ABSTRACT

Title of Dissertation: 1D-CROSSPOINT ARRAY AND ITS CONSTRUCTION, APPLICATION TO BIG DATA PROBLEMS, AND HIGHER DIMENSION VARIANTS

Taeyoung An, Doctor of Philosophy, 2022

Dissertation Directed by: Professor A. Yavuz Oruç
Department of Electrical and Computer Engineering

Increased chip densities offer massive computation power to deal with fundamental big data operations such as sorting. At the same time the proliferation of processing elements (PEs) in settings such as High Performance Computers(HPCs) or servers together with the employment of more aggressive parallel algorithms cause the interprocessor communications to dominate the overall computation time, potentially resulting in reduced computational efficiency. To overcome this issue, this dissertation introduces a new architecture that uses simple crosspoint switches to pair PEs instead of a complex interconnection network. This new architecture may be viewed as a "quadratic" array of processors as it uses $O(n^2)$ PEs rather than $O(n)$ as in linear array processor models. In addition, three different models for sorting big data in a distributed computing environment such as Cloud computing are presented. With the most realistic model of the three, we demonstrate that the high parallelism made possible by the simple communication channels overcomes the seemingly excessive hardware complexity and performs comparable to or better than existing algorithms. Furthermore, two additional algorithms of matrix multiplica-

tion and triangle counting for the 1D-Crosspoint Array are introduced and analyzed. Lastly, two higher dimensional variants, 2D- and 3D-Crosspoint Array are also proposed with a construction method, which succeeds in reducing the number of PEs required by utilizing the communication channels in the added dimensions.

1D-CROSSPOINT ARRAY AND ITS CONSTRUCTION, APPLICATION TO BIG DATA PROBLEMS, AND HIGHER DIMENSION VARIANTS

by

Taeyoung An

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor A. Yavuz Oruç, Chair/Advisor
Professor Manoj Franklin
Professor Donald Yeung
Professor Shuvra S. Bhattacharyya
Professor Maria K. Cameron

Dedication

To my wife, Seongsil Lee,

and my parents, Chongkoo An and Meeai Yoon,

for their love and constant support

Acknowledgments

I wish to express my deepest gratitude to my advisor, Professor A. Yavuz Oruç for his supervision during the work of this dissertation, and for his guidance and encouragement throughout my years in graduate school. I would never have come this far without his help.

I would also like to thank the other members of the dissertation committee, Professor Manoj Franklin, Professor Donald Yeung, and Professor Shuvra S. Bhattacharyya of the Electrical and Computer Engineering Department and Professor Maria K. Cameron of the Mathematics Department for serving on my committee.

I am also thankful to my parents and parents-in-law for their immense support throughout the course of my PhD.

Finally, I would like to thank my wife, Seongsil. I simply could not have completed this journey without her beside me.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Digital data is being generated at an unprecedented rate. Back in 2003, it was reported that the yearly generated data stored in hard disks was 5.1 Exabytes(EB, 10^{18}B)[5]. In 2012, the same amount of data was generated less than a day as it was reported that 2.72 Zettabytes(ZB, 10^{21}B) was generated in that year[6]. Since then, the growth of digital data has been exponential, generating 16.1ZB, 33ZB, and 59ZB of data by the years 2016, 2018, and 2020 respectively[7, 8, 9]. Furthermore, the growth is not predicted to slow down anytime in the near future[8]. Consequently, an increasing number of organizations across the globe are utilizing a variety of big data analysis ideas and techniques. The big data analysis is driving many aspects of society, including mobile services, retail, manufacturing, financial services, life sciences, and physical sciences[10, 11, 12]. In each of these aspects, companies are collecting and analyzing vast sums of data to develop new competitive advantages[8].

Consequently, a lot of research has been done on how to deal with big data problems in computer science and related fields. It is obvious that big data is too large to process by a single processor in a reasonable time and/or is to be kept in a single memory. Parallel processing is a solution to cope with big data and at first, utilizing more processors would seem to be a good approach. However, increasing the number of processors alone may not necessarily translate into increased performance proportionally. The main reason for this limitation is the intra- and inter-

processor communication, which becomes more difficult to deal with as the number of processing elements (PEs) increases. Generally speaking, communication time entails two components: (a) transmission time and (b) contention time. The transmission time refers to the time a signal takes to travel the communication medium such as buses or crosspoints. The contention time refers to the time that is caused by contention for limited communication paths and resources. Both types of communication time can be problematic as the number of PEs increases. In the case of transmission time, using more PEs naturally increases the length of communication paths, resulting in longer transmission times. Using more PEs also generates more contention in the communication medium. In fact, the communication time in a parallel processor is reported to be more dominant than the processing time and can even become a bottleneck for overall computation time[13, 14][15, p.33]. Consequently, the focus of parallel processing research gradually shifted to reducing communication time complexity from the earlier focus on improvement and optimization of each processor.

This dissertation explores some new ideas to mitigate such communication time complexity issues that arise in parallel executions of a number of big data algorithms. To this end, a new network, called 1D-Crosspoint Array is introduced to interconnect a set of PEs with the minimization of the contention time in mind. It is a tightly coupled parallel processing model in which every pair of PEs is connected by a direct link (an on-and-off switch). For an input size of n, this parallel processing model uses $O(n^2)$ PEs whereas a conventional linear array would use $O(n)$ PEs [16, 17, 18, 19, 20]. As it will be shown in the dissertation, one particular benefit of using $O(n^2)$ PEs is $O(1)$ time complexity in sorting a set of n keys, $O(n)$ time complexity when multiplying two $n \times n$ matrices, and $O(|E|^{3/2}/n)$ for counting triangles in a graph $G(E, V)$. In all three cases, 1D-Crosspoint Array model provides an optimal parallel computation in terms

of time complexity. A 2D and 3D variations of 1D-Crosspoint Array are also presented in the dissertation in order to reduce the number of PEs by a constant factor.

The rest of the dissertation is organized as follows. This chapter is concluded with a survey of existing results on reducing the communication cost. The 1D-Crosspoint Array is presented in detail together with construction method in Chapter 2. Next, we show how the 1D-Crosspoint Array can solve three big data related problems, namely sorting, matrix multiplication, and triangle counting in Chapters 3, 4, and 5. In Chapter 6, the higher dimension variants of the 1D-Crosspoint Array are introduced and analyzed and the dissertation is concluded in Chapter 7.

1.1 A Survey of Prior Research

Many of the earlier works that aimed to reduce communication time complexity in multiprocessors were based on hardware solutions. For example, the hypercube architecture[21, 22, 23, 24] attracted many researchers as it offers a relatively low communication time complexity with rich connectivity as compared to mesh architectures. A hypercube is a generalization of the three dimensional(3D) cubic graph to arbitrary number of dimensions such that each PE of an n-dimensional cube has n direct neighbor PEs. Compared to a PE of a cubic architecture, which has 3 neighboring PEs, such a high number of direct links between PEs generate less contention, and $O(n)$ distance between any two PEs. At the same time, n links per PE make its VLSI layout more complex than some other networks such as cube-connected cycles[25], hypercycles[26], or hypernet[27].

A 3D layouts of processors were utilized to compute specific algorithms as well. For example, several parallel matrix multiplication algorithms were proposed to run on processors that

were laid out in 3D[28, 29, 30]. The advantage of the 3D layout compared to the 2D layout is the additional communication channels among PEs, which effectively reduce the communication distance.

When optical techniques were introduced in the field of digital signal processing in 1960s, optical interconnects were studied as a faster substitute for electronic interconnects[31, 32, 33]. Optical signal processing provided many advantages over electrical signal processing in terms of speed, energy, and density[34, 35, 36, 37]. The extremely high bandwidth of optical fiber offers faster communication and lower power consumption, and so high performance computing (HPC) systems employ optical interconnects[36, 37]. However, optical interconnects present some issues that must be dealt with such as interfacing electronic and photonic subsystems, increasing reliability of optical communications and higher upfront costs[36]. Still, optical interconnects are used in on-chip networking[38] as well as networking between computer chips[20, 39, 40, 41].

At higher levels of integration and after big data problems emerged, access to supercomputers and HPCs is now more common using cloud computing[42, 43, 44] and grid computing[45, 46, 47]. However, such systems are built as a general-purpose parallel system, meaning the system is designed with versatility in mind rather than the communication time. To be sure, many communication time reduction techniques are implemented in the HPCs, but minimizing the communication time for each different algorithm still depends on the algorithm itself. For example, the Fast Fourier transform(FFT)[48] is an algorithm that can benefit significantly from reducing communication time. This is because it requires triple all-to-all data exchange, also known as global transpose, which implies intensive inter-processor communications. In fact, the communication of a naive 1D FFT algorithm is reported to account for anywhere from 50% to over 90% of the overall computation time[49]. In [50], a FFT algorithm with no all-to-all data

exchange was proposed. However, the computation time complexity of the algorithm was raised to $O(n^{1.5})$ from $O(n \lg n)$ for n-size input. Due to the increased complexity, the experimental result showed that with 8 PEs, the no-communication algorithm performed faster than the regular FFT when the data size was small, but with bigger data size, the no-communication FFT was slower since the larger workload of each PE increased the computation complexity [50]. A few years later, Tang et al. proposed a more feasible FFT algorithm with single all-to-all data exchange[49, 51]. While reducing the number of all-to-all data exchanges, the authors were able to retain the computation time complexity of $O(n \lg n)$. The algorithm involves some additional inter-processor communication in addition to the all-to-all data exchange, but is reported to be negligible, which was proven to be the case by the experimental result that showed the algorithm being more than two times faster than the regular FFT algorithm.

As access to HPCs became more common, big data computation frameworks, such as MapReduce[52] and Hadoop[53], were accepted by many companies and research groups. The difficulty of programming algorithms to fully exploit the potential parallelism of HPCs was one of the reasons why frameworks gained popularity. In particular, MapReduce framework is composed of *Map* function and *Reduce* function with a *shuffle* phase in between. The input data is key/value pairs. Map and Reduce functions are written by the user, and each of Map function generates an intermediate key/value pairs for each input. Then each intermediate data is mapped to Reduce function associated with specific intermediate key(s). The Reduce function than carries out the user-written computation and returns an output for each intermediate key. Such framework made programming parallel algorithms simple and scalable[54, 55]. However, the framework generated substantial intermediate data which was communicated between processors in the shuffle phase, degrading the performance due to increased communication time. To

address this issue, a communication reduction scheme for MapReduce framework, called Coded MapReduce[56, 57, 58] was proposed. The Coded MapReduce scheme does repetitive distribution, or in other words, it distributes same input data to more than one Map function. With some of the Map functions sharing data, the intermediate data can be 'coded' into smaller data using each shared data as a key, and this effectively reduces the communication time. It must be noted that the initial repetitive distribution exacts additional communication time, but is offset by the fact that the distributed data can be used in the Reduce function locally, assuming the same set of processors that ran Map functions are reused for the Reduce functions.

The MapReduce framework provides a very high level approach to structuring algorithms without focusing on the lower level architectural considerations in the implementation of parallel algorithms. This will likely result in executions of such algorithms with excessive communication time as the framework overlooks the potential trade offs between the available hardware and communication time complexity of the algorithms in question. Such tradeoffs will be explored and illustrated in this dissertation using a big data model for parallel sorting.

1.2 Contributions

The contributions of this dissertation can be summarized as follows.

i. We introduce a new interconnect architecture, called a 1D-Crosspoint Array to minimize communication contentions between PEs by localizing their links.

ii. We propose three parallel processing models to describe realistic parallel processing in big data computations and develop distributed algorithms over cloud computing environments.

iii. We present an $O(1)$ time parallel sorting algorithm using the 1D-Crosspoint Array.

iv. We present an $O(n)$ time parallel matrix multiplication algorithm.

v. We present an $O(n(|V| + |E|) + n(|V_{prx}| + |E_{prx}|) + n|t|)$ time parallel triangle counting algorithm.

vi. We present higher dimensional variants of the 1D-Crosspoint Array to reduce the number of PEs by increasing their connectivities.

Chapter 2: The 1D-Crosspoint Array

This chapter describes the definition, properties and construction of the 1D-Crosspoint Array.

2.1 Overview

The 1D-Crosspoint Array is a tightly coupled linear array of $O(n^2)$ PEs that uses simple crosspoint switches to connect each pair of PEs. Each PE communicates only with those PEs that are directly connected to it, i.e., its left and right neighbor PEs. The locality of connections between the PEs may potentially limit the communication ability of PEs that are not directly connected to each other, but this is avoided by introducing the notion of *classes* of *replicates*. That is, the $O(n^2)$ PEs are divided into n classes, and the PEs belonging to the same class are referred to as replicates of one another in the same class. Replicates are created to distribute the workload to multiple PEs and attain parallelism with local communication links. Those replicates in any given class will be provided the same input dataset and will be carrying out the same computation with their own neighboring replicates. Thus, replicates are functionally similar to other replicates in the same class and when there is no ambiguity, they will be viewed as one and the same. Now, to enforce full connectivity across classes of replicates, the 1D-Crosspoint Array is constructed in such a way that a class will have at least one replicate from every other class next to one of its own replicates. Therefore, the 1D-Crosspoint Array has the same global connectivity

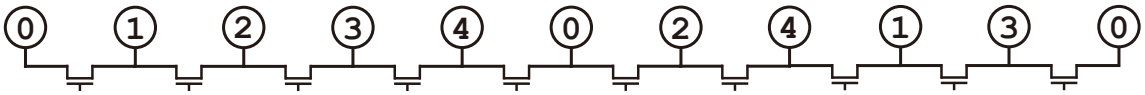


Figure 2.1: 1D-Crosspoint Array for $n = 5$

between classes as a conventional linear array model would have between n PEs. As a result, the 1D-Crosspoint Array can be seen as an $n-$PE array with a global communication connection but without any communication contention between the PEs. For example, a 1D-Crosspoint Array for 5 classes is depicted in Figure 2.1. The figure shows that the replicates, which are represented by the circles, are paired with neighboring replicates by crosspoint switches. The numbers inside the circles denote the classes to which replicates belong. For example, the two replicates belonging to class 1 are next to replicates belonging to classes 0 and 2 as well as classes 4 and 3. Hence, class 1 has replicates of every other class as a direct neighbor, and would not require any more than the minimum communication time (one step) to communicate with any of the other classes of replicates. This is true for all classes in a 1D-Crosspoint Array.

Throughout this dissertation, we say that a crosspoint array has a 1D-layout if (a) each replicate is paired with at most two replicates and (b) replicates are placed on a straight path without any wraparounds. As seen in the figure, all $\binom{5}{2} = 10$ pairings of five replicates are realizable in this layout of 10-crosspoint array in which each PE is replicated no more than three times[1]. Our goal here is to have a pair of replicates one from each of the $\binom{n}{2}$ pairs of n classes share a crosspoint in a 1D-Crosspoint Array, using a minimum number of crosspoints and replicates with the assumption that no replicate is adjacent to more than two replicates. We formalize this goal as a notion of *complete adjacency* due to its frequent usage throughout the dissertation.

[1]If wraparound were allowed then two replications would suffice when $n = 5$. In general, for any odd integer n, $\lfloor n/2 \rfloor$ replications would suffice with wraparounds.

Definition 1. ***Complete adjacency*.** A 1D-Crosspoint Array with n classes in which every one of $\binom{n}{2}$ pairs of n classes is adjacent is said to have *complete adjacency*.

It will be assumed that all 1D-Crosspoint Arrays have the complete adjacency property throughout this dissertation. To construct a 1D-Crosspoint array with complete adjacency using a minimum number of replicates, we must first identify the minimum number of replicates needed for complete adjacency. Propositions 1 through 4 below are used to prove Theorem 1, which establishes the minimum number of replicates needed to construct a 1D-Crosspoint Array with complete adjacency.

Proposition 1. A 1D-Crosspoint Array with n classes and complete adjacency requires $\binom{n}{2} + 1$ replicates in all of its n classes and $\binom{n}{2}$ crosspoints.

Proof. The complete adjacency dictates that each class be adjacent to every other class. Given that there are n classes and each adjacency pairs at most two classes, $n(n-1)/2 = \binom{n}{2}$ adjacencies or crosspoints are required. That $\binom{n}{2}$ crosspoints (edges) require $\binom{n}{2} + 1$ replicates (vertices) follows from Euler's formula $f + v - e = 2$, with $f = 1, e = \binom{n}{2}$ where f, v, and e denote the number of faces, number of vertices, and number of edges in any graph, respectively. □

Proposition 2. In a 1D-Crosspoint Array with n classes and complete adjacency, each class must have no fewer than $\left\lceil \frac{n-1}{2} \right\rceil$ replicates.

Proof. A replicate in a 1D-Crosspoint Array can be adjacent to at most two replicates. Thus, any given class needs at least $\frac{n-1}{2}$ replicates to be adjacent to the replicates of all other $n-1$ classes. Since number of replicates can only be an integer, $\frac{n-1}{2}$ rounds up to the next larger integer, i.e., $\left\lceil \frac{n-1}{2} \right\rceil$. □

The last proposition can be strengthened for the two replicates at the end of the 1D-Crosspoint Array:

Proposition 3. If the two replicates at the two ends of a 1D-Crosspoint Array with n classes and complete adjacency belong to the same class then that class must contain at least $\lceil \frac{n+1}{2} \rceil$ replicates in order to be adjacent to the $n-1$ replicates in the other $n-1$ different classes.

Proof. Let's suppose that the two replicates at the two ends belong to class α. These two replicates can only be adjacent to two replicates in total. Therefore, for class α to be adjacent to $n-1$ other classes, it must be adjacent to some $(n-1)-2 = n-3$ classes excluding the classes adjacent to the two replicates at the two ends. By Proposition 2, we can deduce that class α needs to have at least $\frac{n-3}{2}$ replicates in addition to the two replicates at the ends. Hence, class α requires $\frac{n-3}{2} + 2 = \frac{n+1}{2}$ replicates in total. Again, since the number of replicates must be an integer, it rounds up to $\lceil \frac{n+1}{2} \rceil$. □

Proposition 4. If the two replicates at the two ends of a 1D-Crosspoint Array with n classes and complete adjacency belong to different classes then each of those two classes must have at least $\lceil \frac{n}{2} \rceil$ replicates in order to be adjacent to the $n-1$ replicates in the other $n-1$ classes.

Proof. This time, suppose that the replicates at the two ends of the array belong to class α and β. The class α can be adjacent to only one class via the replicate at its end. Therefore, for class α to be adjacent to $n-1$ classes, it must be adjacent to $n-2$ classes via its remaining replicates. The same holds for class β. This implies that class α and β must each have at least $\frac{n-2}{2} = \frac{n}{2} - 1$ replicates. Hence, including the replicate at its end, class α and β must contain at least $\lceil \frac{n}{2} - 1 + 1 \rceil = \lceil \frac{n}{2} \rceil$ replicates respectively. □

Combining Propositions 1 through 4, we have:

Theorem 1. A 1D-Crosspoint Array with n classes and complete adjacency requires at least $\frac{n(n-1)}{2} + 1$ replicates for odd n, and $\frac{n^2}{2}$ replicates for even n.

Proof. When the replicates at the two ends of the array belong to the same class, by Proposition 3, that class must have $\lceil \frac{n+1}{2} \rceil$ replicates. By Proposition 2, each of the other $n-1$ classes must then have $\lceil \frac{n-1}{2} \rceil$ replicates. Therefore, a 1D-Crosspoint Array with n classes and complete adjacency, it would require at least

$$\left\lceil \frac{n+1}{2} \right\rceil \times 1 + \left\lceil \frac{n-1}{2} \right\rceil \times (n-1) \text{ replicates.} \tag{2.1}$$

Eqn. 2.1 equals $\frac{n(n-1)}{2} + 1$ for odd n, and $\frac{n^2}{2} + 1$ for even n.

When the replicates at the two ends of the array belong to different classes, by Proposition 4, those classes would need to contain $\lceil \frac{n}{2} \rceil$ replicates each. By Proposition 2, each of the other $n-2$ classes would need to have $\lceil \frac{n-1}{2} \rceil$ replicates. Therefore, for all $\binom{n}{2}$ pairs of n different classes of replicates to be adjacent, the 1D-Crosspoint Array would require at least

$$\left\lceil \frac{n}{2} \right\rceil \times 2 + \left\lceil \frac{n-1}{2} \right\rceil \times (n-2) \text{ replicates.} \tag{2.2}$$

Eqn. 2.2 equals $\frac{n(n-1)}{2} + 2$ for odd n, and $\frac{n^2}{2}$ for even n.

Therefore, a 1D-Crosspoint Array with n classes and complete adjacency requires $\frac{n(n-1)}{2} + 1$ or more replicates for odd n, and $\frac{n^2}{2}$ or more replicates for even n. □

With the minimum number of replicates required in a 1D-Crosspoint Array with n classes and complete adjacency determined, we proceed with the construction of such a 1D-Crosspoint Array in the next section.

2.2 Construction of 1D-Crosspoint Array

We borrow ideas from cyclic permutation groups and how they are used for constructing a one-sided binary tree-crossbar switch in [59, 60]. Let $p = (0\ 1\ 2\ 3\ \cdots\ n-1)$ be a permutation of n classes[2]. We will use p as the representation of layout of replicates[3] belonging to n classes, where classes whose ids are adjacent in the cycle representation of powers of p will also be adjacent in the 1D-Crosspoint Array. For example, p indicates a layout where a replicate of class 1 is on the right-hand side of a replicate of class 0, and a replicate of class 2 is on the right-hand side of a replicate of class 1, and so on. The cyclic group of permutations generated by p consists of n permutations $p, p^2, p^3, \cdots, p^n$, where p^n is the identity permutation. The permutation p^j, where $1 \leq j \leq n-1$, specifies that the right neighbor of class i is given by $(i+j) \bmod n$. It was shown in [59] that $p, p^2, \cdots, p^{n-1}$ map every element to a distinct element, which can be interpreted as every replicate having distinct right neighbor in the corresponding 1D-Crosspoint Array. Moreover, it was also shown that p^{n-j}, $1 \leq j \leq \frac{n}{2} - 1$, is p^j written in reverse and it represents the inverse permutation of p^j. To make the discussion in this dissertation more self-contained, we restate these results in Proposition 5 and 6.

Proposition 5. [59, 60] For any i, j_1, j_2 such that $0 \leq i, j_1, j_2 \leq n-1$, $(i+j_1) \bmod n = (i+j_2) \bmod n$ if and only if $j_1 = j_2$.

Proof. $(i+j_1) \bmod n = i + j_1 - k_1 n$, where $k_1 = 0$ or 1, since $0 \leq i + j_1 \leq 2n-2$. Likewise, $(i+j_2) \bmod n = i + j_2 - k_2 n$, where $k_2 = 0$ or 1. When $k_1 = k_2 = 0$, $i + j_1 = i + j_2$ holds if and only if $j_1 = j_2$. Similarly when $k_1 = k_2 = 1$, $i + j_1 - n = i + j_2 - n$ holds if and only

[2] Throughout the rest of the dissertation, p will be fixed to this permutation.

[3] Replicates in our presentation correspond to PEs in [59, 60], which will also be used at places in our presentation to emphasize their processing element functionality.

if $j_1 = j_2$. On the other hand, if $k_1 = 0$ and $k_2 = 1$, then $i + j_1 = i + j_2 - n$ or $j_2 - j_1 = n$ which contradicts the assumption $0 \leq j_1, j_2 \leq n-1$. The $k_1 = 1$ and $k_2 = 0$ case cannot be true for the same reason. Therefore, we conclude that $(i + j_1) \bmod n = (i + j_2) \bmod n$ if and only if $j_1 = j_2$ where $0 \leq i, j_1, j_2 \leq n-1$. □

Definition 2. The k-th image of element $i, 0 \leq i \leq n-1$ in p^j is $p^{kj}(i) = (i + kj) \bmod n$.

Proposition 6. [59, 60] The k-th image of i in p^{n-j}, $(i+k(n-j)) \bmod n$ is the same as $(n-k)$-th image of i in p^j, $(i + (n-k)j) \bmod n$.

Proof. $(i + k(n-j)) \bmod n = ((i - kj) \bmod n + kn \bmod n) \bmod n = (i - kj) \bmod n = (i - kj + jn) \bmod n = (i + (n-k)j) \bmod n$. □

The replicates are identified with the elements in p^j that are generally expressed as a set of disjoint cycles. For example, p consists of one long cycle, i.e., a cycle of n elements, p^2 may or may not be a long cycle depending on n being a prime or not. The elements that are adjacent in the cycles of each p^j determine the crosspoints between the replicates in some unique way. More specifically, two replicates will have a crosspoint between them if the elements that identify their classes are adjacent in a cycle. In [59], replicates (PEs) need not be physically adjacent when crosspoints are placed in-between them. However, in our construction of a 1D-Crosspoint Array with complete adjacency, we restrict the placement of crosspoints between replicates that are physically adjacent. In addition, each replicate is connected to exactly one other replicate in [59], whereas, in our work, each replicate is connected up to two other replicates. The last distinction we need to draw between [59] and our work is that we only use $p, p^2, \cdots, p^{\frac{n}{2}}$ in the construction of the 1D-Crosspoint Array even though we will make use of $p^{\frac{n}{2}+1}, p^{\frac{n}{2}+2}, \cdots, p^{n-1}$ in some of

our proofs. With these ideas in mind, we now describe some preliminary facts that are used in the construction.

Proposition 7. The number of cycles in permutation p^j is equal to $\gcd(n, j)$, i.e., greatest common divisor of n and j.

