tests and some of our parallel tests as well. These instances have been marked with bold in Table 2.

This table only include the memory used per node (GB). This is the calculated amount of memory needed/required to run the optimized algorithm. Bold items are where the input data would have exceeded a set threshold. Also, note that the optimized single node algorithm was run with low-memory version when the threshold was exceeded, but only the one dataset (45,265) ever finished within a reasonable amount of time to perform 30 runs to get the 95 percent confidence intervals.

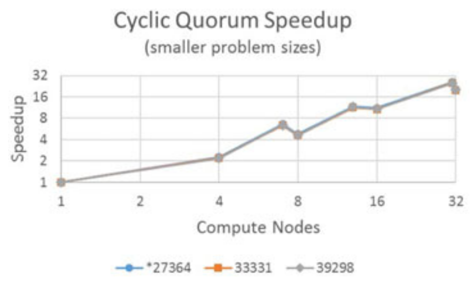
While generally increasing the number of nodes will reduce the memory used, there are some local minimums that initially may not be expected by a user of our cyclic quorums algorithm. For example, choosing seven nodes instead of eight actually results in a lower memory usage per node (57 percent versus only 50 percent reduction), which is a bit counter intuitive upon first inspection. In the $N = 63,166$ row test, this resulted in the eight-node execution exceeding the 60 GB memory limit, while with one fewer nodes, the seven-node test stayed under the limit. This can be explained by cyclic quorum sets based on Singer difference sets [6] can have some advantages like requiring fewer datasets (D_i) to still guarantee the all-pairs property (Section 4). So 7, 13, and 31 nodes reduce memory usage per node by 57, 69, and 80 percent respectively, while if the number of nodes were increased by a small amount to 8, 16, or 32 the reduction is slightly less at 50, 68, and 77 percent. More discussion on this topic is in Section 6.

5.3.2 Runtime Execution Performance

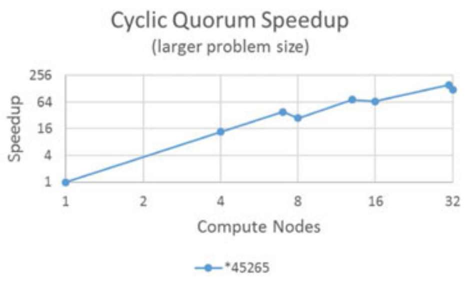
Similar to the analysis of the memory, in Table 3 the single node instance used the PCIT algorithm from [14]. Our cyclic quorums implementation benefits from the additional parallel processing nodes. The all-pairs computation work is equally distributed and the resulting average runtimes on the same datasets decrease significantly. The average runtimes observed over 30 executions are in Table 3 with their 95 percent confidence intervals. As described in our test setup (Section 5.2), we used a generous 60 GB memory limit

TABLE 3
Average Execution Runtimes (Seconds)

#Nodes	*27,364	33,331	39,298	*45,265	51,232	*57,194	63,166	69,133	75,000
1	119.1 ± 2.9	162.5 ± 0.1	245.5 ± 0.1	**$2{,}312.5 \pm 0.2$**	-	-	-	-	-
4	52.1 ± 0.0	73.3 ± 0.1	112.0 ± 0.5	170.6 ± 0.1	-	-	-	-	-
7	17.9 ± 0.0	25.5 ± 0.0	38.2 ± 0.0	60.5 ± 0.0	317.5 ± 0.9	$1{,}209.8 \pm 2.0$	490.8 ± 2.3	-	-
8	24.8 ± 0.0	35.3 ± 0.0	53.2 ± 0.0	83.1 ± 0.0	364.2 ± 1.2	$1{,}631.4 \pm 2.2$	-	-	-
13	10.1 ± 0.0	14.4 ± 0.0	21.3 ± 0.0	32.1 ± 0.0	246.3 ± 2.4	784.8 ± 1.5	372.8 ± 2.1	446.1 ± 1.2	522.8 ± 2.9
16	10.6 ± 0.0	15.2 ± 0.0	22.5 ± 0.0	35.1 ± 0.0	251.2 ± 2.3	827.0 ± 2.0	377.9 ± 2.1	454.2 ± 1.2	530.5 ± 2.0
31	4.6 ± 0.0	6.4 ± 0.0	9.9 ± 0.1	14.7 ± 0.0	178.6 ± 1.2	407.8 ± 2.5	250.1 ± 1.8	306.4 ± 1.1	361.1 ± 1.5
32	5.8 ± 0.0	8.3 ± 0.0	12.2 ± 0.0	18.9 ± 0.0	197.9 ± 1.2	516.4 ± 1.2	284.6 ± 0.7	337.4 ± 2.1	394.6 ± 2.2



(a)



(b)