Proof. Suppose a cycle starts with i. Then the successive images of i in that cycle are obtained by adding $j, 2j, ..,$ to i modulo n. The cycle ends when the result of the modulo addition equals i again, which is when $(i+mj) \bmod n = i$. This implies that the mj is the least common multiple of n and j, i.e., $mj = \text{lcm}(n, j)$. Then, using the equation $\text{lcm}(n, j) \times \gcd(n, j) = n \times j$, we have $mj = \text{lcm}(n, j) = \frac{nj}{\gcd(n,j)}$, and therefore the number of cycles n/m in p^j is given by $n/m = \gcd(n, j)$.

Proposition 8. The elements of a cycle are at least $\gcd(n, j)$ apart from each other.

Proof. Let $w = \gcd(n, j)$ and let u be an element in a cycle in permutation p^j, $0 \leq j \leq \frac{n}{2}$. Further suppose that v is another element that belongs to the same cycle to which u belongs. By the definition of permutation p^j, we can express v as $v = (u+xj) \bmod n, 0 \leq x \leq \frac{n}{w} - 1$, which implies $u + xj = an + v$, where a is a non-negative integer. Since j and n are a multiple of w, letting $j = bw$ and $n = cw$, where b and c are positive integers, we have $u + x(bw) = a(cw) + v$ or $v = u + (bx - ac)w$. Now, since $v \neq u$, and b, x, a, and c are all integers, $(bx - ac)$ is a non-zero integer. Hence, v can only be a multiple of w apart from u, such as $u \pm w, u \pm 2w$, etc. Therefore, every element of a cycle in p^j is at least $\gcd(n, j)$ apart from each other. □

Proposition 9. The smallest element of any cycle in any of p^j, $1 \leq j \leq \frac{n}{2}$, is less than or equal to $\frac{n}{2} - 1$ when n is even and $\frac{n-1}{2} - 1$ when n is odd.

Proof. By Proposition 7, permutation p^j, $1 \leq j \leq \frac{n}{2}$, has $\gcd(n, j)$ cycles. Suppose p^j has a cycle g_i, where $0 \leq i \leq \gcd(n, j) - 1$. Further suppose an element m that is less than or equal to $\gcd(n, j) - 1$ is in g_i. Then, by Proposition 8, m would be the smallest element in that cycle, since $m - \gcd(n, j) < 0$. Furthermore, no other element in g_i will be less than $m + \gcd(n, j)$. Since this holds for any m that is less than $\gcd(n, j)$, all the elements less than $\gcd(n, j)$, must be distributed to distinct cycles and be the smallest elements in their respective cycles. Now, when n is even and $1 \leq j \leq \frac{n}{2}$, the maximum value of $\gcd(n, j)$ is $\frac{n}{2}$. When n is odd and $1 \leq j \leq \frac{n}{2}$, the maximum value of $\gcd(n, j)$ is $\frac{n-1}{2}$. Therefore, the smallest element in any cycle must be less than or equal to $\frac{n}{2} - 1$ when n is even and $\frac{n-1}{2} - 1$ when n is odd. □

2.2.1 Construction of 1D-Crosspoint Array for Even n

The construction of the 1D-Crosspoint Array is given in Algorithm 1 for even n, and Figs. 2.2 and 2.3 illustrate how this algorithm works for $n = 12$.

Suppose that the cycles in $p, p^2, \cdots, p^{\frac{n}{2}}$ are written so that the first element is the smallest in every one of them[4]. Let Q_i denote the set of cycles in which the first element is i. Thus, Q_0 consists of all cycles that begin with 0, Q_1 consists of all cycles that begin with 1, and so on, and $Q_{\frac{n}{2}-1}$ consists of all cycles that begin with $\frac{n}{2} - 1$. The following propositions are used in Steps 3 and 5.

Proposition 10. The cycles in all of $p, p^2, \cdots, p^{\frac{n}{2}}$ can be partitioned into $n/2$ subsets Q_i, $0 \leq i \leq \frac{n}{2} - 1$.

Proof. By Proposition 9, for even n, the first element of any cycle in any of p^j, $1 \leq j \leq \frac{n}{2}$, is

[4]This is done for notational convenience only.

less than or equal to $\frac{n}{2} - 1$. □

Algorithm 1: Construction algorithm for even n.

Step 1. Suppose that an empty frame of a 1D-Crosspoint Array for $\frac{n^2}{2}$ replicates and a crosspoint between each pair of replicates is provided without the actual assignment of the replicates into the replicate spaces in the empty frame.

Step 2. Set $i = 0$.

Step 3. Choose any cycle in Q_i that consists of more than two elements. Then place a replicate that belongs to class i into the left most replicate space available in the 1D-Crosspoint Array. Next, suppose the next element in that cycle is α_i. Place a replicate that belongs to class α_i on the right-hand side of the previously placed replicate. Repeat placing replicates in the same manner from left to right into the 1D-Crosspoint Array until all the elements in that cycle are used. We will refer to this process as placing a cycle on the 1D-Crosspoint Array. (See Fig. 2.3(b).)

Step 4. Choose a cycle in Q_i that has not yet been chosen. Then place the newly chosen cycle into the 1D-Crosspoint Array in the same way as in Step 3. The first replicate associated with the new cycle should be placed on the right-hand side of the last replicate from Step 3 without skipping any replicate spaces.

Step 5. Repeat Step 4 until all the cycles consisting of more than two elements have been processed. Then choose the cycle in Q_i that consists of only two elements (Every Q_i has exactly one cycle of two elements as would be implied by Proposition 11 and the definition of Q_i), and place it into the 1D-Crosspoint Array as before.

Step 6. Set $i = i + 1$ and repeat Steps 3, 4, and 5 until $i = \frac{n}{2} - 1$. Again, replicate spaces must not be skipped between the steps.

Proposition 11. For even n, $p^{\frac{n}{2}}$ is the only permutation among p^j, $1 \leq j \leq \frac{n}{2}$, whose $\frac{n}{2}$ cycles are composed of two elements each.

Proof. The number of cycles in p^j is $\gcd(n, j)$ by Proposition 7. For even n and $j < n$, the maximum of $\gcd(n, j)$ is $\frac{n}{2}$, which occurs only if $j = \frac{n}{2}$. □

We next establish that Steps 3 through 6 construct a 1D-Crosspoint Array in which all $\binom{n}{2}$ pairs of classes of replicates are connected together by crosspoints. Oruç stated in [59] that

$$
\begin{aligned}
p &= (0,1,2,3,4,5,6,7,8,9,10,11)\\
p^2 &= (0,2,4,6,8,10)\ (1,3,5,7,9,11)\\
p^3 &= (0,3,6,9)\ (1,4,7,10)\ (2,5,8,11)\\
p^4 &= (0,4,8)\ (1,5,9)\ (2,6,10)\ (3,7,11)\\
p^5 &= (0,5,10,3,8,1,6,11,4,9,2,7)\\
p^6 &= (0,6)\ (1,7)\ (2,8)\ (3,9)\ (4,10)\ (5,11)
\end{aligned}
$$

(a) Step 1: The cyclic permutations p^j, $1 \leq j \leq n/2$. The smallest element of each cycle is set as the first element.

$$
\begin{aligned}
Q_0 &= \big\{(0,1,2,3,4,5,6,7,8,9,10,11),(0,2,4,6,8,10),\\
&\qquad (0,3,6,9),(0,4,8),(0,5,10,3,8,1,6,11,4,9,2,7),(0,6)\big\}\\
Q_1 &= \big\{(1,3,5,7,9,11),(1,4,7,10),(1,5,9),(1,7)\big\}\\
Q_2 &= \big\{(2,5,8,11),(2,6,10),(2,8)\big\}\\
Q_3 &= \big\{(3,7,11),(3,9)\big\}\\
Q_4 &= \big\{(4,10)\big\}\\
Q_5 &= \big\{(5,11)\big\}
\end{aligned}
$$

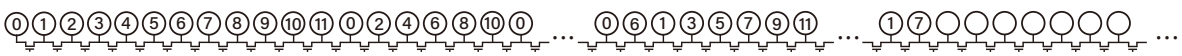
(b) Step 2: All of the cycles in all of p^j, where $0 \leq j \leq 6$, has been partitioned into Q_i, $0 \leq i \leq \frac{n}{2} - 1$.

Figure 2.2: Illustration of Step 1 and 2 for the example case of $n = 12$.

(a) Step 3: The 1D-Crosspoint Array after placing the first cycle of Q_0, assuming $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$ was chosen.

(b) Step 4: The 1D-Crosspoint Array after placing the second cycle of Q_0, assuming $(0, 2, 4, 8, 10)$ was chosen.

(c) Step 5: The 1D-Crosspoint Array after placing all the cycles of Q_0, with the cycle $(0, 6)$ from $p^{\frac{n}{2}}$ is placed last.



(d) Step 6: The 1D-Crosspoint Array after placing all the cycles of Q_1, with the cycle $(1, 7)$ from $p^{\frac{n}{2}}$ placed last. Note that some of the replicates are abbreviated by '$\cdots$'.

(e) The 1D-Crosspoint Array after placing all the cycles of every Q_i.

Figure 2.3: Illustration of Step 3 through 6 for the example case of $n = 12$.

by Proposition 5, $n-1$ permutations p^j, $1 \leq j \leq n-1$ express all $\binom{n}{2}$ pairs, since each of the n elements appears once in each of the $n-1$ permutations and is mapped to a distinct element each time. In the 1D-Crosspoint Array, each mapping corresponds to a crosspoint that is bidirectional and hence only half of the permutations is sufficient to represent all $\binom{n}{2}$ pairs by Proposition 6. More precisely, suppose that $p^j = g_1 g_2 ... g_k$, where each g_i represents a cycle and the cycles do not share any element. Further suppose that $g_i = (b_{i,0} b_{i,1} ... b_{i,r_i-1})$, where r_i is the length of g_i. Then $p^{n-j} = g_1^{-1} g_2^{-1} ... g_k^{-1}$, and we write g_i^{-1} as $g_i^{-1} = (b_{i,0} b_{i,r_i-1} b_{i,r_i-2} ... b_{i,1})$ by Proposition 6. Now, g_i represents the pairs $(b_{i,0}, b_{i,1}), (b_{i,1}, b_{i,2}), (b_{i,2}, b_{i,3}), (b_{i,3}, b_{i,4}), \cdots,$ $(b_{i,r_i-3}, b_{i,r_i-2}), (b_{i,r_i-2}, b_{i,r_i-1}), (b_{i,r_i-1}, b_{i,0})$ in the 1D-Crosspoint Array. These pairs also account for the pairs in g_i^{-1} as can be verified by a direct inspection of the terms in the cycle representation of g_i^{-1}. Therefore, it suffices to use g_i's in the contruction of the 1D-Crosspoint Array.

We note that, when *placing* the cycles on the 1D-Crosspoint Array, the last pair $(b_{i,r_i-1}\, b_{i,0})$, which is a mapping from the last element to the first element of a cycle is accounted for in Step 4. This is done by inserting a crosspoint between the last replicate placed by the previously chosen cycle and very first replicate placed by the newly chosen cycle, since every cycle in the same Q_i, $0 \leq i \leq \frac{n}{2} - 1$, start with the same $b_{i,0}$ by Steps 1 and 2. Also, Step 4 excludes choosing a cycle that consists of only two elements, and saves it for Step 5. This is because for cycles consisting of only two elements, i.e., when $r_i = 2$, $(b_{i,r_i-1}\, b_{i,0})$ coincide with $(b_{i,1}\, b_{i,0})$. Therefore, the crosspoint is not added in Step 4 in this case. It should be added that deferring the placement of cycles of length 2 to the end ensures threading cycles of larger length first to include all the wraparound pairs until a cycle of length 2 is reached. This can be seen in Fig. 2.2, where two cycles are added at the end in placements of $Q_i, 0 \leq i \leq 5$. It should be evident by these explanations that Steps 3 through 6, all $\binom{n}{2}$ pairs that are extracted from the first $\frac{n}{2}$ permutations, $p, p^2, \cdots, p^{\frac{n}{2}}$, are placed in the 1D-Crosspoint Array.

Remark 1a. The number of replicates that are derived from $p, p^2, \cdots, p^{\frac{n}{2}}$ is $n \times \frac{n}{2} = \frac{n^2}{2}$, which matches the lower bound in Theorem 1. Therefore, the 1D-Crosspoint Array described in the algorithm is optimal with respect to the number of replicates used. At the same time, it should be stated that this algorithm does not minimize the number of adjacencies between replicates, i.e., some pairs may appear more than once in the construction. For example, replicates 1 and 6 are adjacent in Fig. 2.3 (c) and (d) in two places. In fact, the pairs made of the last replicate placed in Step 5 and the first replicate placed in the next iteration of Step 3, i.e., the pairs made by elements from two different Q_i's, are already created elsewhere in the 1D-Crosspoint Array. In general, it can be shown that, for all even n, some $\frac{n}{2} - 1$ adjacencies may appear twice in the construction

of the 1D-Crosspoint Array in Algorithm 1. □

2.2.2 Construction of 1D-Crosspoint Array for Odd n

The construction of 1D-Crosspoint Array for odd n is similar to that of the even n case. In fact, we first construct a 1D-Crosspoint Array for $n-1$ classes, using Algorithm 1, except that in Step 1, we start with an empty frame with $\frac{n(n-1)}{2}+1$ replicates, and in Step 6, we skip a replicate space between the placements of consecutive Q_i's. An illustration of the construction in the case of $n = 7$ is shown in Fig. 2.4. When a 1D-Crosspoint Array for $n-1$ classes with complete adjacency is constructed with the empty spaces as shown in Fig. 2.4(c), we place the replicates of the n^{th} class, or class $n-1$, into those spaces after Step 6, as shown in Fig. 2.4(d). There will always be two empty replicate spaces at the end (for any odd n)[5], and we place the replicate of class $n-1$ in the first empty replicate space, and place the replicate of class 0 in the second one as shown in Fig. 2.4(e).

It still remains to establish that all $\binom{n}{2}$ adjacencies of n classes are included in the 1D-Crosspoint Array. By the construction of a 1D-Crosspoint Array for $n-1$ classes with complete adjacency, we form all the adjacencies between $n-1$ classes, and we would only need to add adjacencies between class $n-1$ and the other classes. Previously, in Remark 1a, we pointed out the existence of redundant pairs in the 1D-Crosspoint Array, which were made by elements from two different Q_i's. By inserting class $n-1$ in between the classes that form such pairs, we capture the adjacencies between class $n-1$ and other classes without destroying any of the existing pairs. In fact, this procedure will ensure the 1D-Crosspoint Array to have the adjacencies between class

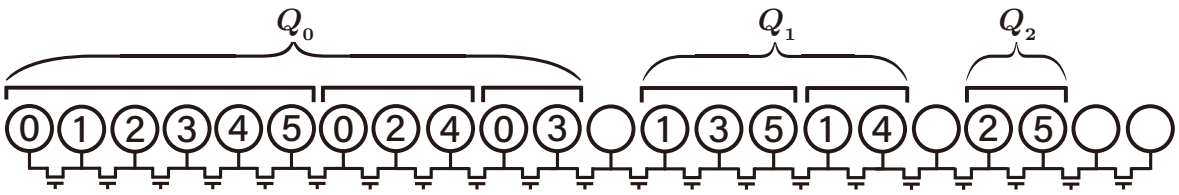
[5]When n is odd, by Theorem 1, the number of replicates is $\binom{n}{2}+1$ and the number of replicates for $n-1$ (even n) is $(n-1)^2/2$. Therefore $\binom{n}{2}+1-(n-1)^2/2 = (n-1)/2+1$ empty spaces remain. Subtracting the number of Q_i's minus 1 from $(n-1)/2+1$ gives 2.

$$p = (0, 1, 2, 3, 4, 5)$$
$$p^2 = (0, 2, 4)\ (1, 3, 5)$$
$$p^3 = (0, 3)\ (1, 4)\ (2, 5)$$

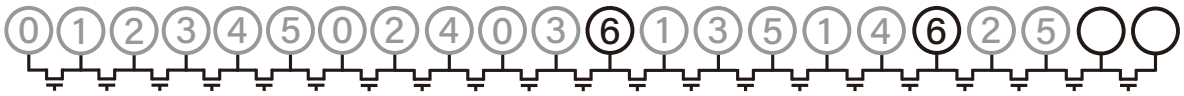
(a) The cyclic permutation for $n - 1 = 6$ classes.

$$Q_0 = \big\{(0, 1, 2, 3, 4, 5), (0, 2, 4), (0, 3)\big\}$$
$$Q_1 = \big\{(1, 3, 5), (1, 4)\big\}$$
$$Q_2 = \big\{(2, 5)\big\}$$

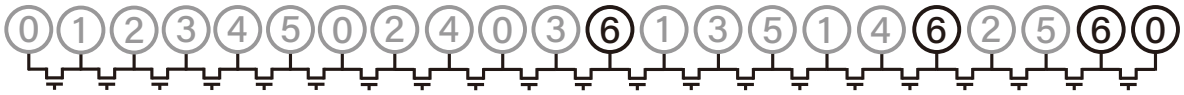
(b) All the cycles has been partitioned into Q_i, $0 \leq i \leq \frac{n-1}{2} - 1$.



(c) 1D-Crosspoint Array for $n - 1 = 6$.
Note that a replicate space was skipped between consecutive Q_i's.



(d) The replicate of class $n - 1$ is placed in the skipped replicate spaces.



(e) The replicate of class $n - 1$ and 0 are placed at the end.

Figure 2.4: Construction of 1D-Crosspoint Array for $n = 7$ case.

$n-1$ and class $1, 2, \cdots, n-3$. This can be verified by identifying the right-hand and left-hand side replicates of the skipped replicate spaces. On the right-hand side of the skipped replicate space will be the first replicate in each of the cycles in Q_i, $1 \leq i \leq \frac{n-1}{2} - 1$, i.e., replicates of class $1, 2, \cdots, \frac{n-1}{2} - 1$. On the left-hand side of the skipped replicate space will be the second replicate of the cycles in $p^{\frac{n-1}{2}}$ except the one cycle belonging to $Q_{\frac{n}{2}-1}$, i.e., replicates of class $\frac{n-1}{2}, \frac{n-1}{2} + 1, \cdots, n-3$. Therefore, by placing the replicate of class $n-1$ in the skipped replicate spaces we form all the adjacencies between class $n-1$ and class $1, 2, \cdots, n-3$. The only adjacencies that are not yet formed are pairs $(n-1, 0)$ and $(n-1, n-2)$. These two adjacencies are formed by placing replicate of classes $n-1$ and 0 at the end of the 1D-Crosspoint Array.

Note that the number of replicates we have placed after constructing a 1D-Crosspoint Array for $n-1$ classes is $\frac{n-1}{2} - 1 + 2 = \frac{n+1}{2}$. Therefore, the total number of replicates is $\frac{n+1}{2} + \frac{(n-1)^2}{2} = \frac{n(n-1)}{2} + 1$, which matches the minimum number of replicates needed for odd n from Theorem 1 and Proposition 1. The ongoing discussion shows that, when n is odd a 1D-Crosspoint Array with n classes and complete adjacency can be constructed so that every pair of n classes is adjacent only once. We state this formally in the following remark as it will play a key role in the memory requirements of the sorting algorithm that is presented in the next chapter.

Remark 1b. For all odd n, when an 1D-Crosspoint Array is constructed by Algorithm 1, each class of replicates is adjacent to every other class of replicates exactly once. □

Chapter 3: Parallel Sorting Algorithms on 1D-Crosspoint Array

In this chapter, we explore how a 1D-Crosspoint Array can be used to significantly reduce the execution time of big data queries such as sorting and searching.

3.1 Introduction

With emerging big data problems in distributed and cloud-based computing systems, designing efficient on-chip networks for fast searching and sorting extremely large sets of data has become a critical task as they form the foundations of most all other operations and queries[61, 62, 63]. Here, we assume that an external searching and sorting algorithm such as those described in [64] is used to handle such big data sets, and searching and sorting time is dominated by the internal searching or sorting phase of external searching and sorting algorithms. This assumption holds if the big data is partitioned into a relatively few sets of large subsets of data, each of which can be searched or sorted by an internal algorithm. To be sure, there exist myriad searching and sorting algorithms [64, 65] that can be applied to processing the internal searching and sorting phase of big data in distributed and cloud-based systems, but many of these algorithms are either not sufficiently fast to deal with such large data sets or they require parallel computers with many PEs with overly complex interconnection networks. In this dissertation, we will use the 1D-Crosspoint Array that was introduced in the earlier chapter to implement the internal searching or sorting steps

using $O(n^2)$ PEs. As we described before, each PE represents a replicate with its class identified by its index. Thus, PE_i is a replicate in class $i, 0 \leq i \leq n-1$. Each PE serves as a simple compare-and-exchange operator together with some simple combinational circuit functions. While using $O(n^2)$ PEs may seem excessive at first, we note that Valiant provided a theoretical parallel sorting model that exhibits a spectrum of multiprocessor architectures, trading the number of PEs with sorting time complexity without actually providing an actual construction for each architecture in the spectrum[66]. Our work focuses on one end of this spectrum, where we aim to obtain O(1) time sorting and searching algorithms using $O(n^2)$ PEs in our 1D-Crosspoint Array. Our search algorithm takes $O(1)$ time and sorting algorithm takes $O(\lg n \lg\lg n)$ time with constant fan-in logic gates and $O(1)$ time with threshold gates. These time complexities compare favorably against parallel searching and sorting architectures that have previously been reported in the literature, especially given the fact that it can be laid out in a VLSI chip with $O(n^2)$ area, where all interconnections are local to physically adjacent PEs. For comparison, we provide a brief survey of such architectures here. Earliest results on parallel sorting appeared in [66, 67, 68, 69, 70, 71, 72, 73]. Batcher's sorters are non-adaptive or oblivious in that they are constructed by a set of 2×2 switches connected together in stages to compare and exchange keys to sort them [67]. Batcher's odd-even and bitonic sorters use $O(n \lg^2 n)$ compare-and-exchange switches and have $O(\lg^2 n)$ sorting time. Another non-adaptive sorting network is the AKS network that can sort a set of n keys in $O(\lg n)$ time using $O(n \lg n)$ compare/exchange switches. The main issue with the AKS network is the large constants in its hardware and sorting time complexities. Adaptive techniques are also used in sorting as described [74] for sorting binary keys in $O(\lg n)$ gate delays with $O(n)$ constant fan-in and fan-out gates. Several other parallel realizations of sorting algorithms have been reported in the literature as well. Many of these rely

on a mesh-connected parallel computer model [70, 72], or more generic SIMD processors models [68, 69]. In general, the studies on mesh-connected parallel computer models establish that n keys can be sorted on a $\sqrt{n} \times \sqrt{n}$ mesh in $O(\sqrt{n})$ time with different constants in the order of time complexity that range between 3 and 6. Another array processor model was introduced in [16] to sort a list of n keys in $2n$ time using $O(n)$ PEs and $O(n)$ memory space. More recent results on sorting on parallel computers with a mesh topology make stronger claims on the time complexity of sorting, effectively reducing the time complexity of sorting to $O((\lg n)(\lg \lg n))$ with n PEs. Examples of such results include those that appeared in [17, 18, 19, 20, 40]. These efforts rely on a reconfigurable pipelined bus system, called *the LARPBS* (Linear Array, Reconfigurable Pipelined Bus System). The work in [20] reduced the time complexity further to $O((\lg \lg n)^2)$ using $n^{1+\epsilon}$ PEs, where $0 < \epsilon < 1$, and the result in [40] provided an $O(\lg n)$-time sorting algorithm on the LARPBS model with $O(n)$ PEs. The time complexity of sorting was reduced further to $O(1)$ in [75] using a mesh topology by assuming that PEs in such a topology may communicate with each other in $O(1)$ time. We find this assumption impractical as it ignores hops between PEs by asserting that transmission of information between PEs take $O(1)$ time regardless of where they are located on the mesh topology. Independent from these results, Valiant reported an abstract parallel processor model and studied the parallel time complexity of merging and sorting problems without specifying a particular topology [66]. He presented both lower and upper bounds on the parallel time complexity of merging and sorting on this abstract model. Valiant's work is important in that it guides what is feasible theoretically as the number of PEs is varied from two PEs to $\frac{n(n-1)}{2}$ PEs, even though it is not at all obvious how his merging and sorting algorithms can be mapped to an actual parallel processor architecture. As stated earlier, in this paper we fill this void in one particular case, namely when Valiant's abstract

model assumes $\frac{n(n-1)}{2}$ PEs, using our 1D-Crosspoint Array. We show that the 1D-Crosspoint Array provides an $O(1)$ time sorting algorithm, matching the time complexity of sorting a list elements on Valiant's parallel processing model [66] when $O(n^2)$ PE's are used. It should be pointed out that Valiant uses an underlying topology with $O(n)$ fan-in and fan-out to sort a list of n elements. The underlying topology of our 1D-Crosspoint Array has a fan-in and fan-out of 2, but to accomplish $O(1)$ sorting time, we employ a threshold logic circuit, which by definition requires $O(n)$ fan-in. We also employ a distribution network with $O(n)$ fan-out in our $O(1)$ time sorting algorithm. The same assumptions hold for finding the minimum and maximum of a list of elements as in Valiant's parallel processor model. In order to deal with big data queries in distributed and cloud-based systems, the 1-D Crosspoint Array is incorporated into three models that combine external and internal sorting and searching steps together, and time complexity of sorting and searching is determined in each case.

3.2 Sorting on the 1D-Crosspoint Array

In this section, an $O(1)$ time internal sorting algorithm, which is based on enumeration sort and devised for the 1D-Crosspoint Array is presented. External sorting models that incorporate the 1D-Crosspoint Array-based internal algorithm into an overall big data sorting system will be described in Section 3.4. In 1D-Crosspoint Array, each PE will serve as a simple compare-and-exchange operator together with some simple combinational circuit functions. The proposed algorithm, Algorithm 2, is a variant of enumeration sort[64, p.75], also known as count sort. In general, enumeration sort involves two steps. First, each element in a set of elements A is compared with every other element. We assume that the comparison of any two elements in

Algorithm 2: Sorting algorithm for a 1D-Crosspoint Array

Input: A, a vector storing n input elements

Output: R, a vector storing ranks of n elements

// $C_{i,j}$ represent j-th PE (or replicate) of class i, where $0 \leq i \leq n-1$, $0 \leq j \leq \frac{n}{2}-1$

// C_{i_r,j_r}, C_{i_l,j_l} denote right and left neighbors of $C_{i,j}$

// T is an $n \times n$ matrix

1 **parFor** $(0 \leq i \leq n-1, 0 \leq j \leq \frac{n}{2}-1)$ **do**

2 **if** $(i_l < i)$ **then**

3 Send $A[i]$ to left neighbor

4 **if** (*Signal from left* $== 1$) **then**

5 $T[i][i_l] \leftarrow 1$

6 **else**

7 Receive $A[i_l]$ from left neighbor

8 **if** $(A[i_l] < A[i])$ || $(A[i_l] = A[i])$ **then**

9 $T[i][i_l] \leftarrow 1$

10 Send 0 to left neighbor

11 **else if** $(A[i_l] > A[i])$ || $(A[i_l] = A[i])$ **then**

12 Send 1 to left neighbor

13 **if** $(i_r < i)$ **then**

14 Send $A[i]$ to right neighbor

15 **if** (*Signal from right* $== 1$) **then**

16 $T[i][i_r] \leftarrow 1$

17 **else**

18 Receive $A[i_r]$ from right neighbor **if** $(A[i_r] < A[i])$ || $(A[i_r] = A[i])$ **then**

19 $T[i][i_r] \leftarrow 1$

20 Send 0 to right neighbor

21 **else if** $(A[i_r] > A[i])$ || $(A[i_r] = A[i])$ **then**

22 Send 1 to right neighbor

23 **parFor** $0 \leq i \leq n-1$, **do**

24 $R[i] = \sum_{k=0}^{n-1} T[i][k]$

A takes $O(1)$ time. Otherwise, it should be factored into the overall time complexity of our sorting algorithm. Second, when all comparisons are completed, for each element, the number of elements that are less than that element is counted, which then gives its rank.

The Algorithm 2 follows this format as we now explain, and the execution of the algorithm is illustrated in Fig. 3.1 for $n = 4$ with an array $A = [6, 7, 8, 5]$ of input values, and in Fig. 3.2 for $n = 5$ with an array $A = [8, 6, 9, 5, 7]$ of input values. The input elements, $A[i]$, $i \leq 0 \leq n - 1$, are assumed to be preloaded to PE $C_{i,j}$, $0 \leq j \leq \frac{n}{2} - 1$. Each of the n input elements is loaded into $\frac{n}{2}$ different PEs each, giving the area for the loading circuit of $O(n \times \frac{n}{2}) = O(n^2)$ as shown in Fig. 3.3. The first `parFor` loop performs the first step of parallel enumeration sort, comparing every element with every other element. As the 1D-Crosspoint Array provides a crosspoint between any two PE classes, this task is carried out in a distributed manner. Each pair of neighboring PEs essentially compare their input elements that they were loaded as follows. Between any two neighboring PEs, the PE with the greater class number sends its element to the PE with the smaller class number (See the third column in Fig. 3.1 and 3.2). Then the PE with the smaller class number carries out the comparison between the received element and the element that it was loaded with. This is done by the `if-else` statements beginning in lines 2 and 13. The two `if-else` statements are identical except that one of them is for communicating with the right-hand side PE, and the other is for communicating with the left-hand side PE. During each of these comparisons, if the PE that carried out the comparison has the greater element, it writes a 1 to the $T[i][i_l]$ (or $T[i][i_r]$), then sends a 0 to the neighboring PE with the smaller element to indicate that the comparison is done. When the PE that carries out the comparison has the smaller element, it sends a 1 to the neighboring PE with the greater element, notifying it that 1 has to be written to the $T[i_l][i]$ (or $T[i_r][i]$). At the end of this step, the array T is set

	$C_{i,j}$	`parFor`(line 2, 13)	`parFor`(line 8, 18)	Array T after line 22	`parFor` (line 23)
⓪	$C_{0,0}$ $A[0] = 6$	$i_r = 1 \not< i = 0$ $\therefore$ Rx $A[1]$ from right	$A[1] = 7 \not< A[0] = 6$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	$R[0] = 1$
①	$C_{1,0}$ $A[1] = 7$	$i_l = 0 < i = 1$ $\therefore$ Tx $A[1]$ to left $i_r = 2 \not< i = 1$ $\therefore$ Rx $A[2]$ from right	Rx 1 from left $\therefore T[1][0] \leftarrow 1$ $A[2] = 8 \not< A[1] = 7$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0&0&0&1 \\ \mathbf{1}&0&0&1 \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	$R[1] = 2$
②	$C_{2,0}$ $A[2] = 8$	$i_l = 1 < i = 2$ $\therefore$ Tx $A[2]$ to left $i_r = 3 \not< i = 2$ $\therefore$ Rx $A[3]$ from right	Rx 1 from left $\therefore T[2][1] \leftarrow 1$ $A[3] = 5 < A[2] = 8$ $\therefore T[2][3] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ 1&\underline{\mathbf{1}}&0&\mathbf{1} \\ 0&0&0&0 \end{bmatrix}$	$R[2] = 3$
③	$C_{3,0}$ $A[3] = 5$	$i_l = 2 < i = 3$ $\therefore$ Tx $A[3]$ to left $i_r = 0 < i = 3$ $\therefore$ Tx $A[3]$ to right	Rx 0 from left Rx 0 from right	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	$R[3] = 0$
⓪	$C_{0,1}$ $A[0] = 6$	$i_l = 3 \not< i = 0$ $\therefore$ Rx $A[3]$ from left $i_r = 2 \not< i = 0$ $\therefore$ Rx $A[2]$ from right	$A[3] = 5 < A[0] = 6$ $\therefore T[0][3] \leftarrow 1$, Tx 0 to left $A[2] = 8 \not< A[0] = 6$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0&0&0&\mathbf{1} \\ 1&0&0&1 \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	
②	$C_{2,1}$ $A[2] = 8$	$i_l = 0 < i = 2$ $\therefore$ Tx $A[2]$ to left $i_r = 1 < i = 2$ $\therefore$ Tx $A[2]$ to right	Rx 1 from right $\therefore T[2][0] \leftarrow 1$ Rx 1 from right $\therefore T[2][1] \leftarrow 1$	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ \mathbf{1}&\underline{\mathbf{1}}&0&1 \\ 0&0&0&0 \end{bmatrix}$	
①	$C_{1,1}$ $A[1] = 7$	$i_l = 2 \not< i = 1$ $\therefore$ Rx $A[2]$ from left $i_r = 3 \not< i = 1$ $\therefore$ Rx $A[3]$ from right	$A[2] = 8 \not< A[1] = 7$ $\therefore$ Tx 1 to left $A[3] = 6 < A[1] = 7$ $\therefore T[1][3] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&\mathbf{1} \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	
③	$C_{3,1}$ $A[3] = 5$	$i_l = 1 < i = 3$ $\therefore$ Tx $A[3]$ to left	Rx 0 from left	$T = \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ 1&1&0&1 \\ 0&0&0&0 \end{bmatrix}$	

Figure 3.1: Illustration of sorting $n = 4$ elements using 1D-Crosspoint Array, with input data $A = \{6, 7, 8, 5\}$. It shows what is done in each `parFor` loop in Algorithm 2. Tx and Rx refers to transmit and receive respectively, and left and right refers to left-hand and right-hand side neighbor PE respectively.

	$C_{i,j}$	`if-else` (line 2, 13)	`if-else` (line 8, 18)	Array T after line 22	`parFor` (line 23)
⓪	$C_{0,0}$ $A[0] = 8$	$i_r = 1 \not< i = 0$ $\therefore$ Rx $A[1]$ from right	$A[1] = 6 < A[0] = 8$ $\therefore T[0][1] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0 & \mathbf{1} & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	$R[0] = 3$
①	$C_{1,0}$ $A[1] = 6$	$i_l = 0 < i = 1$ $\therefore$ Tx $A[1]$ to left $i_r = 2 \not< i = 1$ $\therefore$ Rx $A[2]$ from right	Rx 0 from left $A[2] = 9 \not< A[1] = 6$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	$R[1] = 1$
②	$C_{2,0}$ $A[2] = 9$	$i_l = 1 < i = 2$ $\therefore$ Tx $A[2]$ to left $i_r = 3 \not< i = 2$ $\therefore$ Rx $A[3]$ from right	Rx 1 from left $\therefore T[2][1] \leftarrow 1$ $A[3] = 5 < A[2] = 9$ $\therefore T[2][3] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	$R[2] = 4$
③	$C_{3,0}$ $A[3] = 5$	$i_l = 2 < i = 3$ $\therefore$ Tx $A[3]$ to left $i_r = 4 \not< i = 3$ $\therefore$ Rx $A[4]$ from right	Rx 0 from left $A[4] = 7 \not< A[3] = 5$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	$R[3] = 0$
④	$C_{4,0}$ $A[4] = 7$	$i_l = 3 < i = 4$ $\therefore$ Tx $A[4]$ to left $i_r = 0 < i = 4$ $\therefore$ Tx $A[4]$ to right	Rx 1 from left $\therefore T[4][3] \leftarrow 1$ Rx 0 from right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \mathbf{1} & 0 \end{bmatrix}$	$R[4] = 2$
⓪	$C_{0,1}$ $A[0] = 8$	$i_l = 4 \not< i = 0$ $\therefore$ Rx $A[4]$ from left $i_r = 2 \not< i = 0$ $\therefore$ Rx $A[2]$ from right	$A[4] = 7 < A[0] = 8$ $\therefore T[0][4] \leftarrow 1$, Tx 0 to left $A[2] = 9 \not< A[0] = 8$ $\therefore$ Tx 1 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	
②	$C_{2,1}$ $A[2] = 9$	$i_l = 0 < i = 2$ $\therefore$ Tx $A[2]$ to left $i_r = 4 \not< i = 2$ $\therefore$ Rx $A[4]$ from right	Rx 1 from left $\therefore T[2][0] \leftarrow 1$ $A[4] = 7 < A[2] = 9$ $\therefore T[2][4] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ \mathbf{1} & 1 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	
④	$C_{4,1}$ $A[4] = 7$	$i_l = 2 < i = 4$ $\therefore$ Tx $A[4]$ to left $i_r = 1 < i = 4$ $\therefore$ Tx $A[4]$ to right	Rx 0 from left Rx 1 from right $\therefore T[4][1] \leftarrow 1$	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 1 & 0 \end{bmatrix}$	
①	$C_{1,1}$ $A[1] = 6$	$i_l = 4 \not< i = 1$ $\therefore$ Rx $A[4]$ from left $i_r = 3 \not< i = 1$ $\therefore$ Rx $A[3]$ from right	$A[4] = 7 \not< A[1] = 6$ $\therefore$ Tx 1 to left $A[3] = 5 < A[1] = 6$ $\therefore T[1][3] \leftarrow 1$, Tx 0 to right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	
③	$C_{3,1}$ $A[3] = 5$	$i_l = 1 < i = 3$ $\therefore$ Tx $A[3]$ to left $i_r = 0 < i = 3$ $\therefore$ Tx $A[3]$ to right	Rx 0 from left Rx 0 from right	$T = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	
⓪	$C_{0,2}$ $A[0] = 8$	$i_l = 3 \not< i = 0$ $\therefore$ Rx $A[3]$ to left	$A[3] = 5 < A[0] = 8$ $\therefore T[0][3] \leftarrow 1$, Tx 0 to left	$T = \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$	

Figure 3.2: Illustration of sorting $n = 5$ elements using 1D-Crosspoint Array, with input data $A = \{8, 6, 9, 5, 7\}$. It shows what is done in each `parFor` loop in Algorithm 2.

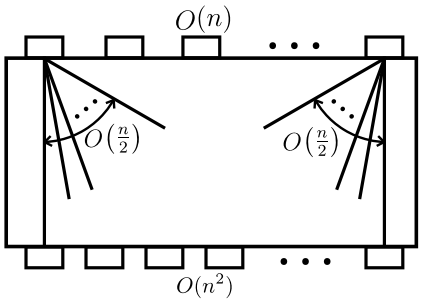


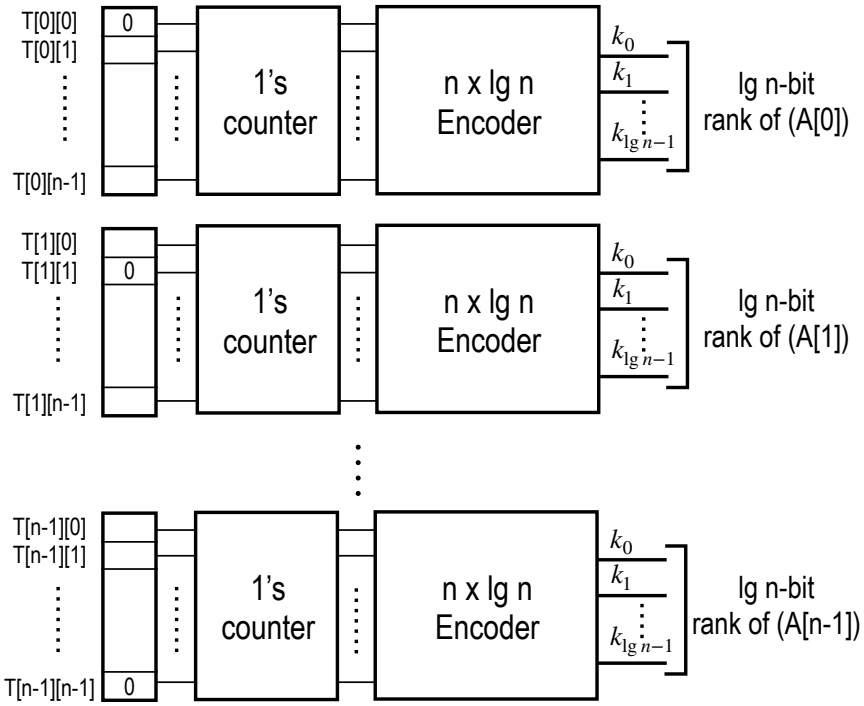
Figure 3.3: Illustration of the loading circuit. The n input elements at the top are loaded to the $O(n)$ PEs at the bottom.

in such a way that $T[i][i_l] = 1$ implies that $A[i_l] < A[i]$ (See the fourth column in Fig. 3.1 and 3.2). It may appear that this step requires a memory that supports a concurrent write in order to run in $O(1)$ time. We first note that when n is odd, by Remark 1b, the 1D-Crosspoint Array has only one adjacency between any two classes of PEs. Therefore, the locations to which the PEs write the comparison results are all distinct as seen in Fig. 3.2, and hence a concurrent write memory is avoided. On the other hand, when n is even, by Remark 1a, there are $\frac{n}{2} - 1$ redundant adjacencies on the 1D-Crosspoint Array. For example, it is seen in Fig. 3.1 that the comparison between PE classes 1 and 2 is done twice, and therefore the PEs $C_{2,0}$ and $C_{2,1}$ within the same class of PEs, i.e., class 2, attempt to write to location $T[2][1]$ at the same time (see the bold faced elements in T in the fifth column). However, a simple workaround to avoid multiple writes when n is even, is adding a dummy class to switch to a 1D-Crosspoint Array with an odd number of PE classes. Thus, assuming that we have an odd number of PE classes, and given that the number of `if-else` statements that a PE in any given class executes is at most two, this `parFor` loop also takes only $O(1)$ time to complete.

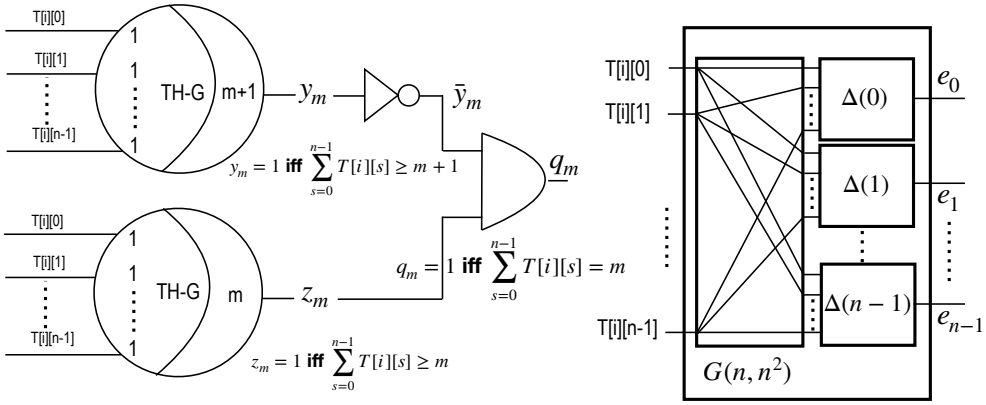
Lastly, the last `parFor` loop in line 23 computes the rank of each element, which corresponds to the second task of enumeration sort. This step is executed by each class of PEs separately, where up to $\frac{n}{2}$ PEs in each class are used to carry out addition operations. Thus, ac-

cessing the entries in $T[i][\,]$ cannot result in any contention across the PE classes. We further note that each 1 in $T[i][\,]$ implies that there is an element less than $A[i]$, and hence counting the number of 1's or summing the entries in $T[i][\,]$ gives the rank of the $A[i]$. In a way, the vectors stored in $T[i][\,], 0 \leq i \leq n-1$ may be viewed as encoded representations of the ranks of the elements in A, and no further operations may be needed. This makes the time complexity of sorting a set of n elements $O(1)$ on an 1D-Crosspoint Array with $O(n^2)$ PEs if the ranks of the elements in A are not needed in decimal or some other, preferred number representation. If the ranks are desirable in decimal or some other numerical notation, the binary entries in $T[i][], 0 \leq i \leq n-1$, can be summed in $O(\lg n \lg\lg n)$ time using a binary tree of n Brent-Kung adders[76], each with two $\lg n$-bit operands and consisting of $O(\lg n)$ logic gates with constant fan-in to compute the ranks of all elements on n PEs in parallel, where $C_{i,0}, 0 \leq i \leq n-1$ computes the rank of $A[i]$.

This increases the overall gate-level time complexity of sorting a set of n elements from $O(1)$ to $O(\lg n \lg\lg n)$, using $O(n \lg n)$ additional logic gates with constant fan-in per PE. We note that it is impossible to reduce the time complexity to $O(1)$ using a polynomial order of AND, OR and NOT gates even with unbounded fan-in[77, 78, 79, 80]. Instead, we use threshold logic as shown in Fig. 3.4. The diagram in Fig. 3.4(a) depicts the overall layout of our architecture. The ith 1's-counter sets its e_m output to 1 when its input string $T[i][]$ has exactly m 1's, $0 \leq i \leq n-1$. The n-input, $\lg n$-output encoder then generates the rank for $T[i][], 0 \leq i \leq n-1$. The 1's counters (part (c)) are constructed from $\Delta(m)$ circuits, each of which is obtained by a pair of threshold gates, where each such gate is assumed to have $O(1)$ delay, which is the standard assumption in threshold logic circuits, even though practical physical constraints may make this assumption overly optimistic[79]. The bipartite graph $G(n, n^2)$ replicates each of its n inputs, and connects each copy to one of the inputs of the n inputs of each $\Delta(m)$ circuit as shown in the figure. The



(a) String of 1's to rank converter.



(b) #1's = m circuit, $\Delta(m)$ (c) 1's counter

Figure 3.4: Computing the ranks from the T matrix.

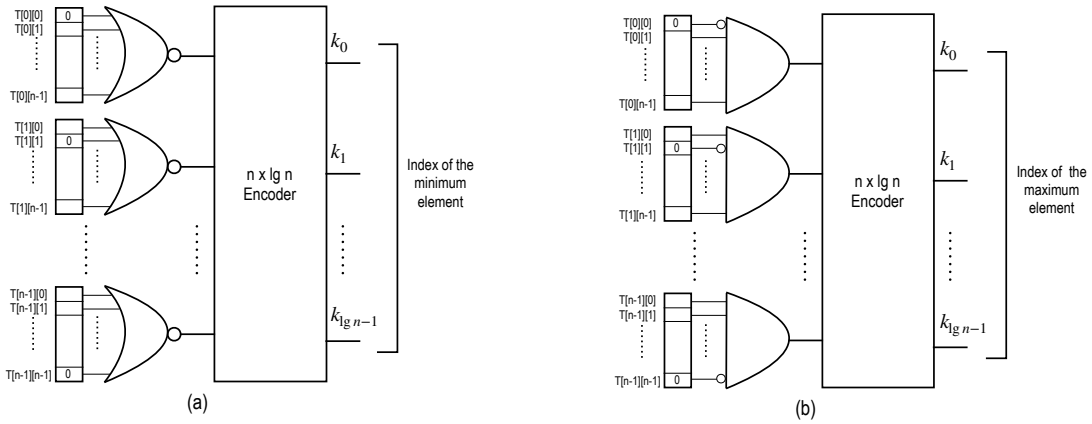


Figure 3.5: Computing the index of the minimum and maximum elements in an array of n elements.

upper threshold gate in a $\Delta(m)$ circuit, (in part (b)), together with an inverter produces a 1 when fewer than $m + 1$ of its binary-valued inputs are equal to 1, whereas the lower threshold gate produces a 1 when m or more of its binary-valued inputs are equal to 1. Combining the outputs of the two threshold gates with an AND gate thus detects if the number of 1's in the input string $T[i][]$ is equal to m. Given that the path from an input to the output of a $\Delta(m)$ circuit traverses a threshold gate, an inverter, and AND gate, its output is computed in $O(1)$ time. Therefore, each output of a 1's counter is also computed in $O(1)$ time. Given that the ranks are computed by a cascade of a 1's counter and an n-input, $\lg n$-output encoder, the ranks are computed in $O(1)$ time as well. Hence, sorting of n numbers is completed in $O(1)$ time using $O(n^2)$ threshold gates. It should be pointed out that sorting is known to be in class TC^0 of problems[81] in which every problem can be solved using a Boolean circuit with $O(1)$ depth that is constructed out of AND, OR, and threshold gates that grows by a polynomial function of the problem size n. Our rank computation architecture provides an actual circuit with threshold gates and other combinational circuit logic that complete our $O(1)$ time sorting algorithm with $O(n^2)$ gates, matching both the depth and circuit complexity bounds.

3.3 Other Sorting-Related Queries

Besides sorting, a significant utility of this encoded form of sorting is the ease with which the minimum or maximum of the elements in A can be determined. Both these tasks can be completed in $O(1)$ time using simple logic gates, and an encoder consisting of OR gates with a fan-in of n. To see why, suppose that the index of the minimum element of A is $i_{\min}$. Then $T[i_{\min}][\,]$ should be $[0, 0, \cdots, 0, 0]$. Thus, if we apply a NOR function to the elements in each row as shown in Fig. 3.5(a), only the row corresponding to the minimum element should output 1. Therefore, we can compute i_{min} by cascading n number of n-input NOR gates with an n-input by $\lg n$-output encoder in $O(1)$ time. The time complexity increases to $O(\lg n)$ if the fan-in of OR gates within the encoder is assumed to be a constant or the fan-in of the NOR gate that is applied to each row of bits in T is constant. Similarly, suppose that the index of the maximum element of A is i_{max}. Then $T[i_{\max}][]$ should be all $1'$s except $T[i_{\max}][i_{max}]$. Therefore, if we apply an AND function to each row after complementing the diagonal elements in T, then only the row that corresponds to the maximum element should output 1. Again, an encoder may be used to obtain the index of the maximum element in A in $O(1)$ time with unlimited fan-in gates and in $O(\lg n)$ time with constant fan-in gates as shown in Fig. 3.5(b). In both cases, we only use $O(\lg n)$ logic gates with n fan-in for the encoder and $O(n)$ OR or AND gates for the first stage in Fig. 3.5.

This querying procedure can be extended to answering other queries as well. For example, we can determine if any particular element in A has a rank of j or higher, $1 \leq j \leq n$. The $j = 1$ case is trivial as it only requires an OR-gate with n inputs. If $j = 2$ then we can add the bits in the corresponding row in the T matrix in pairs to determine if any particular element in A has a rank of j or higher with high probability in $O(1)$ time. To see why, note that exactly three out of

four binary patterns $00, 01, 10, 11$ result in a sum of zero or one, and hence the number of n-bit patterns in which all $n/2$ pairs sum to less than two is $3^{n/2}$. Dividing this by the total number of n-bit patterns, we find that the probability of incorrectly guessing that the rank of an element is at least two tends to 0 as the following computation shows

$$\frac{3^{n/2}}{2^n} = \frac{1}{2^{n(1-0.5\lg 3)}} = \frac{1}{2^{0.2175n}} \rightarrow 0 \text{ as } n \rightarrow \infty.$$

Thus $\frac{n}{2}$ 2-input AND gates together with an $\frac{n}{2}$-input OR gate suffice to determine if the rank of any given element is at least 2 with high probability. In general, adding m bits together, $m \geq 2$ leads to the probability of incorrectly guessing that the rank of an element is at least $j, 1 \leq j \leq n$ is given by

$$\frac{\left\{2^m - \sum_{i=0}^{j-1} \binom{m}{i}\right\}^{n/m}}{2^n} \rightarrow 0 \text{ as } n \rightarrow \infty.$$

We note that it is always possible to determine the rank of any particular element in A exactly, by summing the entries in the corresponding row in the T matrix by a binary tree adder in $O(\lg \lg n)$ time.