Fig. 6. Speedup of our cyclic quorum algorithm using (P) parallel nodes when compared to an optimized single node algorithm. Figure (a) on the left has near identical speedup curves for the computation of three smaller datasets that all fit within the available memory. The log-log scale shows that as additional node resources are added our algorithm scales near linearly to utilize the resources. Figure (b) on the right has a speedup curve for a larger dataset that would have exceeded the single node memory resources, requiring the use of an alternate lower memory optimized algorithm. Here super-linear speedup is observed as our cyclic quorum algorithm is able to distribute the problem and process the input within the memory constraints.

that even current datasets pushed up against (Table 2), and if trends continue, future datasets will push even further beyond. Where this limit was exceeded we did not collect execution runtime data, except in one single node instance which is marked with bold. Here the single node algorithm had to revert to a lower memory alternative and consequently the runtime increased dramatically as a trade off.

The speedup (or scalability) of our algorithm is calculated as how many times faster our algorithm runs given additional processing nodes (P) with respect to the optimized single node algorithm, i.e., $SpeedUp = \frac{Time_{single}}{Time_{parallel}(P)}$. Fig. 6a shows the speedup curves for three smaller datasets all of which fit within the available memory constraints. The curves are nearly identical for the datasets, hence the increased execution speed is independent of the dataset and can be attributed to our algorithm's ability to distribute the data and all-pairs computation work. The trend of the curves on the log-log scale is close to linear demonstrating the ability to utilize additional resources to arrive at results in a shorter amount of time efficiently. The speedup is not ideally linear however, as seen by the speedup value falling slightly short of the number of nodes used. As expected, this indicates there are some overheads when parallelizing across multiple nodes that was not present in the optimized single node algorithm.

Additionally, the curves are not smooth with the increasing nodes used, i.e., speedup of $P = 7, 13,$ and 31 exceeds that of $P = 8, 16,$ and 32. This can be explained by cyclic quorum sets based on Singer difference sets [6] can have some advantages like requiring fewer datasets (D_i) to guarantee the all-pairs property (Section 4), hence there are fewer datasets at each node being paired with each other meaning less work to be performed. This is an interesting result and creates the question of whether additional computation management could improve the performance of quorum sets not based on Singer difference sets, which is the topic in Section 6.

Without the distributed cyclic quorums set solution, the time to compute the ever increasing dataset sizes is prohibitive. To further illustrate this point, Fig. 6b shows the speedup curve for the $N = 45,265$ row dataset. The trend of the curve on the log-log scale is close to linear, again demonstrating our ability to scale to arrive at results in a shorter amount of time efficiently. However, unlike Fig. 6a, the speedup in Fig. 6b is super-linear as seen by the speedup value for some inputs being more than 5x the number of nodes used. This is a common phenomenon where distributed parallel algorithms are able to more efficiently process a larger problem than their single node equivalents. Meaning that not only our solution compute the all-pairs results faster, it will also do so more cheaply by using as much as 5x fewer node compute hours to complete the computation.

When the datasets and generated intermediate/result data fit inside a single node's memory, from Fig. 6a, we see that our parallel algorithm has speed up that scales linearly with the addition of more nodes. There is some overheads with parallelizing across a cluster that the speed up is slightly below ideal 1:1 when compared to the optimized single node algorithm. When the datasets and generated intermediate/result data do not fit inside a single node's memory, the single node algorithm must use an optimized, lower memory version to perform the computation. In Fig. 6b, we see that our algorithm continues to scale, but this time the single node is running slower on this larger dataset.

Lastly, again the speedup curve in Fig. 6b is not smooth just like that of Fig. 6a. This is an interesting result highlighting some of the advantages of Singer difference sets. Section 6 introduces a communication-less management technique to make even non-Singer difference sets efficient.

6 Managing Redundant Computations

In the prior section, Section 5.3, it was observed that our cyclic quorums algorithm scaled well using less memory per node and achieving near linear and at times super-linear speedups. However, the speedup when additional nodes were added was not as smooth as anticipated in Fig. 6 and similarly observed in Table 3 with executions for $P = 8,$ $16,$ and 32 under performing expectations.

In this section, we discuss how the size of the cyclic quorum impacts these results, as well as providing additional computation management logic to compliment the data management techniques provided by the cyclic quorum sets to achieve further efficiency. We also implement and evaluate the effectiveness of our management logic.

6.1 Impact of Cyclic Quorum Size

Common choices for number of parallel nodes are powers of 2, i.e., 4, 8, 16, and 32 nodes. These choices were shown to not perform as well as lesser obvious choices of 7, 13, and 31 nodes for cyclic quorum sets. In Section 5, we attributed this

TABLE 4
Redundant Work Performed When Quorum Pairs Unmanaged

#Nodes	$\mathbf{\lvert S_i \rvert = k}$	Ideal Pairs	Unmanaged Pairs	Redundant Pairs	Redundant Percentage
4	3	10	16	6	37.5%
7	3	28	28	0	0.0%
8	4	36	56	20	35.7%
13	4	91	91	0	0.0%
16	5	136	176	40	22.7%
31	6	496	496	0	0.0%
32	7	528	704	176	25.0%
Average Redundant Percentage for $\#\text{Nodes} = 4, \ldots, 111$					**19.7%**

to the advantages of cyclic quorum sets based on Singer difference sets [6]. We define them (7, 13, and 31 nodes) as the *perfect numbers of processes for cyclic quorum sets* as they result into each pair being computed exactly once.

Ideally one would like to compute each pair interaction exactly once. However, as noted earlier, the choice of the number of processes (and computing nodes) and the corresponding cyclic quorum leads to some pairs being computed in more than one processes. That leads to extra work. In this section, we pay closer attention to the size of the optimal cyclic quorum sets used for $P = 4, \ldots, 111$ and how much extra work they create. Please note that even with the redundant work done, our algorithms scales very well and serves it purpose efficiently.

Table 4 summarizes the characteristics of our test parameter P's impact on the total number of all-pairs computations performed. Column 1 is the number of P nodes. $|S_i| = k$ comes from Eqs. (14) and (16). There is a minimum size that the cyclic quorum can have, while still being valid and hence having the all-pairs property (Section 4). We verified that the optimal cyclic quorums found from [4] were in fact the minimum and column 2 contains these quorum sizes. One of the attractive features is the slow quorum growth rate, $O(\sqrt{P})$, compared to P. This means that the size of the quorum set, i.e., $|S_i| = k$, will grow far slower than the increase in number of P nodes being used.

The distributed all-pairs problem was defined in Section 2.3. Additionally, Eq. (9) described the required global pairing of all D_i for $i \in 0, \ldots, P-1$. What this amounts to is $\binom{P}{2} = \frac{P(P-1)}{2}$ datasets paired with each other and then the same P datasets paired with themselves for a total ideal pairing performed being:

$$\frac{P(P-1)}{2} + P = \frac{P(P+1)}{2}.$$

This value is calculated for each P-nodes in Table 4 and listed in column 3, under the title "Ideal Pairs."

The Unmanaged Pairs column is the total number of pairs computed in our cyclic quorums algorithm without any additional management logic. Each node has k datasets available in memory, where the specific datasets available are defined by Eq. (18). Hence, they can perform $\binom{k}{2} = \frac{k(k-1)}{2}$ pairings and 1 additional pairing of dataset D_{0+i} with itself. When this is performed across all nodes the total unmanaged pairings is

$$P\left(\frac{k(k-1)}{2} + 1\right).$$

This value is calculated for each P-nodes in Table 4 and listed put in column 4, under the title "Unmanaged Pairs."

The "Redundant Pairs," column 5, of Table 4 is the difference between the ideal number of global pairs computed in the distributed system and the number of pairs computed globally (by all nodes) when the cyclic quorum set algorithm is executed without any additional management.

The "Redundant Percentage" is the ratio of the redundant pairs to unmanaged pairs computed that is redundant. The computation is detailed for some values of P. On average for all values of $P = 4, \ldots, 111$, the amount of redundant pairs is significant at 19.7 percent. This really motivates the need for adding computation management logic to our cyclic quorums set solution to achieve higher efficiency.

In Table 4, there are several entries that have 0 redundant pairs. These instances are where the cyclic quorum is formed from a Singer difference set. The distributed global pairings across all of the processing nodes in these instances perfectly cover the necessary pairs without any overlap. An example of a seven-node cyclic quorum covering all pairs can be seen in Fig. 5. Singer difference sets for $P < 100$ are $P = 7, 13, 21, 31, 57, 73,$ and 91. As defined earlier, we term them as the prefect number of processes for cyclic quorum sets based solution.

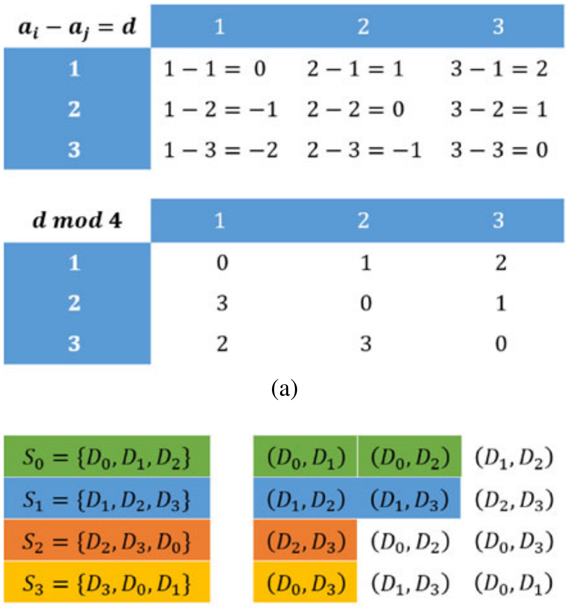
6.2 Computation Management Logic

The previous section identified the issue as distributed nodes performing redundant work and provided a way to account for how much wasted work actually exists in the solution. The next challenge is to locate the source and fix it in a distributed manner.