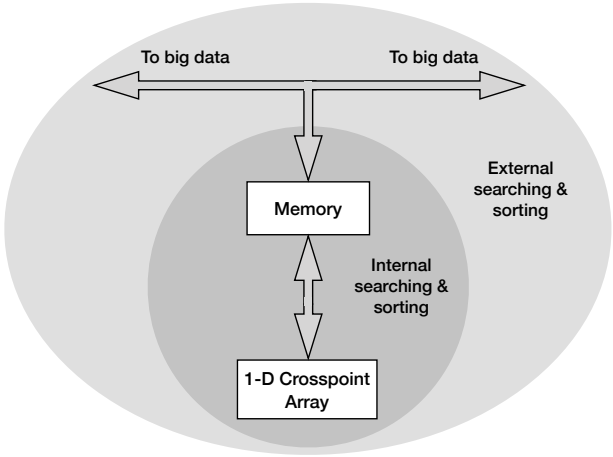
Querying the i-th largest element is only a little more involved. Suppose that the index of the i-th largest element is i_x. Then the sum of the 1's in $T[i_x][\,]$ must be equal to i. Moreover, the sum of the 1's in no other row should be equal to i_x. Therefore, if we subtract i from the sum of all entries in the rows in T then the row that produces a zero provides the key to the i-th largest element. Once identified, the index of this row can be captured as before using an n-input by $\lg n$-output encoder in $O(1)$ or $O(\lg n)$ time as in the prior cases. All we need to do is to flip the bits that are generated by subtraction operations, before feeding them to the encoder. The

only other time complexity that must be added to the total time complexity is that of a $\lg n$-bit subtraction operation. This can be carried out by the first n PEs, $C_{i,0}, 0 \leq i \leq n-1$ in $O(1)$ time, using $O(n \lg^2 n)$ logic gates with $O(n)$ fan-in or $O(\lg \lg n)$ gate-level time using a $\lg n$-bit prefix adder[76] with $O(n \lg n)$ constant fan-in logic gates.

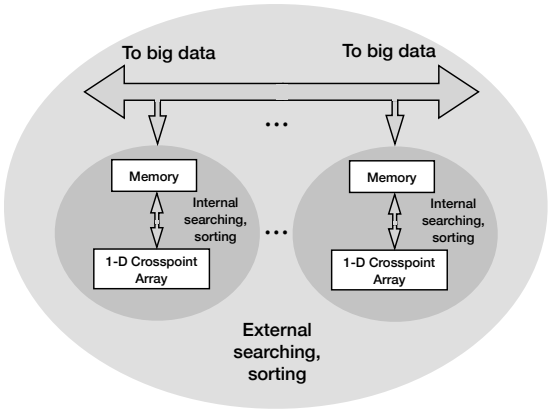
Another possible query is searching for a specific number in a given array. This can be accomplished with an encoder and slightly modified Algorithm 2. Since we only need to compare each element with the number we are searching for, not with $n-1$ other elements, only one replicate from each class is needed to check if the element it has is equal to the number that is searched in the `if` statement in line 8, and 18. We would not need the following `else` blocks in line 11, and 21, and the array T can be a vector of size $n \times 1$. Each element in array T will then be fed into the $n \times \lg n$ encoder, which will output the index of the number that is searched. Due to the fact that algorithm for search query is a cut-down version of the Algorithm 2, and does only require a $n \times \lg n$ encoder, the searching among n elements take $O(1)$ time in the 1D-Crosspoint Array.

3.4 External Sorting Models

As we stated in the introduction, big data sorting algorithms involve dividing a sorting problem into smaller sorting problems and merging their solutions together. Such a technique may be implemented in several ways and there exists a number of software tools such as Google's MapReduce[52, 82, 83] and Apache's Spark[84, 85] that make use of such divide and conquer techniques. The three models introduced in this section is based on the same principle as shown in Fig. 3.6, where a part of an external sorting problem is mapped onto a parallel processor with



(a) External sorting model with single 1D-Crosspoint Array.



(b) External sorting model with multiple 1D-Crosspoint Arrays (k copies of 1D-Crosspoint Array are used.)

Figure 3.6: Two implementation types of the 1D-Crosspoint Array for big data sorting. Big data is assumed to be stored in a cloud network and merging sorted lists is carried by the cloud network itself.

a given number (n) of PEs. It is assumed that some big data of size N is stored over a cloud network[6], and parts of it are downloaded into the local memory of one or more searching and sorting engines, 1D-Crosspoint Arrays in our case, to handle the internal searching and sorting steps. Such steps may be interleaved with periodic downloads and uploads, where merging steps are carried out by the servers on the cloud network.

[6]Throughout the section, N will denote the total number of elements in some big data set, and n will denote a subset of that big data set under the relation $N = nk$, where k is some fixed number, or $N = nkr$, where k and r are some fixed numbers.

Model 3-A. *Single Sorting Engine Model*: In the model in Fig. 3.6(a), the time complexity of an algorithm for searching or sorting N data elements in N/k portions at a time may be expressed as

$$T_{total}(N,n) = kT_i(n) + kT_{d,u}(n) + T_m(n)$$

$$= \frac{N}{n}T_i(n) + \frac{N}{n}T_{d,u}(n) + T_m(n) \tag{3.1}$$

where $T_i(n)$ denotes the time taken by the 1D-Crosspoint Array or another sorting engine to sort $n = \frac{N}{k}$ operands, $T_{d,u}(n)$ denotes the time it takes to download and upload data of size n, and $T_m(n)$ accounts for the time to merge the results together. For future reference, we will denote the hardware complexity of all the internal searching and sorting processor combined by H_i.

Merging k lists: For the purpose of this dissertation, it will be assumed that a cloud network can merge any k sorted lists using the best-known merging algorithm, which at the time of this writing is $O((\lg k)(\lg\lg nk))) = O((\lg N/n)(\lg\lg N)))$. Such a merging algorithm is obtained by using the 2-way merge algorithm in [66] over a binary tree of $k/2$ PEs. More recent merging algorithms have been reported in the literature[86, 87]. The 2-way merge algorithm in [86] takes $O(1)$ time on a PRAM model that would result in an $O(\lg k)$ time k-way merge algorithm. However, it would be difficult to implement such a 2-way merging algorithm with big data lists. In [87], a direct k-way merging algorithm is described using perfect shuffles. This algorithm merges k lists that contain N elements altogether in $O(N/p)$ time [87, p. 11:13] using p PEs. The complexity tends to $O(1)$ as p approaches N, but this is not practical for very large data sets.

Downloading and Uploading Lists: $T_{d,u}(n)$ will be proportional to n assuming that the data channel width between the memory and the big data is $O(1)$. In this case the $kT_{d,u}(n) = k\alpha n = \alpha N$, where α is a constant that is determined by the download and upload speeds of the

network[7].

Overall Execution Time: Adding all the terms together, we have $T_{total}(N) = (N/n)T_i(n) + \alpha N + O((\lg N/n)(\lg \lg N)))$, which implies that reducing the sorting time beyond αN offers no benefit. Sorting n elements with algorithms with time complexity of $O(\lg n)$ or $O(\lg^2 n)$ (for example, see [67, 88]) will take $(N/n)T_i(n) = O((N/n)\lg(n))$ or $(N/n)T_i(n) = O((N/n)\lg^2(n))$ time. It is easy to verify that both these terms are less than $O(N)$, and hence $T_{total}(N) = O(N)$ in this case. Although it may seem that the pursuit for a faster sorting algorithm is no longer needed, this assumes that the data channel width between the memory and the big data would be $O(1)$. If the data channel width can be increased, then the $T_{d,u}$ can be reduced to $O(\lg^2 n)$ or even $O(\lg n)$, in which case, $T_i(n)$ and $T_m(n)$ become the dominating terms of $T_{total}(N)$. Under these circumstances, a parallel sorting algorithm with time complexity lower than $O(\lg n)$, such as the 1D-Crosspoint Array suffices to set $T_{total}(N)$ to $O((N/n)\lg n)$ or $O((N/n)\lg^2 n)$, depending on the parallel sorting algorithm used.

Model 3-B. *Multiple Sorting Engine Model*: The excessive $T_{d,u}$ can be further reduced using multiple 1D-Crosspoint Arrays or other sorting engines as shown in Fig. 3.6(b). In this case, the time complexity of an algorithm for searching or sorting N data elements on k 1D-Crosspoint Arrays may be expressed as

$$T_{total}(N, n) = T_i(n) + T_{d,u}(n) + T_m(n), \tag{3.2}$$

where $T_i(n)$, $T_{d,u}(n)$, and $T_m(n)$ are defined as before. Note that the $k = N/n$ factor no longer appears in the T_i and $T_{d,u}$ terms as internal sorting, download, and upload steps are all done in

[7] If the download and upload times are different then α will be the sum of download and upload times per unit data in each iteration of searching or sorting process.

parallel. Without the N/n factor, the T_{total} can be simplified to

$$T_{total}(N,n) = T_i(n) + \alpha n + O((\lg N/n)(\lg\lg N))),$$

or

$$T_{total}(N,n) = T_i(n) + O(1) + O((\lg N/n)(\lg\lg N))).$$

The first formula assumes that feeding a list of n elements to each sorting engine takes $O(n)$ time, whereas in the second formula, it is assumed that this happens in $O(1)$ time with a wider bus between the sorting engines and the cloud. For sorting with the first formula in place, a parallel sorting algorithm with $O(\lg^2 n)$ time complexity suffices to keep the overall time complexity to $O(n)$ as $T_{d,u} = O(n)$ dominates the overall time complexity. With the second formula, we again need T_i to be $O(1)$ not to increase the overall time complexity beyond $O(1)$ since $T_{d,u}(n)$ and $T_m(n)$ both tend to $O(1)$ as n tends to N. For other values of n, $T_i(n)$ must be matched with $T_m(n) = O((\lg N/n)(\lg\lg N)))$.

The foregoing analysis indicates that sorting algorithms with $O(1)$ time complexities are essential for big data processing.

Model 3-B highlights the possible tradeoffs between the time and hardware complexities of external sorting by dividing data of size N into k data sets of size $n = N/k$, which are then sorted on individual sorting engines. For moderate size data sets and big data (n in orders of gigabytes and N in orders of terabytes), this model can provide sufficiently fast executions of sorting algorithms on big data. However, as N becomes larger, entering the range of exabytes(10^{18}B)[9, 89], Model 3-B would demand excessive hardware to keep up with the execution speed requirements of external sorting algorithms. For example, if $N = 10^{18}$ and the model has 10^6 parallel sorting

engines such as 1-D Crosspoint Arrays or $k = 10^6$ then the hardware complexity of each such engine would require between 10^{12} and 10^{24} PEs. This is not feasible with today's chip technology as the highest performing supercomputers have approximately 10^7 cores[90]. On the other hand, if we let $n = N/k = 10^6$ in order to avoid such unfeasible hardware complexity then the model would require an excessive number of parallel sorting engines, i.e., $k = 10^{12}$ of them. These examples show that Model 3-B requires unreasonable hardware complexity when the scale of big data reaches or exceeds exabytes. To cope with this problem, a more realistic and feasible model is introduced below.

Model 3-C. *Multiple Sorting Engine-Realistic Model*: As in Model 3-B, Model 3-C consists of k 1D-Crosspoint Arrays and the data is divided into k datasets, each of their size being $\frac{N}{k}$. However, in Model 3-C, each of k datasets is further split into r *segments*, where $0 < r \leq \frac{N}{k}$, and each 1D-Crosspoint Array is configured to sort one segment in a single execution of the algorithm. In other words, the size of one segment is $\frac{N}{kr} = n$ in this model, and if we use 1D-Crosspoint Arrays to carry out sorting the segments then $O((\frac{N}{kr})^2)$ PEs would be needed to sort $\frac{N}{kr}$ elements in $O(1)$ time (see Fig. 3.7 for visual illustration of how N elements are split into datasets and segments). The hardware complexity of all the 1D-Crosspoint Arrays combined is $H_i = O(kn^2)$ since there are k 1D-Crosspoint Arrays. By introducing $r = N/kn$, the model provides more feasible configurations for data of massive scales. For the earlier example, where $N = 10^{18}$, by setting $n = 10^2$ and $k = 10^2$, the value of H_i can be confined to $10^2 \times 10^4 = 10^6$ PEs, instead of 10^{16} to 10^{32} PEs, depending on the hardware complexity of the n PE sorting engine.

Fig. 3.8 shows the choices of k and n in Model 3-C to keep the overall hardware complexity H_i within 10^7 PEs, which is currently the feasible number of cores within a single supercomputer

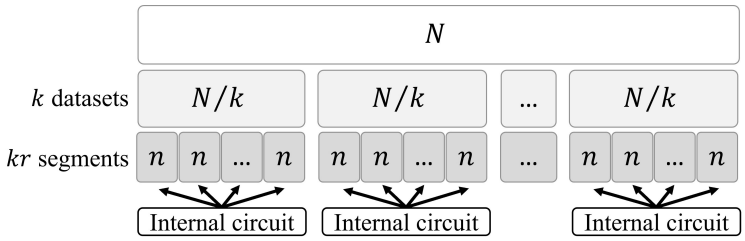


Figure 3.7: Illustration of how input is divided in Model 3-C. Each engine repeats r times, sorting one n-size segment each time.

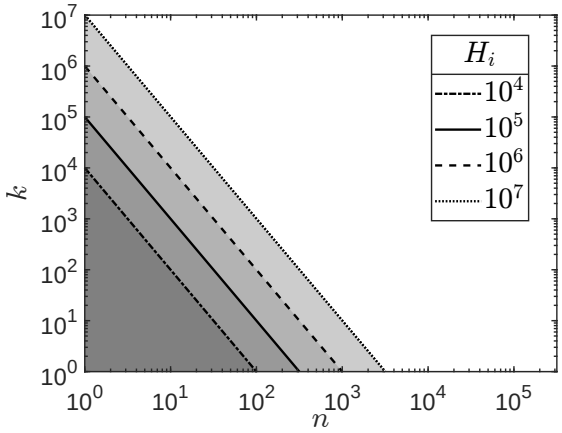


Figure 3.8: Feasible range of n and k to contain hardware complexity within 10^7.

system[90].

As a consequence of the lower hardware complexity, however, the time complexity of the algorithm increases. Assuming that the time complexity of a single execution of a sorting algorithm for an n-element list is $T_i(n)$ as before, its contribution to the T_{total} is $rT_i(n) = \frac{N}{kn}T_i(n)$, since the model requires the execution of the algorithm $\frac{N}{kn}$ times. Therefore, the time complexity of sorting N data elements on Model 3-C may be expressed as

$$T_{total}(N,k,n) = \frac{N}{kn}T_i(n) + T_{d,u}(n) + T_m(N,k). \tag{3.3}$$

If n is kept small to avoid excessive hardware complexity, $\frac{N}{kn}$ will be large and increase $T_{total}(N,k,n)$. On the other hand, if $\frac{N}{kn}$ is made small to keep the $\frac{N}{kn}T_i(n)$ term small, n will

be large and increase the hardware complexity. Similar to Model 3-B, the $T_{d,u}$ can be reduced by increasing the bandwidth of the data channel. In such a case, the best time complexity can be achieved by reducing the $T_i(n)$ as much as possible, ideally reducing it to $O(1)$. It is established in Section 2.1 that the 1D-Crosspoint Array performs sorting an n element list in $O(1)$ time, which thus makes the overall time complexity of external sorting $O(\frac{N}{kn})$ on Model 2C for any given n.

3.5 Comparison of Sorting Algorithms

Tables 3.1(a), (b), and (c) summarize the time and hardware complexities of representative algorithms mentioned in this chapter. The total time complexity(T_{total}) and hardware cost of each algorithm(H_i) along with the time complexities of sorting(T_i), downloading/uploading($T_{d,u}$), and merging steps(T_m) are listed using Eqns. (3.1), (3.2), and (3.3). The downloading and uploading channel was assumed to be wide enough to let $T_{d,u}$ be as small as $O(1)$, and thus was ignored in the T_{total}. Let us first compare the algorithms solely from a performance perspective without taking the hardware complexity into account. The total time complexities in the Table 3.1b are plotted in Fig. 3.9 for $N = 10^{18}$ considering the actual constants involved in the complexities with and without the AKS sorting algorithm. The AKS algorithm has a much higher time complexity due to the large constant multiplied to its time complexity expression. The Batcher's bitonic sorting network(BAT) and Cole's parallel merge sort(CVN) follows the AKS network. Here, it must be noted that although the AKS network and the parallel merge sort has the same total time complexity, the graph of the two diverge. This is because T_m becomes $O(\lg(1/n)) = O(-\lg n)$ when N is fixed and offsets $T_i = O(\lg n)$. Because the constant for AKS network(6100) is much

	$T_{total,3\text{-}A}(N,n)$	$T_i(n)$	$T_{d,u}(n)$	$T_m(N,n)$	$H_{i,3\text{-}A}$
VAL	$O(\frac{N}{n} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n^2)$
PRP	$O(\frac{N}{n}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n\lg n)$
AKS	$O(\frac{N}{n}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n\lg n)$
BAT	$O(\frac{N}{n}\lg^2 n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg^2 n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n\lg^2 n)$
CVN	$O(\frac{N}{n}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n)$
RCF	$O(\frac{N}{n} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n^2)$
XPA	$O(\frac{N}{n} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(n^2)$

(a) Summary table for Model 3-A based on Eqn. 3.1. Note that $N/k = n$.

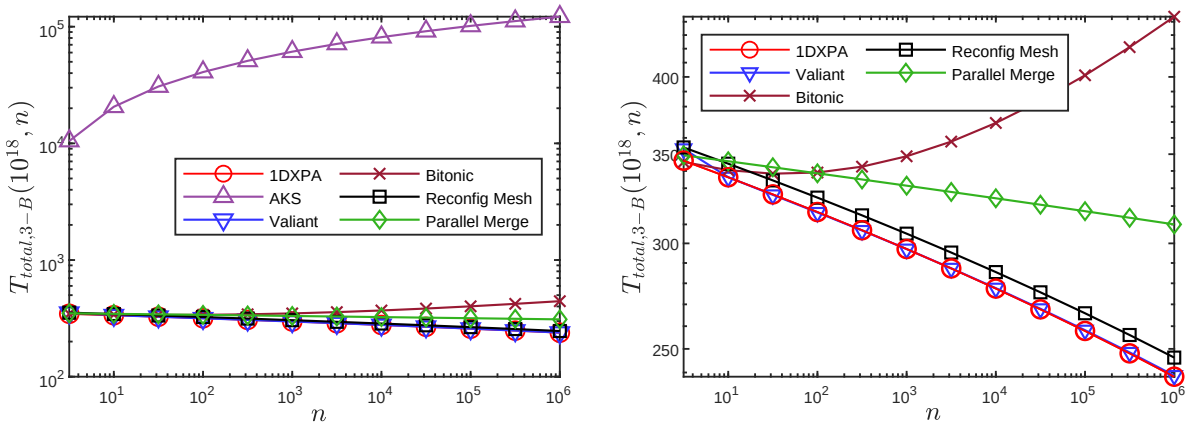
	$T_{total,3\text{-}B}(N,n)$	$T_i(n)$	$T_{d,u}(n)$	$T_m(N,n)$	$H_{i,3\text{-}B}$
VAL	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(nN)$
PRP	$O(\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(N\lg n)$
AKS	$O(\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(N\lg n)$
BAT	$O(\lg^2 n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg^2 n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(N\lg^2 n)$
CVN	$O(\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(N)$
RCF	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(nN)$
XPA	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(nN)$

(b) Summary table for Model 3-B based on Eqn. 3.2. Note that $N/k = n$.

	$T_{total,3\text{-}C}(N,k,n)$	$T_i(n)$	$T_{d,u}(n)$	$T_m(N,n)$	$H_{i,3\text{-}C}$
VAL	$O(\frac{N}{kn} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn^2)$
PRP	$O(\frac{N}{kn}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn\lg n)$
AKS	$O(\frac{N}{kn}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn\lg n)$
BAT	$O(\frac{N}{kn}\lg^2 n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg^2 n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn\lg^2 n)$
CVN	$O(\frac{N}{kn}\lg n + (\lg \frac{N}{n})(\lg\lg N))$	$O(\lg n)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn)$
RCF	$O(\frac{N}{kn} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn^2)$
XPA	$O(\frac{N}{kn} + (\lg \frac{N}{n})(\lg\lg N))$	$O(1)$	$O(1)$	$O((\lg \frac{N}{n})(\lg\lg N))$	$O(kn^2)$

(c) Summary table for Model 3-C based on Eqn. 3.3. Note that $N/kr = n$, and the hardware complexities can be reduced by increasing r. In fact, r captures the trade-off between time and hardware complexity as increasing r increases the sorting time complexity while reducing the hardware complexity.

Table 3.1: Time complexities and hardware complexity (H_i) of different sorting architectures and algorithms. (VAL:Valiant's generic model, PRP: Preparata's algorithm, BAT: Batcher's odd-even and bitonic network, RCF: Reconfigurable mesh, CVN: Parallel conventional algorithms (Merge sort[1], Quicksort[2, 3], Bucket sort[4]), XPA:1D-Crosspoint Array)



(a) (b)

Figure 3.9: $N = 10^{18}$. (a) Sorting times of algorithms before normalization but with the constants incorporated. In other words, it's plot of T_{total} in Table 3.1b with constants. This is a plot from pure speed perspective. (b) Sorting time of algorithms without the AKS.

greater than the constant for parallel merge sort$(15/4)$, the T_i term of AKS dominates T_{total} and results in a regular logarithmic graph, whereas the parallel merge does not and results in a linear graph. This can be indirectly proved in Fig. 3.10, where we plot T_i only. The figure shows the graph of AKS and parallel merge sort are parallel, indicating that the T_m term changes the tendency of the two graphs. Following the parallel merge sort, the 1D-Crosspoint Array and Valiant's algorithm are lowest, while the reconfigurable mesh is slightly higher by a constant factor. As the Valiant's algorithm is plotted without considering the processor allocation overhead of $O(\lg n)$, the 1D-Crosspoint Array can be said to be the fastest among the algorithms in Table 3.1b. Therefore, the 1D-Crosspoint Array is an appealing option for internal sorting engine when the sorting speed is the top priority and hardware complexity is less of a concern.

While T_{total} in the tables provides a basis for comparing the execution speed of listed sorting algorithms, it must be noted that the hardware complexity H_i is different for each algorithm. Particularly for Model 3-C, it is unreasonable to compare T_{total} without normalizing the time complexities with respect to H_i, because the model deliberately introduces a variable k to limit

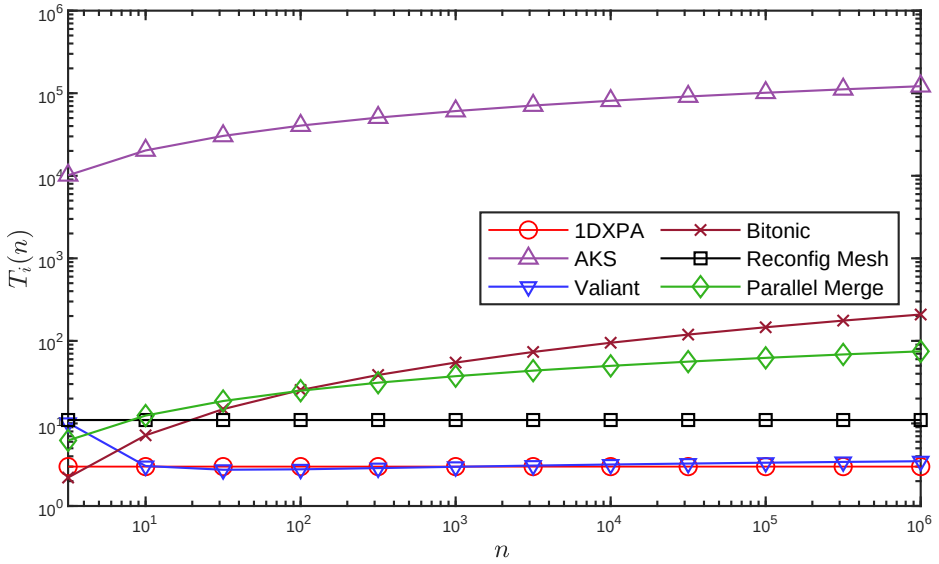


Figure 3.10: T_i in Table 3.1b with the constants incorporated.