6.2.1 Identifying Redundancy

Observe that in order to perform redundant all-pairs work two or more nodes must share (intersect) two or more of the same datasets. Section 4 defined the all-pairs property for quorums and has the corresponding proof that cyclic quorums have the all-pairs property. Notice the key to forming pairs is rooted in the difference set for the cyclic quorum. It is this difference set that defines which datasets are in each quorum, hence we return to Definition 1 (Relaxed (P, k)-difference set).

In Fig. 7a, every $d \neq 0$ modulus $P = 4$ is present, which is required for $A = \{1, 2, 3\}$ to be a difference set (and cyclic quorum). The key observation is that there are instances that differences modulus P occur more than once. This is the source of all of the redundant work for $P = 4$. To see



$a_i - a_j = d$	1	2	3
1	$1-1=0$	$2-1=1$	$3-1=2$
2	$1-2=-1$	$2-2=0$	$3-2=1$
3	$1-3=-2$	$2-3=-1$	$3-3=0$

d mod 4	1	2	3
1	0	1	2
2	3	0	1
3	2	3	0

(a)

$S_0 = \{D_0, D_1, D_2\}$	(D_0, D_1)	(D_0, D_2)	(D_1, D_2)
$S_1 = \{D_1, D_2, D_3\}$	(D_1, D_2)	(D_1, D_3)	(D_2, D_3)
$S_2 = \{D_2, D_3, D_0\}$	(D_2, D_3)	(D_0, D_2)	(D_0, D_3)
$S_3 = \{D_3, D_0, D_1\}$	(D_0, D_3)	(D_1, D_3)	(D_0, D_1)

(b)

Fig. 7. A difference set and cyclic quorum set example with four processes to illustrate the cause and solution to redundant work. Figure (a) on the top with $A = \{1, 2, 3\}$ and $P = 4$ is a valid relaxed difference set because all integer differences modulus 4 are formed, i.e., $0, \ldots, 3$ all occur one or more times. Figure (b) on the bottom has the corresponding cyclic quorum set on the left and the possible all-pairs formed for each on the right. The pairs and quorum are colored with useful work performed, while the uncolored pairs would be redundant work and should not be performed.

why reoccurring differences lead to duplicate pairs, consider for some difference set A

$$a_i - a_j \; mod \; P = a_x - a_y \; mod \; P.$$

Using the definition of cyclic quorums these differences would correspond to pairs for some quorum S_l and S_m

$$\left(a_i + l, a_j + l\right) \bmod P$$
$$\left(a_x + m, a_y + m\right) \bmod P.$$

Knowing that both pairs correspond to equal differences modulus P, then we can add k modulus P to the first difference to obtain the second difference

$$(a_i + k) - (a_j + k) \; mod \; P = a_x - a_y \; mod \; P.$$

This gives way to creating a new pair equivalence

$$\left(a_i + k + l, a_j + k + l\right) \bmod P = \left(a_x + m, a_y + m\right) \bmod P.$$

And now we can define a new quorum index $\hat{l} = l + k \bmod P$

$$\left(a_i + \hat{l}, a_j + \hat{l}\right) \bmod P = \left(a_x + m, a_y + m\right) \bmod P.$$

The result is two quorums $\hat{l}$ and m with a redundant pair all because there was a reoccurring difference in their shared difference set definition A.

6.2.2 *Removing the Redundancy*

With the source of the redundancy identified, ideally we would just remove it. However, we cannot change the difference set to eliminate the redundancy without violating the definition. Instead, we take the simple approach above to identify any redundancy. When one is found, we must decide how to handle it differently than the other unique differences.

To get an idea what this different handling would look like consider Fig. 7b. All of the cyclic quorums are listed on the left for the difference set shown in Fig. 7a. On the right the $\binom{k}{2}$ pairs for each quorum are enumerated using the following procedure:

```
Given: Array $A$
for $x \leftarrow 0$ to $length(A) - 2$do
    for $y \leftarrow x + 1$ to $length(A) - 1$ do
        Pair $\left(D_{a_x + i}, D_{a_y + i}\right)$
    end for
end for
```

This enumeration pattern uses Eq. (18) and gives a consistent order to forming the pairs for all quorums $S_i, i \in 0, \ldots, P-1$. The first pair for each quorum is based on differences $a_0 - a_1$ and $a_1 - a_0$. These differences have not been computed yet, hence their quorum dataset pairs are not redundant work, so we color them with the corresponding quorums' color in Fig. 7b. This is important to observe because the differences

$$a_0 - a_1 \; mod \; 4 = 1 - 2 \; mod \; 4 = 3, \text{ and}$$
$$a_1 - a_0 \; mod \; 4 = 2 - 1 \; mod \; 4 = 1.$$

occur multiple times in Fig. 7a. The repetition of differences was identified as being the source of redundant work. However, the work is not redundant yet, so the work must be performed.

The second pair for each quorum is based on differences $a_0 - a_2$ and $a_2 - a_0$. These pairs have not been computed yet either, but on a closer inspection the differences

$$a_0 - a_2 \; mod \; 4 = 1 - 3 \; mod \; 4 = 2, \text{ and}$$
$$a_2 - a_0 \; mod \; 4 = 3 - 1 \; mod \; 4 = 2,$$

are actually the same difference. This redundancy with itself occurs when $P \; mod \; 2 = 0$, i.e., the number of nodes is even. Hence, if all nodes were to compute this pair, redundant work will occur. Rather, we arbitrarily decide that the first half of the nodes corresponding with quorums $S_i, 0 \le i < \frac{P}{2}$ should do the work, so we color them with the corresponding quorums' color. We leave the others (would be redundant pairs) uncolored to indicate that they are not to be computed.

The last quorum dataset pair is based on differences

$$a_1 - a_2 \; mod \; 4 = 2 - 3 \; mod \; 4 = 3, \text{ and}$$
$$a_2 - a_1 \; mod \; 4 = 3 - 2 \; mod \; 4 = 1.$$

However, these differences have already been computed and hence are redundant and are not colored in Fig. 7b.

We took a simple approach described above and in Fig. 7b to prevent redundancy at the time of computation, since it cannot be eliminated from the difference set. This computation management logic is more formally stated by the algorithm in Fig. 8. First, computing differences multiple times is the source of redundancy. Line 2 is an array of booleans to track which differences have been computed and

```
1:  procedure MANAGEMENT LOGIC($A$,$P$,$i$)
2:      $DC[d] \leftarrow False,\ d \in 0,\ldots,P-1$
3:      for $x \leftarrow 0$ to $length(A)-2$ do
4:          for $y \leftarrow x+1$ to $length(A)-1$ do
5:              $d_{xy} \leftarrow (A[x]-A[y])$ modulus $P$
6:              $d_{yx} \leftarrow (A[y]-A[x])$ modulus $P$
7:              if $DC[d_{xy}] \neq True$ and
                $DC[d_{yx}] \neq True$ then
8:                  $DC[d_{xy}] \leftarrow True$
9:                  $DC[d_{yx}] \leftarrow True$
10:                 if $d_{xy} \neq d_{yx}$ then
11:                     Compute Pair $(D_{A[x]+i}, D_{A[y]+i})$
12:                 else if $i < \frac{P}{2}$ then
13:                     Compute Pair $(D_{A[x]+i}, D_{A[y]+i})$
14:                 end if
15:             end if
16:         end for
17:     end for
18: end procedure
```

Fig. 8. Redundant all-pairs work can be eliminated with simple management logic. This algorithm enumerates all differences in set A and performs the computation only if it is the first time computing the difference. A special case is handled for even number P nodes, when the differences are equal.

which have not. Lines 3 and 4 enumerate all $\binom{k}{2}$ difference pairs. The prevention of redundancy logic is primarily in the if statement on Line 7, which verifies whether a difference has been computed or not. For those differences that have not been computed, most will go on to compute the quorum dataset pairings on Line 11. However, there is a special case for when differences form redundancies with themselves, i.e., $d_{xy} = d_{yx}$. This only occurs when the number of P nodes is even, and in this case only half of the nodes need to compute their quorum dataset pairs (Line 12).

6.3 Impacts of Managing Computations

Table 4 illustrated that there was room for substantial improvement in all-pairs computations using cyclic quorum sets not based on Singer difference sets. On average 19.7 percent of the work would be redundant if something were not done to avoid it.

One way to handle this is to implement a central manager that assigns work and arbitrates which node performs which computation pairs, but that adds another layer of complexity and overhead, potentially negating any benefits. A decentralized communication approach without a leader would appear to be a good fit, considering those applications are what quorum sets are regularly used for. However, that approach would also add an unnecessary layer of complexity and overheads.

We implement the algorithm in Fig. 8 in a distributed system without any communication requirement. All decisions are made locally and with very little overhead other than keeping track of the Differences Computed DC array.