H_i. Thus, as compared to Models 3-A and 3-B, where sorting algorithms are allowed to utilize as many PEs as they need, in Model 3-C, they cannot utilize more than H_i PEs. Therefore, we list the normalized time $T_{norm}(N,n) = T_{total}(N,n)/H_i(N,n)$ formulas in Table 3.2. To normalize, k was expressed in terms of H_i and n, and then substituted in T_{total} for each algorithm. T_{norm} is more meaningful to compare the time complexities of representative algorithms, because N and H_i are the same for all of them, and n can be considered as the only variable with which the efficacy of a sorting algorithm can be assessed. Overall, Valiant's sorting algorithm, Reconfigurable Mesh, 1-D Crosspoint Array all have the same time complexity. Parallel versions of merge, column and bucket sort algorithms have the smallest time complexity, followed by Preparata's and AKS sorting algorithms, and those followed by Batcher's sorting algorithms, when the number of PEs is kept the same across all representative sorting algorithms. Again, this is without taking into account the large constants in the complexity notations. The constant incorporated T_{norm} is plotted in Fig. 3.11, where the constants are calculated by the number of comparison operations. Once again, the AKS network is slowest due to the large constant factor caused by the depth

	$T_{norm}(N,n)$
VAL	$O\big(\frac{N}{H_i}n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
PRP	$O\big(\frac{N}{H_i}\lg^2 n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
AKS	$O\big(\frac{N}{H_i}\lg^2 n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
BAT	$O\big(\frac{N}{H_i}\lg^4 n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
CVN	$O\big(\frac{N}{H_i}\lg n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
RCF	$O\big(\frac{N}{H_i}n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$
XPA	$O\big(\frac{N}{H_i}n + \big(\lg\frac{N}{n}\big)(\lg\lg N)\big)$

Table 3.2: The normalized execution times for different sorting algorithms and networks based on Model 3-C.

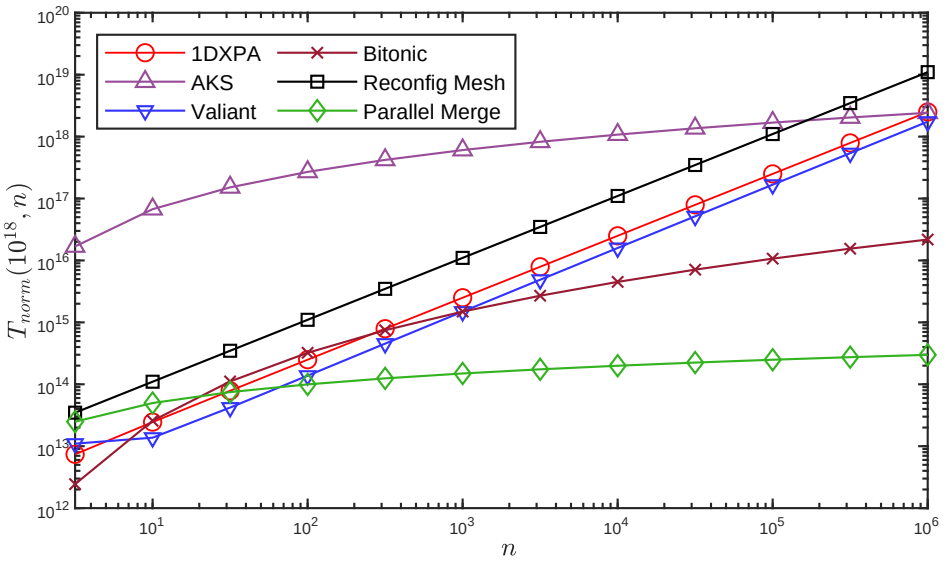


Figure 3.11: $N = 10^{18}$, $H_i = 10^6$. T_{norm} in Table 3.2 with the constants incorporated.

of the network, and the reconfigurable mesh follows next, which is slower than other constant time algorithms due to more number of stages the algorithm is composed of. 1D-Crosspoint Array is faster than the reconfiguration mesh by a constant factor, but the Valiant's algorithm, parallel merge sort, and bitonic sorting network are faster than the 1D-Crosspoint Array. The Valiant's algorithm is faster by a constant factor, but the algorithm do not consider the $O(\lg n)$ processor allocation overhead. In the case of Cole's parallel merge sort, it is faster than the 1D-Crosspoint Array by a factor of $n/\lg n$. It must be noted, however, that the algorithm assumes a PRAM(Parallel Random Access Machines) model which requires $O(1)$-time memory access and does not consider the communication time. In fact, the parallel merge sort and the bitonic sorting network was implemented on a PRAM in [91]. It was reported that the bitonic sorting network was faster in practice due to the communication time even though the T_i of parallel merge sort is lower by a factor of $\lg n$. Lastly, the bitonic sorting network is faster than 1D-Crosspoint Array by a factor of $n/\lg^4 n$, and is the second fastest for $n > 10^3$. Unlike the parallel merge sort, the bitonic sorting network is free from communication time issues because the network consists of dedicated paths for each individual input. Therefore, the Batcher's bitonic sorting network can be said to be the better option for internal sorting engine when the hardware complexity is also a priority along with the sorting speed, especially for $n > 10^3$. Although not the fastest, the 1D-Crosspoint Array can also be a competitive option for n in the lower range, $10^1 < n < 10^3$. and be a alternative choice among other constant time algorithms considering the minimal communication time complexity.

In conclusion, in this section, the 1D-Crosspoint Array was shown to be the fastest algorithm for the internal sorting engine when the sorting speed is the top priorty. Compared to other constant time sorting algorithms, the 1D-Crosspoint Array lead the competition owing to

the minimal communication time. In situations where the hardware complexity is also a priority, Batcher's bitonic sorting network was shown to be the optimal algorithm for the internal sorting engine. The 1D-Crosspoint Array was shown to be a competitive option for $10^1 < n < 10^3$, considering its minimal communication time.

Chapter 4: Parallel Matrix Multiplication

In this chapter, we use 1D-Crosspoint Array to obtain an optimal mapping of a classical matrix multiplication algorithm on to $O(n^2)$ PEs, thereby achieving an $O(n^2)$ speed-up.

4.1 Introduction

Matrix multiplication is one of the most widely used operations in many computational fields such as linear algebra[92, 93], machine learning[94], neural networks[95, 96], and graph processing[97, 98]. The time complexity of the naive matrix multiplication of two $n \times n$ matrices is $O(n^3)$. Although such a naive approach involves relatively simple multiplication-addition operations, $O(n^3)$ steps renders it ineffective, considering the wide and frequent usage of matrix multiplication. Consequently, developing faster matrix multiplication algorithms attracted much attention over the years. In 1969, Strassen introduced a matrix multiplication algorithm with $O(n^{2.8074})$ time complexity, which was based on partitioning each of the two matrices into four submatrices and then forming the product using seven rather than eight multiplications to reduce the overall time complexity of the matrix multiplication [99]. Since the work of Strassen, the serial matrix multiplication algorithm has been constantly improved(e.g. [100, 101, 102, 103, 104, 105, 106, 107]), to the point where the fastest algorithm has a time complexity of $O(n^{2.3728596})$ at the time of this writing[108]. Along another direction, the

parallelization of matrix multiplication algorithms has been extensively studied as well for reducing the time complexity of matrix multiplication. Since matrix multiplication can be divided into several multiplication of submatrices, it is suitable for divide-and-conquer type of parallelization. However, as it happens in other parallelization techniques the interprocessor communication time complexity dominates the computation time complexity when divide-and-conquer matrix multiplication algorithms are converted into parallel algorithms. One way of addressing this issue is to implement the underlying parallel architecture of matrix multiplication in a multi-dimensional structure, which inherently has more communication channels to help reduce communication time complexity. Cannon's algorithm was the first algorithm to assume a 2D grid layout of PEs[109]. By utilizing the horizontal and vertical channels by shifting the data within the grid, the algorithm parallelized matrix multiplication while optimizing the memory usage, albeit it required square input matrices and square grid of PEs. The algorithm was further developed in [23, 110] to cope with rectangular matrices and different processor decompositions and communication patterns. PUMMA(Parallel Universal Matrix Multiplication Algorithm)[111] was proposed as an algorithm that generalized previous works to transposed matrices and different data layouts. In 1995, Van De Geijn and Watts presented SUMMA(Scalable Universal Matrix Multiplication Algorithm), which minimized the communication time and memory usage by pipelining communication and computation steps[112]. SUMMA is one of the most widely implemented matrix multiplication algorithm, such as in ScaLAPACK library[113].

The idea of "3D" algorithm that runs on cubic layout of PEs was first introduced in [28], which had factor of $O(n^{1/6})$ less communication time overhead compared to 2D algorithms, but required $O(n^{1/3})$ more memory space. In 2011, algorithms that run on 2D layouts of PEs with extra memory, commonly referred to as '2.5D' algorithms, were introduced[29, 30]. To be

more specific, these types of algorithms assume multiple 2D grids stacked vertically in order to partially mimic the parallelism of a 3D layout. However, although 2.5D algorithms were optimal for square matrices, they performed poorly when matrices were significantly long or wide[114].

In this chapter, a parallel matrix multiplication algorithm for the 1D-Crosspoint Array is introduced and its performance is analyzed. We already established that the locality of the inter-processor connections in a 1D-Crosspoint Array leads to fast sorting and searching algorithms in the earlier chapter. Here, it will be shown that the 1D-Crosspoint Array provides an effective communication structure for a parallel implementation of matrix multiplication algorithms as well. The rest of this chapter is organized as follows. In Section 4.2, the parallel matrix multiplication algorithm devised for the 1D-Crosspoint Array will be proposed. In Section 4.3, the time complexity and the communication cost of the matrix multiplication algorithm will be analyzed.

4.2 Parallel Matrix Multiplication Algorithm

The new parallel matrix multiplication algorithm that runs on the 1D-Crosspoint Array is shown in Algorithm 3. Before the start of the algorithm, the two multiplicand matrices are assumed to be distributed to PEs. To be more specific, a_i, the i-th row vector, is assumed to be distributed to class i PEs and likewise b_i, the i-th column vector of B, is assumed to be distributed to class i PEs. In `parFor` loop in line 1, each PE calculates the dot product of a_i and b_i it was distributed with. Then communicate a_i and b_i with its direct neighbor PEs and calculate the dot product of a_i, b_i and the vectors from the neighbor PE. An illustration of Algorithm 3 for the case of $n = 5$ is shown in Fig. 4.1. Fig. 4.1(a) shows how A and B is partitioned into row and column vectors before being distributed to replicates, and Fig. 4.1(b) show which element of C is being calculated in which

replicate in which line of Algorithm 3. Once the `parFor` loop is complete, every element of the result matrix C is guaranteed to be calculated, which is proved in the following proposition.

Algorithm 3: Matrix Multiplication Algorithm

Input: A, B, $n \times n$ matrices

Output: C, $n \times n$ matrix

// P_i is a PE belonging to class i
// a_i is the i-th row vector of A
// b_i is the i-th column vector of B
// $c_{i,j}$ is the element of C at i-th row and j-th column
// a_i and b_i are assumed to be distributed to class i PEs

1 **parFor** P_i, $0 \leq i \leq n-1$ **do**
2 $\quad c_{i,i} = a_i \cdot b_i$
3 $\quad$ Send a_i and b_i to the right-hand neighbor. (Receive a_L and b_L from the left-hand neighbor.)
4 $\quad c_{i,L} = a_i \cdot b_L$
5 $\quad c_{L,i} = a_L \cdot b_i$

Proposition 12. When the `parFor` loop in line 1 of Algorithm 3 is complete, every element of C is calculated.

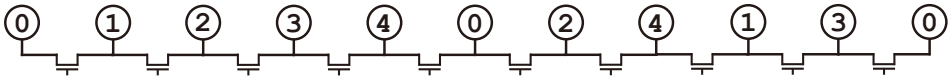
Proof. The diagonal elements of C, or $c_{i,i} = a_i \cdot b_i$, is calculated in line 2 with the distributed data. To calculate the non-diagonal elements, we make use of the complete adjacency. A non-diagonal element of the result matrix $c_{i,j}$ is a dot product of i-th row vector of A and j-th column vector of B where $i \neq j$. In other words, it is a dot product of vectors that are distributed to different classes. Therefore, due to the complete adjacency, if each adjacent pair on the 1D-Crosspoint Array calculate the dot product of vectors in both PE's local memory, every element of C will be calculated. □

In line 3 of Algorithm 3, the PEs uniformly send the row and column vectors to the right-hand side neighbor. We note that this right shift of row and column vectors may lead to a possible

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} \end{bmatrix}$$

(a) Row and column vectors of A and B that is distributed to replicates.

Class of replicates	0	1	2	3	4	0	2	4	1	3	0
Initially distributed a_i and b_i	$[a_1]$ $[b_1]$	$[a_2]$ $[b_2]$	$[a_3]$ $[b_3]$	$[a_4]$ $[b_4]$	$[a_5]$ $[b_5]$	$[a_1]$ $[b_1]$	$[a_3]$ $[b_3]$	$[a_5]$ $[b_5]$	$[a_2]$ $[b_2]$	$[a_4]$ $[b_4]$	$[a_1]$ $[b_1]$
Communicated vectors in line 3		$[a_1]$ $[b_1]$	$[a_2]$ $[b_2]$	$[a_3]$ $[b_3]$	$[a_4]$ $[b_4]$	$[a_5]$ $[b_5]$	$[a_1]$ $[b_1]$	$[a_3]$ $[b_3]$	$[a_5]$ $[b_5]$	$[a_2]$ $[b_2]$	$[a_4]$ $[b_4]$
$c_{i,i}$ calculated in line 2	c_{11}	c_{22}	c_{33}	c_{44}	c_{55}	c_{11}	c_{33}	c_{55}	c_{22}	c_{44}	c_{11}
$c_{i,L}$ and $c_{L,i}$ calculated in lines 4 and 5		c_{21}, c_{12}	c_{32}, c_{23}	c_{43}, c_{34}	c_{54}, c_{45}	c_{15}, c_{51}	c_{31}, c_{13}	c_{53}, c_{35}	c_{25}, c_{52}	c_{42}, c_{24}	c_{14}, c_{41}



(b) Detailed demonstration of Algorithm 3 for each line.

Figure 4.1: Illustration of Algorithm 3 for $n = 5$ case.

deadlock if the PEs are following a message passing type protocol and wait to receive the vectors from the left-hand neighbor. However, the actual form of communication is not the crucial to the correct execution the algorithm as there are numerous other methods that can resolve such deadlock issues (such as deciding based on the class number), and also because such problems can be often solved by the communication protocol itself. The key aspect of the communication happening in line 3 is that it only happens between direct neighboring PEs and that the number of communications required is constant, or $O(1)$.

Algorithm 3 is complete once every element of C is calculated. If the entire matrix C must be collected in a local memory or a shared memory, additional steps is needed. Likewise, the a_i and b_i, which are assumed to be distributed to PEs beforehand would also require additional steps for they were to be done within the algorithm. However, because the matrix multiplication is mostly a subroutine in a larger computation, the multiplicands, a_i and b_i are likely to be in-

termediate results from that subroutine, will already be residing in local memories of PEs. The elements of resultant matrix C are also likely to be an input for the next subroutine and reaady for execution in local memories of PEs. It is therefore reasonable to assume the distribution of input matrices and collection of result matrices is not necessary[115].

4.3 Parallel Matrix Multiplication Performance Analysis

Let us analyze the run time of the matrix multiplication on the 1D-Crosspoint Array. Each PE must compute three dot products of length n vectors. Each dot product includes n multiplications, so the computation time is $O(n)$. The communication in line 3 sends or receives two length n vectors, thus the communication time would be $O(n)$, assuming one element can be communicated at a time. Since the `parFor` loop is carried out in parallel, the time complexity of Algorithm 3 is $O(n)$.

In addition to the arithmetic time complexity, another complexity that must be examined is the communication time complexity. Many of the recent studies of parallel matrix multiplication focus on reducing the communication time complexity by optimizing the algorithm to better fit the data communication pattern(e.g., the dependency between operations or stages) of the matrix multiplication algorithm. More precisely, studies aim to lower the I/O complexity(Q) and latency(L), where Q refers to the total amount of data that is sent or received per PE, and L denotes the number of messages that is sent or received per PE. Q is expressed in number of words, since each element of a matrix, in general, is assumed to be the size of a word of the system. Also, since it is reasonable to assume that the maximum size of each message can only be as large as the size of the local memory S, the value of L can be theoretically calculated as $L =$

Q/S, albeit the attainability of such latency depends on the structure or algorithm[116]. These two metrics naturally became the two of the most common metrics used to compare different parallel matrix multiplication algorithms, and the lower bound on Q and L have become another focus in parallel matrix multiplication [114, 117, 118].

To provide a reference point for the communication time complexity of matrix multiplication on a 1D-Crosspoint Array, a survey of the lower bound on Q and L is provided below. Since the classic $O(n^3)$ algorithm and the Strassen matrix multiplication algorithm have different data communication requirements, the studies on these lower bounds are generally categorized into two types, namely classic algorithm and Strassen-like algorithms. The lower bound for parallel classical $O(n^3)$ multiplication algorithm has been substantially studied over the years(e.g., [114, 115, 119, 120]). One of the early contributions on the communication time complexity of classical $O(n^3)$ multiplication algorithms is due to Irony et al., who devised a lower bound on Q, assuming that input distribution and output collection is not required[115]. They reported that, for multiplying two $n \times n$ matrices with P PEs, each with local memory of size S words, the I/O complexity is

$$Q_{Irony} = \frac{n^3}{2\sqrt{2}PS^{1/2}} - S < Q, \tag{4.1}$$

and from the relation $L = Q/S$, we have

$$L_{Irony} = \frac{n^3}{2\sqrt{2}PS^{3/2}} - 1 < L. \tag{4.2}$$

However, both lower bounds degrade to zero when $S \geq n^2/(2P^{2/3})$. Since the 1D-Crosspoint

Array is composed of $n(n-1)/2+1$ PEs, and since each PE must store 4 vectors(i.e., a_i, b_i, a_L, b_L) and three scalar data(i.e., $c_{i,i}, c_{i,L}, c_{L,i}$), let us assume $P = n^2$ and $S = 5n$. In that case, $S = 5n \geq n^2/(2n^{2\times 2/3}) = n^{2/3}/2$, and thus the lower bound does not provide practical information on the Q and L of the 1D-Crosspoint Array. The communication lower bound presented by Kwasniewski et al. in [118] can be an alternative, although it must be noted that these lower bounds are assuming that the distribution of the input and the collection of the output is also included in the algorithm. Nevertheless, they derived a lower bound for I/O complexity and latency for matrix multiplication,

$$Q_{classic} \geq \min\left\{\frac{2n^3}{P\sqrt{S}} + S,\ 3\left(\frac{n^2}{P^{\frac{2}{3}}}\right)\right\}, \tag{4.3}$$

$$L_{classic} \geq \frac{4ab}{S-a^2}\lg\left(\frac{n}{a}\right),\ \text{where } a = \min\left\{\sqrt{S}, \frac{n}{P^{\frac{1}{3}}}\right\},\ b = \max\left\{\frac{n^3}{PS}, \frac{n}{P^{\frac{1}{3}}}\right\}, \tag{4.4}$$

and devised a matching parallel algorithm called COSMA(Communication Optimal S-partition-based Matrix multiplication Algorithm). COSMA first arrange an optimal sequential schedule of multiplication of each element, then convert it to a parallel schedule while preserving the optimal communication time.

On the other hand, the family of matrix multiplication algorithms that stem from Strassen's algorithm[99], which will be referred to as *Strassen-like* algorithms, has also been studied considerably. This line of studies focus on optimizing the algorithm to fit the unique data pattern of the Strassen-like algorithms[116, 117, 121, 122, 123, 124]. It was proven that the lower bound

of I/O complexity [117] and latency [114] for Strassen-like algorithms is

$$Q_{Strassen} = \Omega\left(\left(\frac{n}{\sqrt{S}}\right)^{\omega_0} \cdot \frac{S}{P}\right) \tag{4.5}$$

$$L_{Strassen} = \Omega\left(\frac{n^3}{PS^{1.5}} + 1\right) \tag{4.6}$$

when $\omega_0 < 3$.

Given the fact that Algorithm 3 is a variant of the classic algorithm, we use Eqns. 4.3 and 4.4 to calculate the possible lower bound of Q and L for 1D-Crosspoint Array. With $P = n^2$ and $S = 5n$, the lower bound of Q and L for 1D-Crosspoint Array is

$$Q_{bound} = 3n^{2/3} \tag{4.7}$$

$$L_{bound} = (8 \lg n)/(15n^{1/3} - 3). \tag{4.8}$$

It must be noted that the $L_{bound} < 1$ for any n and given that the latency must be an integer since it is the number of messages, L_{bound} is 1 for any n.

Let us now analyze the I/O complexity and the latency of Algorithm 3. In line 3, assuming each element of the matrices is a word, each PE sends $2n$ words to the right and receives $2n$ words from the left. In addition, each PE is distributed two vectors(i.e., a_i, b_i), and each PE will have 3 elements of C to send to the shared memory. In total, the data that each PE must send or receive is

$$Q_{XPA} = 4n + 2n + 3 = 6n + 3.$$

The input distribution and result collection each can be done in one message, and the line 3 need

two messages(left and right), the latency for 1D-Crosspoint Array is

$$L_{XPA} = 4.$$

It can be seen that the I/O complexity of the 1D-Crosspoint Array is greater than the lower bound by a factor of $n^{1/3}$, which indicates the Algorithm 3 needs to communicate $n^{1/3}$ more words compared to the COSMA. However, it must be noted that the I/O complexity refers to the total amount of data communicated and does not consider the communication contention among PEs. This is partially due to the fact that most algorithms assume a general parallel computing system, so that the algorithm can be versatile to be implemented on different systems. The downside of such versatility is that the lower bound is not guaranteed in practice, but it depends on the details of the network of the system[116]. The 1D-Crosspoint Array, on the other hand, guarantees a contention-free communication and the communication pattern is simple and predictable. Predictable communication can be a benefit from programming standpoint since it makes the job of optimizing synchronization across PEs less challenging. In addition, because there is no communication contention in the crosspoints, and also because the communication in line 3 can be completed in parallel, we can effectively estimate that the line 3 can be completed in $O(Q/\alpha)$, where α denotes the bandwidth of the crosspoints. In the case of latency, given that L_{bound} is 1 for any n, the latency of the 1D-Crosspoint Array $L_{XPA} = 4$ is close to the theoretical optimum as it only differs by a constant factor.

The matrix multiplication on 1D-Crosspoint Array is adaptable such that it can provide optimal algorithm for different number of PEs. Moreover, while Algorithm 3 is a parallel version of naive $O(n^3)$ algorithm, other serial or parallel algorithms can also be implemented on the

1D-Crosspoint Array with minimal effort, since the design can be apt for different divide-and-conquer type algorithms. For example, the parallel bit-serial matrix multiplication algorithm such as the one explained in [125] can be implemented in 1D-Crosspoint Array, where each bit multiplications is done by one PEs. Strassen-like algorithms can also be implemented in 1D-Crosspoint Array as well, by dividing the inputs submatrices instead of vectors and then computing the submatrix multiplication using Strassen's algorithm in parallel. In this case, the communication cost will not be optimal and will require optimization. Finally, as in external sorting, large matrices can be multiplied by partitioning the multiplicand matrices and performing submatrix multiplications on a 1D-Crosspoint Array and swapping them between cloud servers and the 1D-Crosspoint Array. For example, the parallel bit-serial matrix multiplication algorithm such as the one explained in [125] can be implemented in 1D-Crosspoint Array, where each bit multiplications is done by one PEs. Strassen-like algorithms can also be implemented in 1D-Crosspoint Array as well, by dividing the inputs submatrices instead of vectors and then computing the submatrix multiplication using Strassen's algorithm in parallel. In this case, the communication cost will not be optimal and will require optimization. Finally, as in external sorting, large matrices can be multiplied by partitioning the multiplicand matrices and performing submatrix multiplications on a 1D-Crosspoint Array and swapping them between cloud servers and the 1D-Crosspoint Array. In particular, let us assume assume the matrix is of size $N \times N$ and there are k number of 1D-Crosspoint Array composed with $N/\sqrt{k} = n$ classes. Further assuming Model 3-B, the $N \times N$ matrix can be divided into submatrices of size $N/\sqrt{k} \times N/\sqrt{k}$ and each 1D-Crosspoint Array can carry out one submatrix multiplication in $O(1)$ time. In this case, the time complexity is $T_{MM} = T_i + T_{d,u} + T_{add} = O(1) + T_{d,u} + T_{add}$, where $T_i = O(1)$ refers to the submatrix multiplication time complexity, $T_{d,u}$ refers the download/upload time, and T_{add}

refers to time complexity of adding the result of the submatrix multiplication result. If Model 3-C is assumed with each 1D-Crosspoint Array consisting of $n = N/kr$ classes, the submatrix size is reduced to $N/\sqrt{kr} \times N/\sqrt{kr}$, each 1D-Crosspoint Array would carry out r number of submatrix multiplications, and the time complexity would be $T_{MM} = \frac{N}{kn}T_i + T_{d,u} + T_{add} = O(\frac{N}{kn} + T_{d,u} + T_{add})$.

Chapter 5: Parallel Triangle Counting

In this chapter, we describe an effective mapping of a triangle counting graph algorithm with n vertices to a 1D-Crosspoint Array with $O(n^2)$ PEs.

5.1 Introduction

With the rapid growth of the Internet technologies, data from various domains, including social, communication, co-authorship networks[126], business and finance[127], biological networks[128] are modeled as graphs to capture relations and structures within the data[129]. Among many metrics that were introduced to analyze such graphs, one of the most important and widely used one is the transitivity, i.e., the ratio of the number of triangles to the number of triples (set of three vertices) in a graph, and clustering coefficient, i.e., the ratio of the triangles and triples that include a particular vertex. It has been shown that the two metrics can be used to detect malicious activities[130], optimize query planning in databases[131], find fake accounts in social networks[132], and discover communities[133]. Since triangle counting plays a major role in computing the two metrics, it has attracted much attention in recent years. Although the fact that the problem of counting triangles is algorithmically not too hard, it is the sheer volume of the data that makes the problem time consuming and draws scientists from diverse communities into the investigation of triangle and similar counting problems. One summary and categorization of

the numerous triangle counting algorithms can be found in [134], and a few categories of triangle counting will be introduced in this section. Before diving into each categorie in detail, we first provide a formal definition of the problem of triangle counting below.

Definition 3. ***Triangle Counting*.** The problem of triangle counting is counting the number of triangles in a graph denoted by $G(V, E)$, where V is the set of vertices and E is the set of edges. The number of vertices and edges are denoted by $|V|$ and $|E|$ respectively. Each vertex is uniquely identified by a number between 1 and $|V|$. We assume G is a simple, connected, and undirected graph. Since G is simple, there exist at most one edge between a pair of vertices u and v, which is denoted as (u, v), where $u < v$. For a vertex u, adj(u) denotes the set of u's neighboring vertices, and $d(u)$ denote the degree of u. □

The adjacency matrix, A of a graph G in which the edge between vertices u and v is indicated by 1, as shown below:

$$x_{u,v} = \begin{cases} 1, & \text{if } (u, v) \in E \\ 0, & \text{if } (u, v) \notin E \end{cases}$$

In the rest of this chapter, we present a survey of related literature on parallel triangle counting and a triangle counting algorithm for the 1D-Crosspoint Array and its performance analysis.

5.2 Related Works

The first approach to counting triangles is naturally brute force that checks all of $\binom{|V|}{3}$ triples and see which ones form a triangle. Such an approach would take $O(|V|^3)$ time which is undoubtedly

inefficient. A more efficient way is to go through every vertex in the graph and check if either of the two neighboring vertices are neighbors themselves. In other words, it checks every length-2 path, instead of every $\binom{|V|}{3}$ triple, to see if it is a triangle. This type of triangle counting is commonly referred to as a NodeIterator algorithm and is formalized in Algorithm 4.

A similar way is to iterate through every edge in the graph and check if the two vertices that the edge consists of have a common neighboring vertex. This type is commonly referred to as the EdgeIterator algorithm and is described in Algorithm 5. The problem with both NodeIterator and

Algorithm 4: NodeIterator

Input: $G(V, E)$

Output: triangle $count$, triangle $list$

```
1 count = 0
2 for v ∈ V do
3     count_v = 0
4     for each pair of distinct u, w ∈ adj(v) do
5         if (u, w) ∈ E then
6             count_v = count_v + 1
7             append (u, v, w) to list
8     count = count + count_v
9 return count, list
```

EdgeIterator is that each triangle can be counted more than once. For example, triangle (u, v, w) can be counted as any other permutation of the three vertices, such as (u, w, v) or (v, u, w). Such duplicate count can be avoided by applying a total ordering to the vertices and applying the order as a condition to the triples being checked by the NodeIterator and EdgeIterator algorithms. Thus, if we check only the triples (u, v, w) with the ordering $u \prec v \prec w$, each triple is then counted exactly once. One of the frequently used ordering schemes is the degree of the vertices, and some examples of such improved algorithms can be found in [135] and [136]. The NodeIterator and

Algorithm 5: EdgeIterator

Input: $G(V, E)$

Output: triangle $count$, triangle $list$

```
count = 0
for (u, v) ∈ E do
    W_{u,v} = adj(u) ∩ adj(v)
    count = count + |W_{u,v}|
    for w ∈ W_{u,v} do
        append (u, v, w) to list
return count, list
```

EdgeIterator algorithms can be thought of as a *triangle listing* algorithm, as it is the only type that can *list* which vertices make triangles in the graph in addition to counting the number of triangles. The exact list of triangles can be valuable information for tasks such as discovering communities[137] or filtering spam[130]. It was proven in [138] that these improved triangle listing algorithms are optimal and have time complexity of $O(|E|^{3/2})$. The following lemma proves the time complexity of triangle listing is $O(|E|^{3/2})$.

Lemma 1. [138, Corollary,page.23] The lower bound time complexity of a triangle listing algorithm is $\Omega(|E|^{3/2})$.

Proof. Consider the worst case for such an algorithm, which occurs when there are as many length-2 paths as possible in the graph. The number of such paths is equal to the number of triangles. The maximum number of triangles in a simple graph in terms of $|V|$ is $\binom{|V|}{3} = O(|V|^3)$. Since the maximum number of edges is $\binom{|V|}{2} = |V|(|V|-1)/2$, we have $|E| \leq |V|(|V|-1)/2 \leq |V|^2$, and in turn, we have $\sqrt{|E|} \leq |V|$. Thus, the maximum number of triangles in terms of $|E|$ is $O(\sqrt{|E|}^3) = O(|E|^{3/2})$. The triangle listing algorithm must iterate through every length-2 path in the graph, and hence the lower bound on the time complexity of the triangle listing algorithm

is $\Omega(|E|^{3/2})$. $\square$

It must be noted that the equality in $|E| \leq |V|(|V|-1)/2$ is an extreme case of the graph being a $|V|$-clique, where every vertex is connected to every other vertex. In general, the relation between $|E|$ and $|V|$ is $|E| \ll |V|^2$(e.g. see, [139], or the data used in [135, 136, 140, 141, 142]).

Another type of exact counting algorithm employs linear algebra using matrix multiplication[143, 144, 145, 146]. At the base of this type of algorithm is the fact that if A is the adjacency matrix, the diagonal elements of the A^3, say $A^3(i,i)$, would indicate a length-3 path(a triangle) that starts and ends at vertex i. This method is becoming more popular because it can utilize the linear algebra kernels that most machines already have and are very much optimized for the system[146]. One of the earliest literatures to report this type of triangle counting was [143], where Alon et al. proved that if a matrix multiplication algorithm with time complexity of $O(n^\omega)$ was used, the triangle counting could be completed in $O(|E|^{2\omega/(\omega+1)})$ time. However, since summing up the diagonal elements of A^3 would count each triangle ${}_3P_3 = 6$ times repeatedly, there exists room for optimization by eliminating duplicate counting. In [144], the algorithm was optimized to count each triangle twice repeatedly by computing $C = (L \cdot U) \circ A$ instead of A^3, where L and U denote the lower and upper triangular matrix respectively, and $\circ$ denotes the element-wise multiplication. The elements of result matrix $C(i,k)$ would indicate that (i,j,k) is a triangle for some j, so every element of the result matrix needed to be summed up to obtain the number of triangles. Also, since such variant counted each triangle twice, in [145], the algorithm was further optimized to $(L \cdot L) \circ L$, such that each triangle was counted exactly once.

Along another direction, approximate triangle counting algorithms gained popularity in applications where the exact triangle count is not needed, because the $O(|E|^{3/2})$ time complexity

can be excessive due to the vast scale of the data. Graph sparsification[147, 148, 149] and triple sampling[150, 151, 152] are some examples of approximate triangle counting. The main idea behind the approximate counting is to project the number of triangles of the entire data based on the number of triangles counted in a portion of the data. Graph sparcification method probabilistically deletes a subset of edges in the graph and then counts the triangle in the sparcified graph. The triple sampling method samples length-2 paths and counts triangles among the sampled length-2 paths. In both cases, once the number of triangles is counted in the subgraph, the triangle count of the original graph is projected based on the probability that was used in the sparcification or sampling stage. The main challenge of the approximate counting algorithms is to pick as probabilistically unbiased sample as possible to increase the accuracy of the algorithm. In addition, since big data is overly large for a system memory to hold at once, access to random vertex or edges in any part of the graph may be impossible or intolerably inefficient, making the triangle counting algorithms mentioned above impractical. As an alternative, triangle counting algorithms, employin distributed memory systems have also been explored. Note that since optimal sequential algorithms for counting triangles already exist, the main focus of these research efforts has been on either dividing the graph to guarantee the correctness of triangle counting or evenly distributing the workload[135, 140, 141, 153, 154].

The task of dividing the graph is not as simple as dividing the vertex set V into equal sized subsets. In order to avoid excessive communication in distributed triangle counting, the ideal procedure is to let each PE count the triangles locally and then collect the results at the end. Therefore, a graph dividing scheme, where triangles can be counted independently while not missing any triangles is desired. For this reason, simply dividing V into equal sized non-overlapping subsets is inadequate, because the triangles also can be divided at the same time,

necessitating communication in the triangle counting step. One of the earliest methods that was proposed to properly divide the graph was the GraphPartition(GP) algorithm[140] for MapReduce framework. The GP algorithm first partitions V into equal sized subsets, and then creates bigger subsets by combining every possible choice of three subsets. Finally, the algorithm counts triangles within each of the bigger subsets, which guarantees no missing triangles in the graph. Although this method indeed partitions the graph while not missing any triangles, the size of the intermediate data created by the subgraphs is too large and it degrades the performance of the algorithm[135]. To address this problem, many studies focused on optimizing the GP algorithm for memory and communication cost[153, 155, 156, 157, 158]. These studies employ marginally different methods, and we will describe the approach in Hoang et al. called distTC[158] as an example of graph dividing. In [158], V is first divided into non-overlapping subsets. Then, for a subset V_i and its induced edge set E_i, any vertex that is in adj(V_i) but is not in V_i is identified as a *vertex proxy*, and any edges between the vertex proxies are identified as an *edge proxy*. The vertex proxies and edge proxies are added to each of the subsets, then are distributed to each of the PEs. With the vertex proxies and edge proxies added, every triangle is included in at least one subset, and thus counting the triangles locally and then collecting the results will count all the triangles in the entire graph correctly.

Similarly, distributing the graph by dividing the set of edges were studied in [156, 157]. Although these algorithms divide the set of edges into subsets, the algorithm eventually induces the vertex set and neighbor set from the subset of edges and count the triangles within the induced vertex set. In 2013, an algorithm focused on I/O-efficiency was introduced in [156] for single CPU environment, and then a parallelized and optimized version named PDTL was introduced[157]. It was proved the communication cost of the PDTL is $\Theta(n + |E| + |t|)$, where

n is the number of PEs, and $|t|$ is the number of found triangles.

The task of evenly distributing the workload is another area of research in distributed triangle counting. Because all triangle listing algorithms search the neighbor set adj(v) of every vertex v, the degree of the vertices, $d(v)$, is the major factor that is directly related to the work carried out by the PEs. The distribution of $d(v)$, $v \in V$, for most of the real-life large graphs is known to follow the power law[159]. The power law states that the degree of vertices of a large graph is skewed, and certain vertices in any graph will almost always have a degree linearly proportional to the size of the graph while most of the others have very small degree[140]. Thus, to evenly distribute the workload, the degree of each vertex must be accounted for. A basic framework for such distribution was proposed in [135]. Several cost functions for estimating the workload for each vertex were introduced. These extend from simple vertex degree-based cost functions more complicated, but also more accurate aggregate degree-based functions such as $\sum_{u \in \text{adj}(v)}(\hat{d}(v) + \hat{d}(u))$, where $\hat{d}(v)$ refers to the number of neighbors that satisfy the total ordering condition, $v \prec u$. Naturally, the more accurate the estimations were, the more time it took to estimate the cost.

5.3 Parallel Triangle Counting on the 1D-Crosspoint Array

For triangle counting on the 1D-Crosspoint Array, two different solutions are proposed. If list of the triangles is not needed but the exact count of triangles is needed, then the matrix multiplication algorithm (Algorithm 3) in Section 4.2 is the optimal triangle counting algorithm for 1D-Crosspoint Array. This is because such an algorithm is designed to minimize communications between PEs, and the 1D-Crosspoint Array offers an ideal architecture for minimizing commu-

nications among PEs based on its complete adjacency. However, if the list of triangles is desired, then a variant of the NodeIterator or the EdgeIterator must be used. Algorithm 6 given below is a type of NodeIterator algorithm combined with the partitioning method of the distTC[158]. As mentioned above, the distTC partitions the graph by identifying the vertex proxies and edge proxies and including them to the respective subsets in the preprocessing step. Unlike the distTC which handles the preprocessing in a single CPU, in Algorithm 6, the preprocessing is modified such that it is processed in parallel in each of the PEs. To be more precise, the non-overlapping subset of V is distributed in line 1 to each PE by the P_{master}, which is the *master* PE that initiates the algorithm and collects the result. Then, the vertex and edge proxies are identified and added to the subsets in parallel by the `parFor` loop in line 2. The triangles are then listed locally in line 5, then the result is returned to P_{master}. Due to the complete adjacency, our approach accomplishes the same result as the distTC, while reducing the communication contention between the PEs. However, it must be noted that the total amount of communication is not different because the reduced amount of communication happens between PEs.

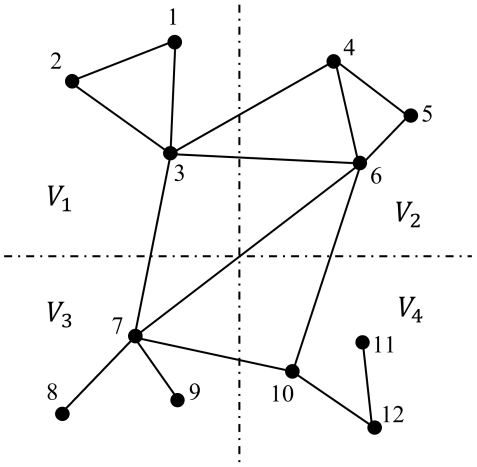
Algorithm 6: tc algorithm

Input: V, E

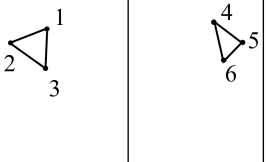
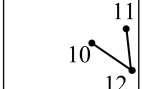
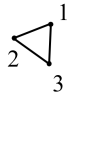
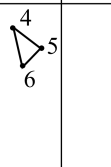
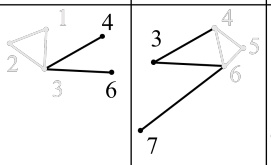
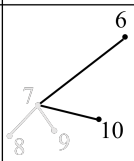
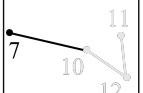
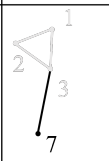
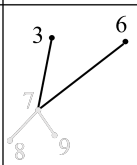
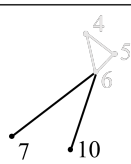
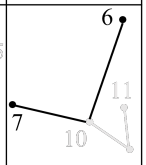
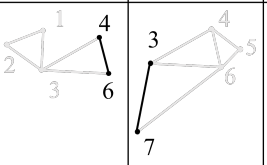
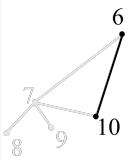
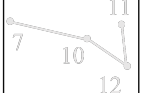
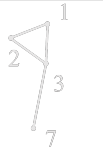
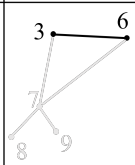
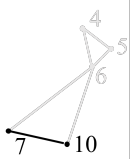
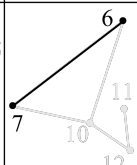
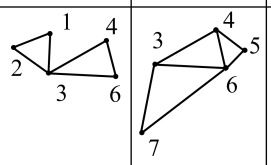
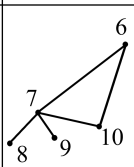
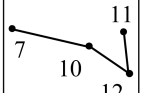
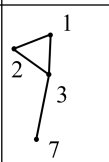
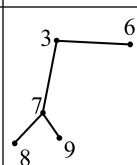
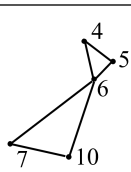
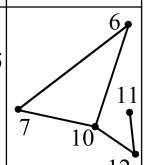
Output: t, the list of triangles

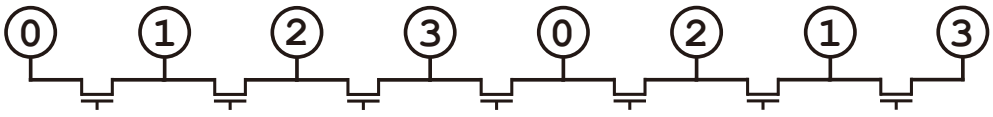
// P_i is a PE belonging to class i
// P_{master} is the master PE
// V_i, $0 \leq i \leq n-1$, is non-overlapping subset of V

1 Distribute V_i to P_i.
2 **parFor** P_i, $0 \leq i \leq n-1$ **do**
3 Identify the vertex proxies with direct neighbor PEs
4 Identify the edge proxies.
5 List triangles using Algorithm 5
6 Return t to the P_{master}.



(a) Example graph of $G(11, 16)$.

Class	0	1	2	3	0	2	1	3
V_i distributed to P_i (Line 1)								
Vertex proxies (Line 3)								
Edge proxies (Line 4)								
Subgraph each PE will search for triangles								
Found triangles	(1,2,3) (3,4,6)	(3,4,6) (4,5,6) (3,6,7)	(6,7,10)		(1,2,3)		(4,5,6) (6,7,10)	(6,7,10)



(b) Illustration of Lines 1 through 4 for the above graph with $n = 4$.

Figure 5.1: Illustration of Algorithm 6 for an example graph $G(11, 16)$ for $n = 4$. The solid black lines and numbers refers to the vertex and edge proxies that are identified by each PE.

A more detailed analysis on the communication time complexity is as follows. Assume the graph is stored in the same format as the distTC and PDTL, Compressed Sparse Row(CSR) format[160], which is a representation that stores each row of the adjacency matrix separately. Each row is compressed by omitting zeros, resulting in space complexity of $O(|V|+|E|)$. Assuming $O(n^2)$ number of PEs, the distribution in line 1 causes $O(n(|V|+|E|))$ communication time as each subset is sent to $n(n-1)/2$ PEs and each adjacency matrix is of size $O((|V|+|E|)/n)$. Similarly, if the total numbers of vertex proxies and edge proxies are denoted as $|V_{prx}|$ and $|E_{prx}|$ respectively, then the communication occurring in the `parFor` for identifying the proxies is $O(n(|V_{prx}|+|E_{prx}|))$. The communication needed for collection of the triangle list can be expressed as $O(n|t|)$, where t is the list of triangles, because the triangles that are completely within a subset will be counted repeatedly by all replicate PEs of the same class. Therefore, the total communication time can be expressed as $O(n(|V|+|E|)+n(|V_{prx}|+|E_{prx}|)+n|t|)$. Not many studies have reported the communication time. A recent one that does is the PDTL[157], where it was reported that the communication time is $\Theta(n^2+|E|+|t|)$ when using n^2 PEs. Because the maximum possible $|E|$ is $\binom{|V|}{2} = O(|V|^2)$ only in the extreme case of a $|V|$-clique, and that generally $|V|, |E| > n$, the PDTL can be said to have lower communication time compared to Algorithm 6 in general. However, it is worth noting that even though the communication time is higher, due to the n^2 term, the fact that 1D-Crosspoint Array can manage the communication of the proxy effortlessly and without any contentions, the communication time difference would get smaller for larger n. Lastly, as in the case of external sorting, large graphs can be divided subgraphs which is distributed from the could servers to different internal engines which would count or list the triangles and sent back to the cloud servers. If we assume Model 3.2, for a graph $G(E,V)$, when there are k number of 1D-Crosspoint Arrays each of size

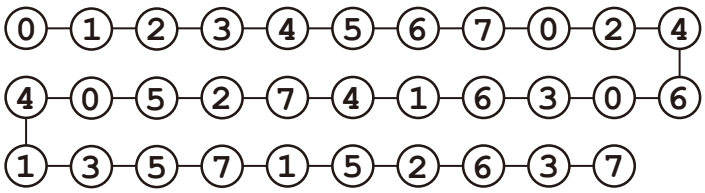
$n = N/k$, the triangle counting time complexity would be $T_{total,TC} = T_{i,TC} + T_{d,u} + T_{comb}$, where $T_{i,TC} = O(n(|V|/k + |E|/k) + n(|V_{prx}|/k + |E_{prx}|/k) + n|t/k|)$, $T_{d,u}$ refers to download/upload time, and T_{comb} refers to the time complexity of the server to combine the triangle count and the list. When Model 3-C is assumed with each 1D-Crosspoint Array is composed of $n = N/kr$ classes, then the total time complexity would become $T_{total,TC} = \frac{N}{kn} T_{i,TC} + T_{d,u} + T_{comb}$.

Although the performance of triangle counting on 1D-Crosspoint Array did not exceed any of the existing algorithms, it is worth noting that it showed the 1D-Crosspoint Array can be implemented to solve a graph problem. Utilizing the high parallelism and the minimal communication time, we anticipate the potential of the 1D-Crosspoint Array in solving other graph problems as well.

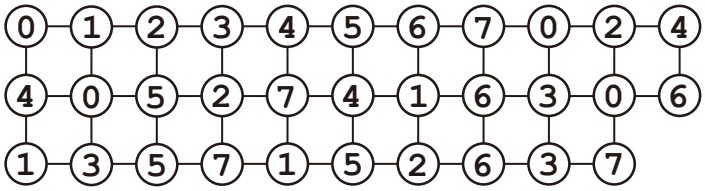
Chapter 6: 2D- and 3D-Crosspoint Array

6.1 2D-Crosspoint Array

Although the one dimensional(1D) topology is the most simple structure, it is limited in the number of communication channel between PEs. To overcome this limit, the two dimensional(2D) variant of the 1D-Crosspoint Array, which will be referred to as 2D-Crosspoint Array, is explored in this section. The main purpose and the benefit of the 2D variant is the reduction in number of PEs owing to the added communication channels between PEs. That being said, a crosspoint array has a 2D-layout if (a) each PE is paired with at most four PEs and (b) no wires can cross each other without contacting each other, or in other words, the 2D-Crosspoint Array is placed on a 2D plane. The goal in constructing a 2D-Crosspoint Array is to achieve the complete adjacency in 2D plane with reduced number of PEs compared to the 1D-Crosspoint Array. As it can be seen in Fig. 6.1(a), the 1D-Crosspoint Array can be bent or curved in any shape, which technically can be deemed a 2D-Crosspoint Array. However, such snake-like layout is not taking any advantage of additional communication channels between PEs and is merely '2D shaped' 1D-Crosspoint Array. Crosspoints can be added to create a grid as in Fig. 6.1(b), but addition of such crosspoints without any consideration to adjacency of PEs is a misuse of hardware because numerous redundant adjacency will be created and because the number of PEs will be the same as 1D-Crosspoint Array. To fully utilize the benefit of being constructed on a 2D plane, a new



(a) A 1D-Crosspoint Array bent to a rectangular shape.



(b) A 1D-Crosspoint Array with added vertical crosspoints between PEs above and below.

Figure 6.1: Illustration of 1D-Crosspoint Array for the example case $n = 8$. The crosspoints are shown as simple lines for simplicity.

construction method for 2D-Crosspoint Array will be proposed in the next subsection.

6.1.1 Construction of 2D-Crosspoint Array

The cyclic permutation used in 1D-Crosspoint Array will be used, as it guarantees the complete adjacency. The construction method is shown in Algorithm 7, and an example construction following the Algorithm 7 for the case of $n = 8$ is shown in Fig. 6.2. The cyclic permutation p^j, $0 \leq j \leq n/2$, is shown in Fig. 6.2(a), and the empty frame mentioned in Step 1 is shown in Fig. 6.2(b). Fig. 6.2(c) shows how each of the PE corresponding to the first (and the only) cycle of p, $(0, 1, 2, 3, 4, 5, 6, 7)$ is place on the array starting from the leftmost top position, and then right lower position, right upper position, and so on. Fig. 6.2(d) shows the 2D-Crosspoint Array after Step 5, where the first two elements of the cycle, 0 and 1 is placed following the cycle. Note that with the permutation p placed on the 2D-Crosspoint Array, the adjacencies of the p^2 is already realized on the array. In Fig. 6.2(e), it can be seen the top row (marked as (1)) is the first cycle of p^2, and the bottom row (marked as (2)) is the second cycle of p^2. This is the reason for

Algorithm 7: Basic construction method for 2D-Crosspoint Array

Step 1: Suppose that the cycles in $p, p^2, \cdots, p^{\frac{n}{2}}$ are written so that the first element is the smallest in every one of them. Suppose that an empty frame of a two dimensional array of PEs in a layout shown in Fig. 6.2(b) is provided without the actual assignment.

Step 2: Set P as a queue of all p^j's, $1 \leq j \leq \lfloor n/2 \rfloor$, in ascending order of j, $P = [p^1, p^2, \cdots, p^{\lfloor n/2 \rfloor}]$.

Step 3: Initialize $i = 1$.

Step 4: Place a cycle in p^i on the empty frame from left to right in a zig-zag fashion. See Fig. 6.2(c) for detail.

Step 5: If the size of the cycle placed in Step 4 is greater than 2, place the first two elements of the cycle on the array starting from the leftmost empty position.

Step 6: Repeat Step 4 and 5 until all cycles are placed on the array.

Step 7: Remove p^i and p^{2i} from P.

Step 8: Update $i = i + 1$ until $p^i \in P$, then go back to Step 4. Repeat until P is empty.

removing p^{2j} together with p^j in Step 7, and is the main property that let 2D-Crosspoint Array require less number of PEs compared to 1D-Crosspoint Array.

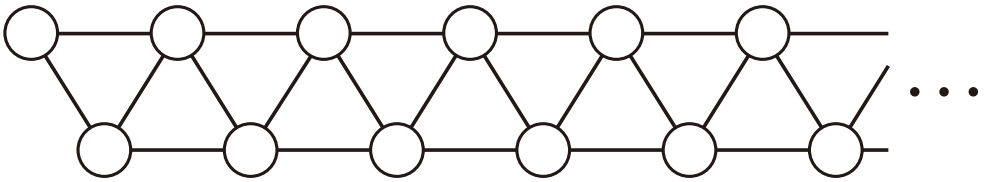
Throughout this chapter, the two overlapping permutations will be referred to as being *paired*. The following lemma proves which permutations can be paired on the 2D-Crosspoint Array.