To test out this computation management logic, we used the same application and same test setup as described in Sections 5.1 and 5.2. Once again we restricted our application's memory usage to only 60 GB per node to model. The three real and six simulated input datasets described in Table 1 were used again as well.

We modified our prior cyclic quorum to add the computation management logic. For each pair of dataset input and number of HPC nodes, we executed 30 runs of the application in order to establish confidence in the runtime measurements. There is a column for every dataset input and a row for the number of nodes used. The memory used per node is same as in Table 2 and the observed average execution runtimes with 95 percent confidence intervals is in Table 5.

6.3.1 Memory Usage Performance

The single node instance is the PCIT algorithm from [14] for comparison. Our cyclic quorums implementation with the new management logic has a substantially similar memory footprint to the implementation without the management. This was expected considering that the redundancies were not able to be removed from the difference sets, i.e., the data continues to be distributed in a predictable way using the cyclic quorum design described in Section 3.2. Also, we only added a small amount of overhead to keep track of which differences had, and which ones had not yet, been computed.

We continue to see that as additional nodes are used, fewer memory resources per node are required to execute the same dataset. This again results in up to an 80 percent reduction in memory requirements per node. This was a critical result for our larger real and simulated input datasets where memory used per node ran into the upper 60 GB limit for the single node tests and some of our parallel tests as well. These instances have been marked with bold in Table 2.

The advantages of using Singer difference sets [6] is still present in the memory usage data for the same reason. $P =$ 7, 13, and 31 nodes reduce memory usage per node by 57, 69, and 80 percent, respectively, while if the number of nodes were increased by a small amount to 8, 16, or 32 the reduction is slightly less at 50, 68, and 77 percent. Generally increasing the number of nodes will reduce the memory used. However, these results illustrate that there are some local minimums that initially may not be expected by a user of our cyclic quorums algorithm.

6.3.2 Runtime Execution Performance

The average runtimes observed over 30 executions are in Table 5 with their 95 percent confidence intervals. As described in our test setup (Section 5.2), we used a generous 60 GB memory limit that even current datasets pushed up against (Table 2). Where this limit was exceeded we did not collect execution runtime data, except in one single node instance which is marked with bold. Here, the single node algorithm had to revert to a lower memory alternative and consequently the runtime increased dramatically as a trade off.

Fig. 9 is a comparison of speedup between our unmanaged cyclic quorum set implementation of the PCIT algorithm from Section 5 and a modified version that has additional computation management logic. The unmanaged implementation is denoted with a "U" and managed implementation with an "M". Speedup curves for two datasets are shown in the figure. A smaller $N = 39,298$ rows dataset which fits within the available memory limit, and a larger $N = 45,265$ rows dataset which exceeded the memory limit of a single node.

The trend of the $N = 39,298$ curves on the log-log scale are close to linear demonstrating both implementations' abilities to scale with increasing node resources. The

TABLE 5
Managed-Average Execution Runtimes (Seconds)

#Nodes	*27,364	33,331	39,298	*45,265	51,232	*57,194	63,166	69,133	75,000
1	119.1 ± 2.9	162.5 ± 0.1	245.5 ± 0.1	**$2{,}312.5 \pm 0.2$**	-	-	-	-	-
4	38.4 ± 0.1	54.0 ± 0.1	82.4 ± 0.2	124.0 ± 0.0	-	-	-	-	-
7	18.0 ± 0.0	25.4 ± 0.1	38.3 ± 0.1	60.4 ± 0.1	317.6 ± 0.9	$1{,}213.4 \pm 1.6$	491.9 ± 2.3	-	-
8	18.0 ± 0.0	25.7 ± 0.1	38.6 ± 0.1	57.9 ± 0.1	341.5 ± 1.0	$1{,}175.0 \pm 2.8$	-	-	-
13	10.1 ± 0.0	14.4 ± 0.0	21.5 ± 0.1	32.3 ± 0.0	244.1 ± 1.8	788.3 ± 1.9	371.9 ± 2.2	446.5 ± 1.3	519.4 ± 2.5
16	9.0 ± 0.0	12.9 ± 0.0	19.0 ± 0.1	29.0 ± 0.0	244.5 ± 2.1	711.2 ± 2.3	364.3 ± 1.8	439.3 ± 1.2	509.6 ± 2.0
31	4.8 ± 0.0	6.6 ± 0.0	10.0 ± 0.0	14.9 ± 0.1	178.8 ± 1.3	415.4 ± 1.8	249.9 ± 2.2	307.1 ± 1.1	361.4 ± 1.2
32	4.8 ± 0.0	6.9 ± 0.0	10.1 ± 0.0	15.4 ± 0.0	193.8 ± 1.1	430.8 ± 1.2	277.8 ± 1.2	328.4 ± 1.5	381.4 ± 1.7

managed cyclic quorum implementation's speedup was up to 27 percent faster for this input dataset than without the management logic. However for Singer difference sets [6], which require fewer datasets (D_i) to guarantee the all-pairs property (Section 4), our cyclic quorum implementation without management was already efficient without any redundant work. Hence, when the management logic was added, the speedup decreased by up to 1 percent for this input dataset due to the overheads introduced.

It is worth noting that the managed "M-39,298" and "M-*45,265" curves are considerably more smooth with increasing nodes than their unmanaged curve partners. This is the impact of being able to apply the computation management logic to prevent redundant work. Every additional node is now contributing to complete useful work without any redundant work being performed. The unmanaged implementation is not as smooth because some choices of P nodes result in redundant work being performed, hence taking longer to complete decreasing the observed speedup.

Lastly, the speedup curves corresponding to the $N = 45,265$ dataset in Fig. 9, and the runtime data in Table 5, work toward illustrating our distributed cyclic quorums set solution enables the scaling to larger dataset sizes that previously may have been time and resource prohibitive. However unlike the $N = 39,298$ curves, the speedup is super-linear as seen by the speedup value for some inputs being more than 5x the number of nodes used. When renting the processing resources from the cloud or utilizing resources on a local HPC system, this means not only will our solution compute the all-pairs for larger input datasets results faster, it will also do so more cheaply by using as much as 5x fewer node compute hours to complete the computation.

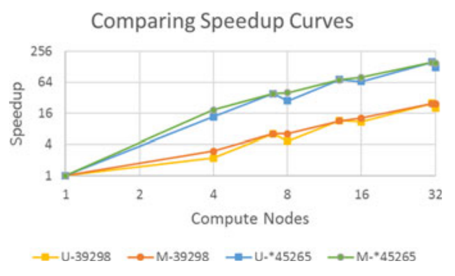


Fig. 9. Comparing the speedup of our unmanaged cyclic quorum algorithm with that of our algorithm with additional computation management logic (U versus M). Both were executed using (P) parallel nodes with speedups in references to an optimized single node algorithm. Two input datasets are compared, one with $N = 39,298$ rows which fit within available memory, and another with $N = 45,265$ rows which exceeded the single node memory limit, requiring the use of an alternate lower memory optimized algorithm. For non-Singer difference sets, the managed all-pairs computations were up to 30 percent faster than the unmanaged implementation. The Singer difference sets the overhead of the management was typically less than 1 percent, which put the unmanaged speedup slightly ahead.

7 Conclusions

In this paper, we proposed using cyclic quorums sets to scale all-pairs computation problems. We show that cyclic quorum sets have an all-pairs property that allows for data replication to be minimal and leads to simple computation management as well. The corresponding dataset quorums were $N/\sqrt{P}$ in size, up to 50 percent smaller than the dual $N/\sqrt{P}$ array implementations, and significantly smaller than solutions requiring all data. The cyclic quorums led to a simple algorithm design with all of the data needed for pairing existing in a node's dataset quorum.

Implementation evaluation took a single node bioinformatics all-pairs implementation and demonstrated scalability with our cyclic quorum set methods on real and synthetic datasets. Average runtimes showed the ability to scale linearly with additional nodes added. On one real dataset, greater than 150x (super-linear) runtime speedup and 80 percent reduction in memory usage per node was was observed using 31 nodes.

For non-Singer difference sets, we developed a decentralized, communication-less computation management technique to identify and avoid all redundant computations in the $\binom{k}{2}$ quorum dataset pairings. This showed a greater flexibility for users to efficiently utilize all their compute resources. For some P nodes, the additional computation management logic increased the average runtime speedup by as much as 30 percent, while the overhead of the management was typically less than 1 percent.

Future work includes investigating applications where dependability is important. Applying quorum redundancy to deliver memory and computationally efficient solutions to these applications could enable new growth and scalability.

Acknowledgments

Research funded in part by US National Science Foundation Graduate Research Fellowship Program, IBM Ph.D. Fellowship Program, Symbi GK-12 and Trinect Fellowships at Iowa State University, and the Philip and Virginia Sproul Professorship at Iowa State University. The research reported in this paper is partially supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by

NSF under MRI grant number CNS 1229081 and CRI grant number 1205413. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies. The authors would like to thank the anonymous reviewers for detailed reviews and excellent suggestions to improve the quality of this paper. The authors also would like to thank Venkata-Kasi Yeleswarapu for thoroughly reading the revised manuscript and making excellent suggestions to improve the quality of presentation.

REFERENCES

[1] C. Drew, "Military is awash in data from drones," Jan. 2010. [Online]. Available: http://www.nytimes.com/2010/01/11/business/11drone.html

[2] R. Hedegaard, "Handshake problem," Jan. 2016. [Online]. Available: http://mathworld.wolfram.com/HandshakeProblem.html

[3] C.-M. Chao and Y.-Z. Wang, "A multiple rendezvous multichannel MAC protocol for underwater sensor networks," in *Proc. IEEE Wireless Commun. Netw. Conf.*, 2010, pp. 1–6.