Lemma 2. Placing permutation p^{j_1} following Steps 4 and 5 of Algorithm 7 simultaneously creates the adjacency of p^{2j_1} on the 2D-Crosspoint Array.

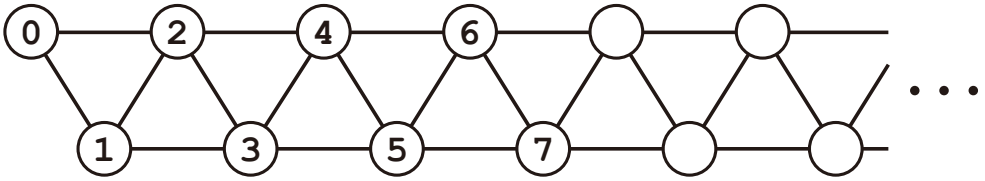
Proof. Consider a cyclic permutation, p^{j_1}. Once the p^{j_1} is placed on the 2D-Crosspoint Array in the zig-zag fashion, each of the top and bottom row has the adjacency of every other element of the p^{j_1}. Given that the cyclic permutation formula for p^{j_1} is $(i + j_1) \bmod n$, the formula for top and bottom row elements is $(i + 2j_1) \bmod n$, indicating that the adjacency of the top and bottom row is in fact the adjacency of the permutation p^{2j_1}. Therefore, placing p^{j_1} creates the adjacency

$$
\begin{aligned}
p &= (0,1,2,3,4,5,6,7) \\
p^2 &= (0,2,4,6)\ (1,3,5,7) \\
p^3 &= (0,3,6,1,4,7,2,5) \\
p^4 &= (0,4)\ (1,5)\ (2,6)\ (3,7)
\end{aligned}
$$

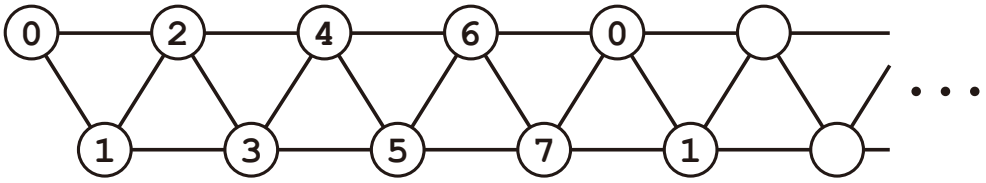
(a) p^j, $0 \leq j \leq n/2$, for $n = 8$ case



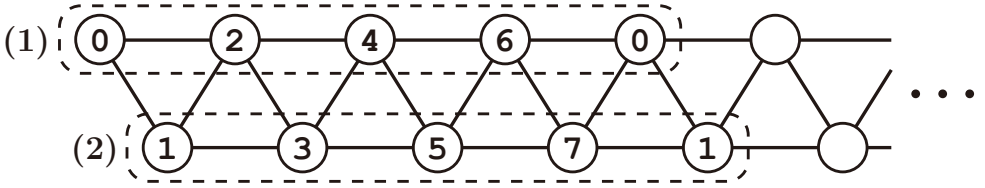
(b) Empty frame for 2D-Crosspoint Array



(c) The 2D-Crosspoint Array after placing the first cycle in Step 4



(d) The 2D-Crosspoint Array after placing the two elements in Step 5



(e) The PEs marked as (1) and (2) is adjacency of the first and second cycle of p^2 respectively

Figure 6.2: Construction of 2D-Crosspoint Array for the case of $n = 8$.

of not only p^{j_1}, but also p^{2j_1}. □

From Lemma 2, it can be concluded that the Algorithm 7, even with the Step 7 skipping a permutation, creates the complete adjacency on the array because all of the adjacencies in $\lfloor n/2 \rfloor$ permutations are placed on the array. The adjacency of the two end elements of a cycle is retained by the addition of first two elements after placing a cycle on the array. With the valid construction method, now let us analyze the 2D-Crosspoint Array in the next subsection.

6.1.2 Analysis of 2D-Crosspoint Array

The time and hardware complexity of the 2D-Crosspoint Array will be analyzed in this subsection. Given that the algorithms derived for 1D-Crosspoint Array utilize the complete adjacency and implement high level parallelism, the order of time complexities of the algorithms run on the 2D-Crosspoint Array is the same as the complexities of 1D-Crosspoint Array. The 2D-Crosspoint Array achieves the complete adjacency, so there is minimal difference in the communication pattern. More precisely, because the algorithms implement high level of parallelism, the workload of each PE is rather small, and the only modification needed to implement 1D-Crosspoint Array algorithms on the 2D-Crosspoint Array is to instruct the PEs to check and communicate with four neighbors instead of two neighbors. For example of sorting algorithm(Algorithm 2), the PE of class 2 in Fig. 6.2(e) would send the input element it was distributed with to the PEs of class 0 and 1, and receive elements from PEs of class 3 and 4 and carry out the comparison. This would double the time complexity of the computations for some of the PEs, but since the order of the time complexity do not change by doubling, the order of time complexity remains unchanged.

In terms of the hardware complexity, the number of permutations needed to achieve com-

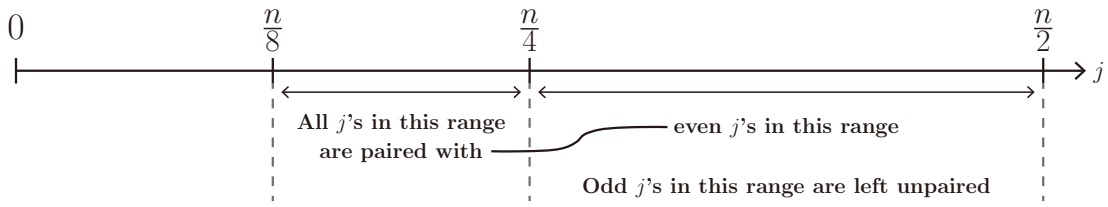


Figure 6.3: Illustration of how j's are paired.

plete adjacency can be assessed by computing how many permutations overlap for a given n. It is not as simple as dividing the number of PEs of 1D-Crosspoint Array in half, because the exponents must an integer, and because no permutation can be paired more than once. The following proposition proves the order on the number of PEs required for 2D-Crosspoint Array for n that is power of 2.

Proposition 13. For n that is a power of 2, the minimum number of permutations that needs to be placed on the 2D-Crosspoint Array by Algorithm 7 in order to create the complete adjacency is $\frac{n}{3}\left(1-\left(\frac{1}{4}\right)^{\lfloor(\lg n-1)/2\rfloor}\right)$, and for large n, it can be simplified to $n/3$.

Proof. Since no p^j, $1 \leq j \leq n/2$, can be paired more than once, we take a top-down approach: pair the permutations with greater j first. We divide the range of j into three parts, $1 \leq j \leq n/8$, $n/8 < j \leq n/4$, and $n/4 < j \leq n/2$ as shown in Fig 6.3. We start paring the j's in $n/4 < j \leq n/2$ to the j's in $n/8 < j \leq n/4$. Every even j's in $n/4 < j \leq n/2$ will be paired, and every odd j's will not be. The odd j's in $n/4 < j \leq n/2$ cannot be paired with any j's since the only possible pairing can happen with even greater j's in the range $n/2 < j \leq n$ which is out of bounds. Therefore, out of $3n/8$ j's in the $n/8 < j \leq n/2$, $2n/8$ j's are paired together($n/8$ pairs) and $n/8$ j's are not paired. Meaning, $n/8 + n/8 = n/4$ permutations are enough to create the complete adjacency of the $3n/8$ permutations in the $n/8 < j \leq n/2$ range.

Because the above pairing process did not pair any j's in the lesser range, $1 \leq j \leq n/8$, we can

recursively apply the same procedure to the j's in that lesser range. Then, the total number of permutations needed for complete adjacency on the 2D-Crosspoint Array can be calculated by the following series,

$$\frac{n}{4} + \frac{n}{4} \cdot \frac{1}{4} + \frac{n}{4} \cdot \frac{1}{4} \cdot \frac{1}{4} \cdots + \frac{n}{4} \cdot \left(\frac{1}{4}\right)^{x-1} = \frac{n}{3}\left(1 - \left(\frac{1}{4}\right)^{x}\right),$$

where $x = \lfloor (\lg n - 1)/2 \rfloor$, the number of recurrence. When n is large, x is also large, and the above equation simplifies to $n/3$. □

Proposition 13 indicates that the 2D-Crosspoint Array needs $2/3$ number of permutations compared to how many 1D-Crosspoint Array needs, $n/2$. To assess the number of PEs required on the 2D-Crosspoint Array to accomplish the complete adjacency, the number of permutations proved by the above proposition is insufficient, as two PEs are added after each cycles is placed. The following lemma proves the total number of PEs added after each cycle, for an n that is power of 2.

Proposition 14. For n that is power of 2, the total number of PEs added after placing a cycle in Step 5 of Algorithm 7 is $n(\lg n + 3)/6$.

Proof. Assume $n = 2^{\alpha}$. Proposition 7 proved that the number of cycles in permutation p^j is $\gcd(n, j)$, where $1 \leq j \leq n/2$. Therefore, calculating the total number of added PEs is equivalent to calculating

$$2\sum_{j=1}^{n/2} \gcd(n, j) = 2(\gcd(n, n/2) + \gcd(n, n/2 - 1) + \cdots + \gcd(n, 2) + \gcd(n, 1)). \quad (6.1)$$

However, because of the permutation pairing by Proposition 13, not all of the permutations are

placed. By taking a similar approach to the proof of Proposition 13, it is known that out of the $n/8 < j \leq n/2$ range, if all p^j for $n/8 < j \leq n/4$ is placed, then only the p^j for odd j's in $n/4 < j \leq n/2$ need to be placed on the 2D-Crosspoint Array. So we first divide the summation of Eq. 6.1 into three summations as

$$2\sum_{j=1}^{n/2}\gcd(n,j) = 2\left(\sum_{j=1}^{n/8}\gcd(n,j) + \sum_{j=n/8}^{n/4}\gcd(n,j) + \sum_{j=n/4}^{n/2}\gcd(n,j)\right). \tag{6.2}$$

Of the three summations, similarly to the proof of Proposition 13, we don't consider $\sum_{j=1}^{n/8}\gcd(n,j)$ yet, and recursively apply the logic of calculating $\sum_{j=n/8}^{n/4}\gcd(n,j) + \sum_{j=n/4}^{n/2}\gcd(n,j)$ to the range later. That being said, the total number of added PEs for the range of $n/8 < j \leq n/4$ is

$$2\sum_{j=n/8}^{n/4}\gcd(n,j) = 2(\gcd(n,n/4) + \gcd(n,n/4-1) + \cdots + \gcd(n,n/8)). \tag{6.3}$$

Since $n = 2^\alpha > j$, every $\gcd$ term is a power of 2. However, instead of checking the divisors of each j, we check how many j's are the multiple of each power of 2 numbers. In other words, we convert the above equation to

$$2\sum_{j=n/8}^{n/4}\gcd(n,j) = 2(\beta_{\alpha-2}2^{\alpha-2} + \beta_{\alpha-3}2^{\alpha-3} + \cdots + \beta_0 2^0), \tag{6.4}$$

where β_x denotes the number of j's such that $\gcd(n,j) = 2^x$. Because we need to check the 'greatest' common divisor, we start with the greatest power of 2. In particular, we check how many numbers between $n/8 = 2^{\alpha-3}$ and $n/4 = 2^{\alpha-2}$ are a multiple of $2^{\alpha-2}$, and then check how many are a multiple of $2^{\alpha-3}$, and repeat until we check for multiple of 1. That is, $2^{\alpha-2}$ is divided

by each of 2^x where $\alpha - 2 \geq x \geq 0$. The results are the number of j's that are multiple of 2^x, where $1 \leq j \leq n/4$. Since every other j of the divided result will be a multiple of 2^{x+1} (except $x = \alpha - 2$ case), only half of their $\gcd(n, j)$ will be 2^x. Also, the range of the summation in Eq. 6.3 is $n/8 \leq j \leq n/4$, which is half of the $1 \leq j \leq n/4$. Therefore, the result of $2^{\alpha-2}/2^x$ must be divided by 4 in order to evaluate β_x. The table below shows the flow of such approach.

2^x	$2^{\alpha-2}/2^x = \beta_x$ (=num of j's that $\gcd(n, j) = 2^x$)
$2^{\alpha-2}$	$2^{\alpha-2}/2^{\alpha-2} = 1$
$2^{\alpha-3}$	$(2^{\alpha-2}/2^{\alpha-3})/4 < 1 \therefore 0$
$2^{\alpha-4}$	$(2^{\alpha-2}/2^{\alpha-4})/4 = 1$
$2^{\alpha-5}$	$(2^{\alpha-2}/2^{\alpha-5})/4 = 2$
$\vdots$	$\vdots$
2^1	$(2^{\alpha-2}/2^1)/4 = 2^{\alpha-5}$
2^0	$(2^{\alpha-2}/2^0)/4 = 2^{\alpha-4}$

(A demonstration of calculating the β_x's for the case of $n = 128$ can be found in Table 6.1.) Based on the table above, the Eq. 6.3 can be simplified to

$$
\begin{aligned}
2\sum_{j=n/8}^{n/4} \gcd(n, j) &= 2(1 \cdot 2^{\alpha-2} + 1 \cdot 2^{\alpha-4} + 2 \cdot 2^{\alpha-5} + \cdots + 2^{\alpha-5} \cdot 2^1 + 2^{\alpha-4} \cdot 2^0) \\
&= 2(2^{\alpha-2} + 2^{\alpha-4} + 2^{\alpha-4} + \cdots + 2^{\alpha-4}) \\
&= 2(2^{\alpha-2} + (\alpha - 3) \cdot 2^{\alpha-4}) \\
&= 2\left(2^{\alpha-2} + (\lg n - 3) \cdot \frac{n}{16}\right) \\
&= \frac{n(\lg n + 1)}{8}.
\end{aligned} \tag{6.5}
$$

Note that Eq. 6.5 is sum of the added PEs in Step 5 of Algorithm 7 for j's in the range of $n/8 < j \leq n/4$. Recall that for the range of $n/4 < j \leq n/2$, only the p^j with odd j is placed on

the 2D-Crosspoint Array and consist of one cycle. Then, the sum of added PEs for the range of $n/8 < j \leq n/2$ is

$$2\sum_{j=n/8}^{n/4} \gcd(n,j) + 2\sum_{j=n/4}^{n/2} \gcd(n,j) = \frac{n(\lg n+1)}{8} + 2\cdot\frac{1}{2}\cdot\frac{n}{4} = \frac{n(\lg n+3)}{8} \tag{6.6}$$

Thus, the total number of added PEs for the range of $1 \leq j \leq n/2$, or the Eq. 6.1, can be calculated by the following series.

$$\frac{n(\lg n+3)}{8}\left(1+\frac{1}{4}+\left(\frac{1}{4}\right)^2+\cdots+\left(\frac{1}{4}\right)^x\right) = \frac{n(\lg n+3)}{6}\left(1-\left(\frac{1}{4}\right)^x\right), \tag{6.7}$$

where $x = \lfloor(\lg n - 1)/2\rfloor$. For large n, the equation simplifies to $n(\lg n+3)/6$. □

Combining Proposition 13 and 14, the following Theorem proves the number of PEs placed on the 2D-Crosspoint Array by Algorithm 7.

Theorem 2. The number of PEs placed on the 2D-Crosspoint Array by Algorithm 7 is $n^2/3 + n(\lg n+3)/6$ for a large n that is power of 2.

Proof. Proposition 13 proved the number of permutations that must be placed on the 2D-Crosspoint Array is $n/3$ for large n. Since each permutation is of length n, the hardware complexity of the placed permutations is $n^2/3$. By adding the number of added PEs after placing a cycle from Proposition 14, we attain the hardware complexity of 2D-Crosspoint Array, $n^2/3 + n(\lg n+3)/6$. □

For large n, the $n^2/3$ term dominates $n(\lg n+3)/6$ term and the number of PEs needed become $n^2/3$, which is $2/3 \approx 0.67$ of how many the 1D-Crosspoint Array needs. The reduction trend can be seen in Fig. 6.4.

2^x	Prospective β_x	Corresponding j's	Comment
32	$32/32 = \mathbf{1}$	32	Single j that $\gcd(128, j) = 32$
16	$32/16 = 2$	$16, 32$	Two possible j's, but every other j is multiple of 32. So, $\swarrow$
	$\hookrightarrow 2/2 = 1$	16	But the range is $n/8 < j \leq n/4$. So, $\swarrow$
	$\hookrightarrow 1/2 < 1$	$\varnothing$	No j's in $16 < j \leq 32$ such that $\gcd(128, j) = 16$.
8	$32/8 = 4$	$8, 16, 24, 32$	Four possible j's, but every other j is multiple of 16(or greater). So, $\swarrow$
	$\hookrightarrow 4/2 = 2$	$8, 24$	But the range is $n/8 < j \leq n/4$. So, $\swarrow$
	$\hookrightarrow 2/2 = \mathbf{1}$	24	One j in $16 < j \leq 32$ such that $\gcd(128, j) = 8$.
4	$32/4 = 8$	$4, 8, 12, 16,$ $20, 24, 28, 32$	Eight possible j's, but every other j is multiple of 8(or greater). So, $\swarrow$
	$\hookrightarrow 8/2 = 4$	$4, 12, 20, 28$	But the range is $n/8 < j \leq n/4$. So, $\swarrow$
	$\hookrightarrow 4/2 = \mathbf{2}$	$20, 28$	Two j's in $16 < j \leq 32$ such that $\gcd(128, j) = 4$.
2	$32/2 = 16$	$2, 4, 6, 8,$ $\cdots, 28, 30, 32$	Sixteen possible j's, but every other j is multiple of 4(or greater). So, $\swarrow$
	$\hookrightarrow 16/2 = 8$	$2, 6, 10, 14,$ $\cdots, 22, 26, 30$	But the range is $n/8 < j \leq n/4$. So, $\swarrow$
	$\hookrightarrow 8/2 = \mathbf{4}$	$18, 22, 26, 30$	Four j's in $16 < j \leq 32$ such that $\gcd(128, j) = 2$.
1	$32/1 = 32$	$1, 2, 3, \cdots, 32$	All numbers possible, but even numbers are multiple of 2. So, $\swarrow$
	$\hookrightarrow 32/2 = 16$	$1, 3, 5, \cdots, 31$	But the range is $n/8 < j \leq n/4$. So, $\swarrow$
	$\hookrightarrow 16/2 = \mathbf{8}$	$17, 19, \cdots, 31$	All odd j's that are $16 < j \leq 32$.

Table 6.1: Demonstration of calculating β_x for an example case of $n = 128$. The comments with '$\swarrow$' symbol at the end is the explanation to why the prospective β_x in each line is divided by 2 in the next line.

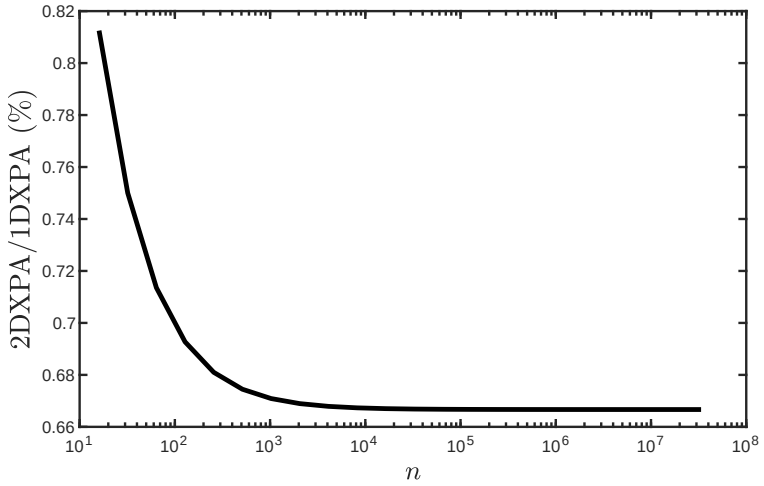


Figure 6.4: The hardware complexity reduction of 2D-Crosspoint Array compared to that of 1D-Crosspoint Array. It is calculated by dividing number of PEs needed for 3D-Crosspoint Array, $n^2/3 + n(\lg n - 1)/2$, by that of 2D-Crosspoint Array, $n^2/2$.

6.2 3D-Crosspoint Array

The structure can further be extended to three dimensional(3D) structure, allowing more communication channels between PEs. Most of 3D structures are in a cubic formation(e.g., [24, 41, 161, 162]) and thus the 3D-crosspoint array is defined as a crosspoint array that each PE is paired with at most six PEs. There is no restriction on the dimension or wires crossing over each other. Similar to 2D-Crosspoint Array, the goal of 3D-Crosspoint Array is reducing the number of PEs required to achieve the complete adjacency by utilizing the added communication channels of 3D space. In Subsection 6.2.1, a construction method for 3D-Crosspoint Array is presented, and in Subsection 6.2.2, the hardware complexity of the 3D-Crosspoint Array is analyzed.

6.2.1 Construction of 3D-Crosspoint Array

The mechanism of the construction for 3D-Crosspoint Array is similar to that of the 2D-Crosspoint Array. The cyclic permutation is placed on an empty skeleton structure, but takes the advantage of the overlapping permutations and skip some of the permutation. Algorithm 8 shows the construction method of the 3D-Crosspoint Array, and the cyclic permutations p^j,$0 \leq j \leq n/2$, for $n = 16$ case are listed in Fig. 6.5. Using those permutations, an example construction for $n = 16$ case is shown in Fig. 6.6. Fig. 6.6(a) is the empty frame of the 3D-Crosspoint Array and illustrates the added communication channels(crosspoints) which connects every other PEs on the top and bottom row respectively. Fig. 6.6(b) and (c) shows how the permutations are placed on the empty array in Steps 4 and 5. Fig. 6.6(d) and (e) illustrates the adjacencies of the p^4 created on the array in addition to the adjacencies of p and p^2.

The following lemma proves exactly which permutations overlap in the 3D-Crosspoint Array.

Lemma 3. Placing permutation p^{j_1} following Steps 4 and 5 of Algorithm 8 creates the adjacency of p^{j_1}, p^{2j_1} and p^{4j_1} on the 3D-Crosspoint Array at the same time.

Proof. Consider two cyclic permutation p^{j_1}, p^{j_2}, where $2j_1 = j_2$. From Lemma 2, it is known that the top and bottom row creates the adjacency of p^{j_2}. Given that the added crosspoints of the 3D-Crosspoint Array connects every other elements of the top and bottom row respectively, the formula of those elements is $(i + 2j_2) \bmod n = (i + 4j_1) \bmod n$, indicating that the adjacency of p^{4j_1} is created by the added crosspoints. Therefore, placing p^{j_1} creates the adjacency of p^{j_1}, p^{2j_1} and p^{4j_1}. □

$$
\begin{aligned}
p &= (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)\\
p^2 &= (0,2,4,6,8,A,C,E)\ (1,3,5,7,9,B,D,F)\\
p^3 &= (0,3,6,9,C,F,2,5,8,B,E,1,4,7,A,D)\\
p^4 &= (0,4,8,C)\ (1,5,9,D)\ (2,6,A,E)\ (3,7,B,F)\\
p^5 &= (0,5,A,F,4,9,E,3,8,D,2,7,C,1,6,B)\\
p^6 &= (0,6,C,2,8,E,4,A)\ (1,7,D,3,9,F,5,B)\\
p^7 &= (0,7,E,5,C,3,A,1,8,F,6,D,4,B,2,9)\\
p^8 &= (0,8)\ (1,9)\ (2,A)\ (3,B)\ (4,C)\ (5,D)\ (6,E)\ (7,F)
\end{aligned}
$$

Figure 6.5: p^j, $0 \leq j \leq n/2$, for $n = 16$ case. Class IDs are written in hex digits.

Similar to the 2D-Crosspoint Array case, the Algorithm 8 creates the complete adjacency on the array, even though it skips two permutations in Step 7. In addition, the adjacency of the two end elements of a cycle is retained by the addition of the four elements added in Step 5. With the valid construction method, now let us analyze the 2D-Crosspoint Array in the next subsection.

Algorithm 8: Basic construction method for 2D-Crosspoint Array

Step 1: Suppose that the cycles in $p, p^2, \cdots, p^{\frac{n}{2}}$ are written so that the first element is the smallest in every one of them. Suppose that an empty frame of a two dimensional array of PEs in a layout shown in Fig. 6.6a is provided without the actual assignment.

Step 2: Set P as a queue of all p^j's, $1 \leq j \leq \lfloor n/2 \rfloor$, in ascending order of j, $P = [p^1, p^2, \cdots, p^{\lfloor n/2 \rfloor}]$.

Step 3: Initialize $i = 1$.

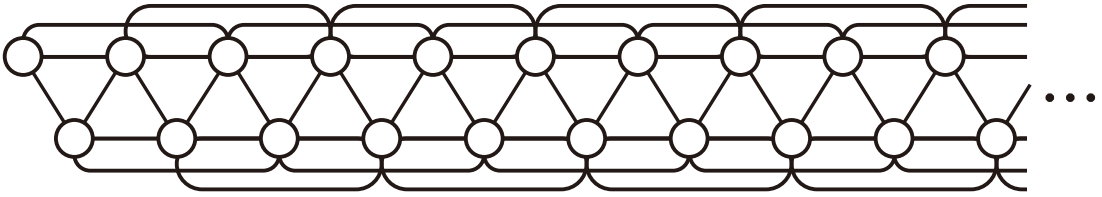
Step 4: Place a cycle in p^i on the empty frame from left to right in a zig-zag fashion. See Fig. 6.6b for detail.