[4] W.-S. Luk and T.-T. Wong, "Two new quorum based algorithms for distributed mutual exclusion," in *Proc. 17th Int. Conf. Distrib. Comput. Syst.*, 1997, pp. 100–106.

[5] C. J. Kleinheksel and A. K. Somani, "Scaling distributed all-pairs algorithms," in *Proc. Int. Conf. Inf. Sci. Appl.*, 2016, pp. 247–257.

[6] C. J. Colbourn, *CRC Handbook of Combinatorial Designs*. Boca Raton, FL, USA: CRC Press, 2010.

[7] 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 Inf. Sci. Syst.*, vol. 1, no. 1, 2013, Art. no. 6.

[8] P. J. Phillips, et al., "Overview of the face recognition grand challenge," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2005, pp. 947–954.

[9] T. Chapman and A. Kalyanaraman, "An OpenMP algorithm and implementation for clustering biological graphs," in *Proc. 1st Workshop Irregular Appl. Archit. Algorithm*, 2011, pp. 3–10.

[10] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An abstraction for data-intensive computing on campus grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 33–46, Jan. 2010.

[11] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995.

[12] M. Driscoll, E. Georganas, P. Koanantakool, E. Solomonik, and K. Yelick, "A communication-optimal N-body algorithm for direct interactions," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 1075–1084.

[13] A. Reverter and E. K. Chan, "Combining partial correlation and an information theory approach to the reversed engineering of gene co-expression networks," *Bioinf.*, vol. 24, no. 21, pp. 2491–2497, 2008.

[14] L. Koesterke, et al., "Optimizing the PCIT algorithm on stampede's xeon and xeon phi processors for faster discovery of biological networks," in *Proc. Conf. Extreme Sci. Eng. Discovery Environ. Gateway Discovery*, 2013, pp. 14:1–14:8. [Online]. Available: http://doi.acm.org/10.1145/2484762.2484794

[15] N. S. Watson-Haigh, H. N. Kadarmideen, and A. Reverter, "PCIT: An R package for weighted gene co-expression networks based on partial correlation and information theory approaches," *Bioinf.*, vol. 26, no. 3, pp. 411–413, 2010.

[16] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *Proc. 4th Symp. Frontiers Massively Parallel Comput.*, 1992, pp. 120–127.

[17] V. Kumar and A. Agarwal, "Multi-dimensional grid quorum consensus for high capacity and availability in a replica control protocol," *High Performance Archit. Grid Comput.*, vol. 169, pp. 67–78, 2011.

[18] M. Maekawa, "An algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 2, pp. 145–159, 1985.

[19] H. Chae, S. Rhee, K. P. Nephew, and S. Kim, "BioVLAB-MMIA-NGS: MicroRNA–mRNA integrated analysis using high-throughput sequencing data," *Bioinf.*, vol. 31, no. 2, pp. 265–267, 2015.

[20] Amazon, "AWS high performance computing," Jan. 2016. [Online]. Available: https://aws.amazon.com/hpc/

[21] Amazon, "Amazon EC2 instance types," Jan. 2016. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

Cory James Kleinheksel received the BS degree in computer engineering from Iowa State University, in 2008 and the PhD degree in computer engineering from Iowa State University, in 2016. He has received fellowships from the US National Science Foundation Graduate Research Fellowship Program, IBM PhD Fellowship Program, and Symbi GK-12 and Trinect Fellowship Programs at Iowa State University. His research interests include the area of big data communication and processing with emphasis on parallel and distributed systems. He is a member of the IEEE.

Arun K. Somani received the MSEE and PhD degrees in electrical engineering from McGill University, Montreal, Canada, in 1983 and 1985, respectively. He is serving as Anson Marston distinguished professor and Philip and Virginia Sproul professor of electrical and computer engineering with Iowa State University. He also served as scientific officer for Govt. of India, New Delhi, from 1974 to 1982 and as a faculty member with the University of Washington, Seattle, Washington, from 1985 to 1997.. His research interests include the area of computer system design and architecture, fault tolerant computing, computer interconnection networks, WDM-based optical networking, and reconfigurable and parallel computer systems. He has also served as IEEE distinguished visitor, IEEE distinguished tutorial speaker, and IEEE Communication Society distinguished visitor and has delivered several key note speeches, tutorials and distinguished and invited talks all over the world. He is a fellow of the IEEE for his contributions to theory and applications of computer networks in 1999 and as a Distinguished Engineer of the ACM in 2006. He also has been elected a fellow of AAAS in 2012.