Step 5: If the size of the cycle placed in Step 4 is greater than 2, place the first four elements of the cycle on the array starting from the leftmost empty position.

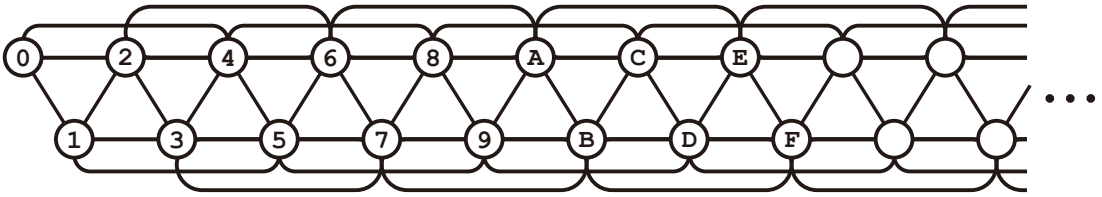
Step 6: Repeat Steps 4 and 5 until all cycles are placed on the array.

Step 7: Remove p^i, p^{2i}, and p^{4i} from P.

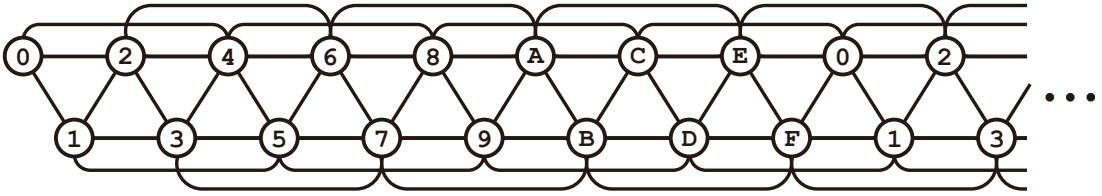
Step 8: Update $i = i + 1$ until $p^i \in P$, then go back to Step 4. Repeat until P is empty.



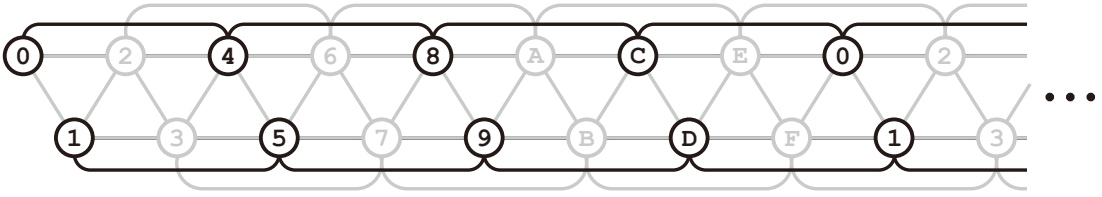
(a) Empty frame for 3D-Crosspoint Array



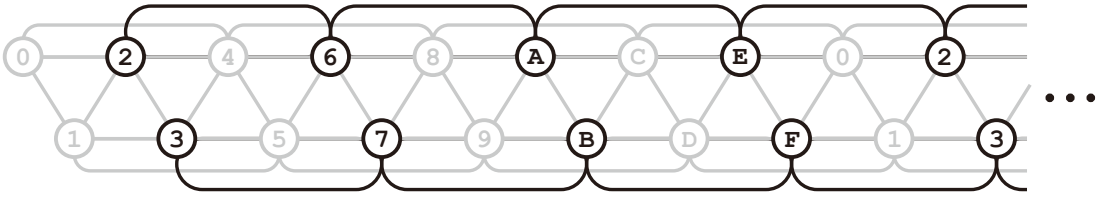
(b) The 3D-Crosspoint Array after Step 4 of Algorithm 8



(c) The 3D-Crosspoint Array after Step 5 of Algorithm 8



(d) The adjacency of first and second cycle of p^4 is highlighted



(e) The adjacency of third and fourth cycle of p^4 is highlighted

Figure 6.6: Construction of 3D-Crosspoint Array for the case of $n = 16$.

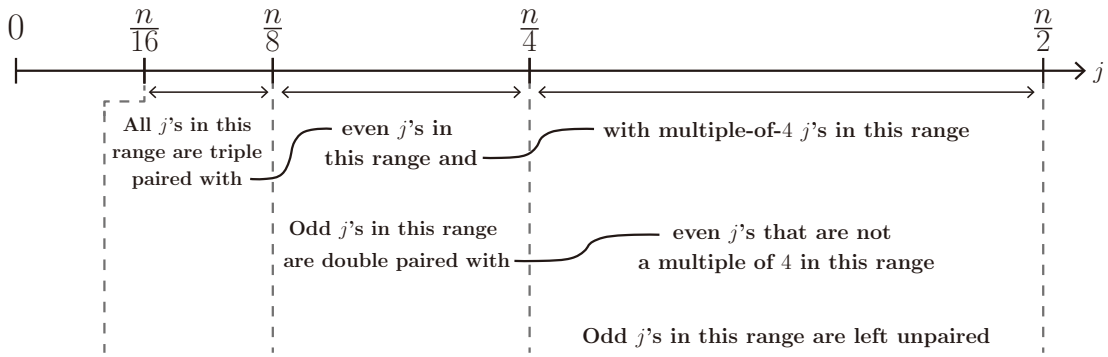


Figure 6.7: Illustration of how j's are paired.

6.2.2 Analysis of 3D-Crosspoint Array

The hardware complexity of the 3D-Crosspoint Array will be analyzed in this subsection. Since the time complexity remains the same similar to the 2D-Crosspoint Array, the reduction of number of PEs will be closely analyzed. To differentiate the pairing of two and three permutations, we refer to as *triple pair* when three permutations are paired as in Lemma 3, and *double pair* when two permutations are paired as in Lemma 2. The number of permutations needed to achieve the complete adjacency on 3D-Crosspoint Array is proved in the following proposition.

Proposition 15. For n that is a power of 2, the minimum number of permutations that needs to be placed on the 3D-Crosspoint Array by Algorithm 8 in order to create the complete adjacency is $\frac{2n}{7}\left(1-\left(\frac{1}{8}\right)^{\lfloor(\lg n-1)/3\rfloor}\right)$.

Proof. We divide the $1 \leq j \leq n/2$ range into four parts, $1 \leq j \leq n/16$, $n/16 < j \leq n/8$, $n/8 < j \leq n/4$, and $n/4 < j \leq n/2$ as shown in Fig. 6.7. We start pairing the j's in $n/16 < j \leq n/8$ range to the j's in $n/8 < j \leq n/4$ and $n/4 < j \leq n/2$ range. Assume a j_1 in $n/16 < j \leq n/8$ range. For any j_1 in $n/16 < j \leq n/8$, $2j_1$ and $4j_1$ will be in $n/8 < j \leq n/4$ and $n/4 < j \leq n/2$ range respectively and will be paired with corresponding j_1. Note that the $2j_1$'s that are paired

are all even numbers, and the $4j_1$'s that are paired are all multiple of 4. What is left is the odd numbers in $n/8 < j \leq n/4$ and odd numbers and even numbers that are not a multiple of 4 in $n/4 < j \leq n/2$. No other j's in $n/16 < j \leq n/2$ can be triple paired because no multiple of 4 is left in this range. However, the odd numbers in $n/8 < j \leq n/4$ and the even numbers in $n/4 < j \leq n/2$ can be double paired. Therefore, out of $7n/16$ j's in the $n/16 < j \leq n/2$ range, $n/16$ j's are triple paired, $n/16$ j's are double paired, and $2n/16$ j's are not paired. Meaning, $n/16 + n/16 + 2n/16 = n/4$ permutations are enough to create the complete adjacency of the $7n/16$ permutations in the $n/16 < j \leq n/2$ range. We can recursively apply the same procedure to the $1 \leq j \leq n/16$ range, and the total number of permutations needed can be expressed as the following series,

$$\frac{n}{4} + \frac{n}{4} \cdot \frac{1}{8} + \frac{n}{4} \cdot \frac{1}{8} \cdot \frac{1}{8} \cdots + \frac{n}{4} \cdot \left(\frac{1}{8}\right)^{x-1} = \frac{2n}{7}\left(1 - \left(\frac{1}{8}\right)^{x}\right),$$

where $x = \lfloor (\lg n - 1)/3 \rfloor$, the number of recurrence. When n is large, x is also large, and the above equation simplifies to $2n/7$. □

Proposition 15 suggests that the 3D-Crosspoint Array needs $4/7$ of the number of permutations the 1D-Crosspoint Array needs. To assess the number of PEs required achieve complete adjacency on the 3D-Crosspoint Array, the total number of PEs added in Step 5 of Algorithm 8 need to be evaluated. The following lemma proves the total number of PEs added for an n that is power of 2.

Proposition 16. For n that is power of 2, the total number of PEs added after placing a cycle in Step 5 of Algorithm 8 is $n(3 \lg n + 2)/12$.

Proof. Most of the proof is similar to that of Proposition 14. Assume $n = 2^{\alpha}$. Proposition 7 proved that the number of cycles in permutation p^j is $\gcd(n, j)$, where $1 \leq j \leq n/2$. Therefore, calculating the total number of added PEs is equivalent to calculating

$$2\sum_{j=1}^{n/2} \gcd(n, j) = 2(\gcd(n, n/2) + \gcd(n, n/2 - 1) + \cdots + \gcd(n, 2) + \gcd(n, 1)). \tag{6.8}$$

However, because of the permutation pairing by Proposition 15, not all of the permutations are placed. From the Proposition 15, it is known that out of the $n/16 < j \leq n/2$ range, all p^j where $n/16 < j \leq n/4$ is placed, and only the p^j for odd j in $n/4 < j \leq n/2$ is placed on the 3D-Crosspoint Array. So we first divide the summation of Eq. 6.8 into three summations as

$$2\sum_{j=1}^{n/2} \gcd(n, j) = 2\left(\sum_{j=1}^{n/16} \gcd(n, j) + \sum_{j=n/16}^{n/4} \gcd(n, j) + \sum_{j=n/4}^{n/2} \gcd(n, j)\right). \tag{6.9}$$

Of the three summations, similarly to the proof of Proposition 15, we don't consider $\sum_{j=1}^{n/16} \gcd(n, j)$ yet, and recursively apply the logic of calculating $\sum_{j=n/16}^{n/4} \gcd(n, j) + \sum_{j=n/4}^{n/2} \gcd(n, j)$ to the range later. That being said, the total number of added PEs for the range of $n/16 < j \leq n/4$ is

$$2\sum_{j=n/16}^{n/4} \gcd(n, j) = 2(\gcd(n, n/4) + \gcd(n, n/4 - 1) + \cdots + \gcd(n, n/16)). \tag{6.10}$$

Similar to the proof of Proposition 14, we convert the above equation to

$$2\sum_{j=n/16}^{n/4} \gcd(n, j) = 2(\beta_{\alpha-2}2^{\alpha-2} + \beta_{\alpha-3}2^{\alpha-3} + \cdots + \beta_0 2^0), \tag{6.11}$$

where β_x denotes the number of j's such that $\gcd(n, j) = 2^x$. The β's can be evaluated by checking how many numbers between $n/16 = 2^{\alpha-4}$ and $n/4 = 2^{\alpha-2}$ are multiple of each power of 2, in the same way as in the proof of Proposition 14. The $2^{\alpha-2}$ is divided by each of 2^x where $\alpha - 2 \geq x \geq 0$. The results are the number of j's that are multiple of 2^x, where $1 \leq j \leq n/4$. Since every other j of the divided result will be a multiple of 2^{x+1} (except $x = \alpha - 2$ case), only half of their $\gcd(n, j)$ will be 2^x. Also, the range in the summation in Eq. 6.10 is $n/16 < j \leq n/4$, which is $3/4$ of the $1 \leq j \leq n/4$. Therefore, the result of $2^{\alpha-2}/2^x$ must be multiplied by $3/8$ in order to evaluate β_x. Because β must be an integer, the floor function is applied as well. The table below shows the flow of such approach.

2^x	$\lfloor 2^{\alpha-2}/2^x \rfloor = \beta_x$ (=num of j's that $\gcd(n, j) = 2^x$)
$2^{\alpha-2}$	$\lfloor 2^{\alpha-2}/2^{\alpha-2} \rfloor = 1$
$2^{\alpha-3}$	$\lfloor (2^{\alpha-2}/2^{\alpha-3}) \cdot 3/8 \rfloor = \lfloor 2 \cdot 3/8 \rfloor < 1 \therefore 0$
$2^{\alpha-4}$	$\lfloor (2^{\alpha-2}/2^{\alpha-4}) \cdot 3/8 \rfloor = \lfloor 2^2 \cdot 3/8 \rfloor = 1$
$2^{\alpha-5}$	$\lfloor (2^{\alpha-2}/2^{\alpha-5}) \cdot 3/8 \rfloor = \lfloor 2^3 \cdot 3/8 \rfloor = 3$
$2^{\alpha-6}$	$\lfloor (2^{\alpha-2}/2^{\alpha-6}) \cdot 3/8 \rfloor = \lfloor 2^4 \cdot 3/8 \rfloor = 3 \cdot 2^1$
$2^{\alpha-7}$	$\lfloor (2^{\alpha-2}/2^{\alpha-7}) \cdot 3/8 \rfloor = \lfloor 2^5 \cdot 3/8 \rfloor = 3 \cdot 2^2$
$\vdots$	$\vdots$
2^1	$\lfloor (2^{\alpha-2}/2^1) \cdot 3/8 \rfloor = \lfloor 2^{\alpha-3} \cdot 3/8 \rfloor = 3 \cdot 2^{\alpha-6}$
2^0	$\lfloor (2^{\alpha-2}/2^0) \cdot 3/8 \rfloor = \lfloor 2^{\alpha-2} \cdot 3/8 \rfloor = 3 \cdot 2^{\alpha-5}$

Based on the table above, the Eq. 6.3 can be simplified to

$$
\begin{aligned}
&2\sum_{j=n/16}^{n/4}\gcd(n,j)\\
&\quad=2(1\cdot 2^{\alpha-2}+1\cdot 2^{\alpha-4}+3\cdot 2^{\alpha-5}+3\cdot 2^{1}\cdot 2^{\alpha-6}+\cdots+3\cdot 2^{\alpha-6}\cdot 2^{1}+3\cdot 2^{\alpha-5}\cdot 2^{0})\\
&\quad=2(2^{\alpha-2}+2^{\alpha-4}+3\cdot 2^{\alpha-5}+3\cdot 2^{\alpha-5}+\cdots+3\cdot 2^{\alpha-5})\\
&\quad=2(2^{\alpha-2}+2^{\alpha-4}+(\alpha-4)\cdot 3\cdot 2^{\alpha-5})\\
&\quad=\frac{n(3\lg n-2)}{16}.
\end{aligned}
\tag{6.12}
$$

Note that Eq. 6.12 is sum of the added PEs in Step 5 of Algorithm 8 for j's in the range of $n/16 < j \leq n/4$. Recall that for the range of $n/4 < j \leq n/2$, only the p^j with odd j is placed on the 2D-Crosspoint Array and consist of one cycle. Then, the sum of added PEs for the range of $n/16 < j \leq n/2$ is

$$
2\sum_{j=n/16}^{n/4}\gcd(n,j)+2\sum_{j=n/4}^{n/2}\gcd(n,j)=\frac{n(3\lg n-2)}{16}+2\cdot\frac{1}{2}\cdot\frac{n}{4}=\frac{n(3\lg n+2)}{16} \tag{6.13}
$$

Thus, the total number of added PEs, or the Eq. 6.8, can be calculated by the following series.

$$
\frac{n(3\lg n+2)}{16}\left(1+\frac{1}{8}+\left(\frac{1}{8}\right)^{2}+\cdots+\left(\frac{1}{8}\right)^{x}\right)=\frac{n(3\lg n+2)}{14}\left(1-\left(\frac{1}{8}\right)^{x}\right)
$$

where $x=\lfloor(\lg n-1)/2\rfloor$. For large n, the equation simplifies to $n(3\lg n+2)/14$. □

Combining Proposition 15 and 16, the following Theorem proves the number of PEs placed on the 3D-Crosspoint Array by Algorithm 8.

Theorem 3. The number of PEs placed on the 3D-Crosspoint Array by Algorithm 8 is $2n^2/7+$

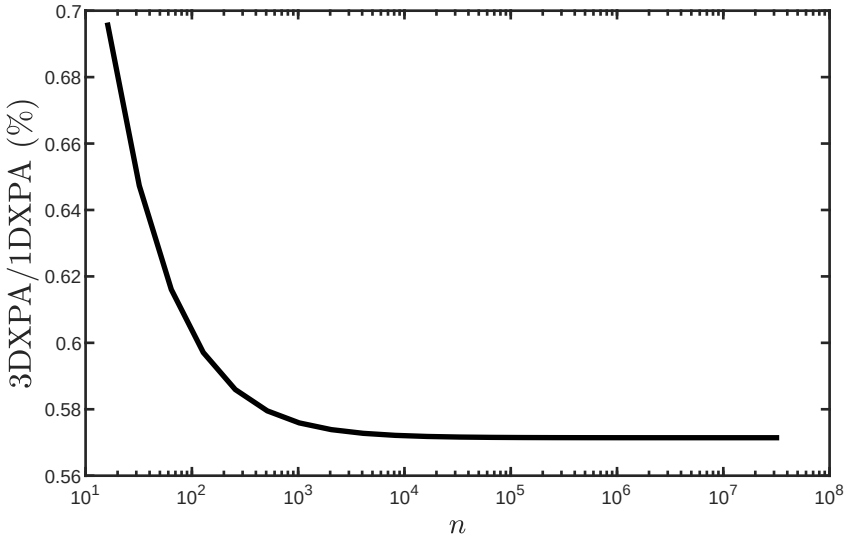


Figure 6.8: The hardware complexity reduction of 3D-Crosspoint Array compared to that of 1D-Crosspoint Array. It is calculated by dividing number of PEs needed for 3D-Crosspoint Array, $2n^2/7 + n(3\lg n + 2)/14$, by that of 2D-Crosspoint Array, $n^2/2$.

$n(3\lg n + 2)/14$ for a large n that is power of 2.

Proof. Proposition 15 proved the number of permutations that must be placed on the 3D-Crosspoint Array is $2n/7$ for large n. Since each permutation is of length n, the hardware complexity of the placed permutations is $2n^2/7$. By adding the number of added PEs after placing a cycle from Proposition 16, we attain the hardware complexity of 2D-Crosspoint Array, $2n^2/7 + n(3\lg n + 2)/14$. □

For large n, the $2n^2/7$ term dominates $n(3\lg n+2)/14$ term and the number of PEs needed become $2n^2/7$, which is $4/7 \approx 0.57$ of how many the 1D-Crosspoint Array needs. The reduction trend can be seen in Fig. 6.8.

Chapter 7: Concluding Remarks

In this dissertation, we introduced a new network, called 1D-Crosspoint Array to interconnect a set of PEs. We first derived the optimal number of PEs needed to achieve complete adjacency, which were shown to align with Valiant's upper bound on number of PEs for a sorting network. A method was then presented to construct the 1D-Crosspoint Array using the optimal number of PEs for any input size. Then algorithms to solve three different big data problems, namely sorting, matrix multiplication and triangle counting, were devised and analyzed to demonstrate that the 1D-Crosspoint Array performs comparably to or better than existing parallel architectures for the three problems. The absence of communication contention and the increased number of PEs were shown to play a key role even when the communication and computation time complexities were normalized. Although not all algorithms performed better than existing algorithms, it is worth noting that the three algorithms have shown that the 1D-Crosspoint Array can be versatile and be implemented to solve different problems when given a proper algorithm. Furthermore, the three external models for big data introduced an adaptable model of solving big data problems such that different configurations can be set depending on the priorities and available resources. Finally, the higher dimension variants, the 2D- and 3D-Crosspoint Array, were explored and proven to reduce the hardware complexity, albeit the order of magnitude remaining the same.

7.1 Future Work

For the 1D-Crosspoint Array, the minimum number of PEs that is possible to construct the 1D-Crosspoint Array was proven. However, for the 2D- and 3D-Crosspoint Array, while the construction method and the reduction in the number of PEs required were shown, the minimum or optimal number of PEs needed for the two variants were not proven. A mathematical proof for the minimum number of PEs will be useful in analyzing how optimal the 2D and 3D variant presented in this dissertation is. However, it will not be an easy task as the 2D and 3D structures can have a wide range of different shapes, which lead to different number of neighboring PEs for each case. A possible approach would be to fix a shape, for example, a square or a cube, and prove for such specific shapes.

Another useful data would be the experimental results. Due to the unique design of the 1D-Crosspoint Array, experimenting on general purpose computers or servers were not an option. A possible solution would be experimenting on a FPGA board, although it would still be difficult to experiment the situations where the 1D-Crosspoint Array deals with full scale big data, given the general size of FPGA boards.

Bibliography

[1] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988. doi: 10.1137/0217049. URL https://doi.org/10.1137/0217049.

[2] Bogdan S. Chlebus and Imrich Vrťo. Parallel quicksort. *Journal of Parallel and Distributed Computing*, 11(4):332–337, 1991. ISSN 0743-7315. doi: https://doi.org/10.1016/0743-7315(91)90040-G. URL https://www.sciencedirect.com/science/article/pii/074373159190040G.

[3] Charles U. Martel and Dan Gusfield. A fast parallel quicksort algorithm. *Information Processing Letters*, 30(2):97–102, 1989. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(89)90116-6. URL https://www.sciencedirect.com/science/article/pii/0020019089901166.

[4] D. S. Hirschberg. Fast parallel sorting algorithms. *Commun. ACM*, 21(8):657–661, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359582. URL https://doi.org/10.1145/359576.359582.

[5] Hal Varian and Peter Lyman. How much information. *University of California, Berkeley: UC Press*, 2003.

[6] John Gants. Digital universe decade-are you ready? *http://idcdocserv. com/925*, 2010.

[7] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. *Don't Focus on Big Data*, 2, 2017.

[8] David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core. *IDC White Paper*, 13, 2018.

[9] D Reinsel, J Rydning, and JF Gantz. Worldwide global datasphere forecast, 2020–2024: The covid-19 data bump and the future of data growth, 2020.

[10] Hosagrahar V Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[11] M Chen. How the financial services industry is winning with big data, 2018.

[12] Thomas H Davenport and Jill Dyché. Big data in big companies. *International Institute for Analytics*, 3(1-31), 2013.

[13] Samuel H Fuller and Lynette I Millett. *The future of computing performance: game over or next level?* National Academy Press, 2011.

[14] Jeffrey D Ullman. Designing good mapreduce algorithms. *XRDS: Crossroads, The ACM Magazine for Students*, 19(1):30–34, 2012.

[15] J Dongarra, J Hittinger, J Bell, L Chacon, R Falgout, M Heroux, P Hovland, E Ng, C Webster, and S Wild. Applied mathematics research for exascale computing. Technical report, 2 2014. URL `https://www.osti.gov/biblio/1149042`.

[16] Lin Yen-Chun. On balancing sorting on a linear array. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):566–571, 1993.

[17] Yi Pan, Mounir Hamdi, and Keqin Li. Efficient and scalable quicksort on a linear array with a reconfigurable pipelined bus system. *Future Generation Computer Systems*, 13(6): 501–513, 1998.

[18] Yi Pan. Basic data movement operations on the LARPBS model. In *Parallel Computing Using Optical Interconnections*, pages 227–247. Springer, 1998.

[19] Yi Pan and Keqin Li. Linear array with a reconfigurable pipelined bus system—concepts and applications. *Information Sciences*, 106(3-4):237–258, 1998.

[20] Amitava Datta, Subbiah Soundaralakshmi, and Robyn Owens. Fast sorting algorithms on a linear array with a reconfigurable pipelined bus system. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):212–222, 2002.

[21] Laxmi N. Bhuyan and Dharma P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on computers*, 33(04):323–333, 1984.

[22] John P Hayes and Trevor Mudge. Hypercube supercomputers. *Proceedings of the IEEE*, 77(12):1829–1841, 1989.

[23] Geoffrey C Fox, Steve W Otto, and Anthony JG Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel computing*, 4(1):17–31, 1987.

[24] B. Abali, F. Ozguner, and A. Bataineh. Balanced parallel sort on hypercube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):572–581, May 1993. doi: 10.1109/71.224220.

[25] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Commun. ACM*, 24(5):300–309, May 1981. ISSN 0001-0782. doi: 10.1145/358645.358660. URL `http://doi.acm.org.proxy-um.researchport.umd.edu/10.1145/358645.358660`.

[26] Nikilas J Dimopoulos, RD Rasmussen, GS Bololin, BF Lewis, and RM Manning. Hypercycles, interconnection networks with simple routing strategies. In *Proceedings of the Canadian Conference on Electrical & Computer Engineering*. Citeseer, 1988.

[27] Kai Hwang and Joydeep Ghosh. Hypernet: A communication-efficient architecture for constructing massively parallel computers. *IEEE Transactions on Computers*, 100(12): 1450–1466, 1987.

[28] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[29] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[30] Daichi Mukunoki and Toshiyuki Imamura. Implementation and performance analysis of 2.5 d-pdgemm on the k computer. In *International Conference on Parallel Processing and Applied Mathematics*, pages 348–358. Springer, 2017.

[31] Alexander A Sawchuk and Timothy C Strand. Digital optical computing. *Proceedings of the IEEE*, 72(7):758–779, 1984.

[32] Joseph W Goodman, Frederick J Leonberger, Sun-Yuan Kung, and Ravindra A Athale. Optical interconnections for vlsi systems. *Proceedings of the IEEE*, 72(7):850–866, 1984.

[33] LA Bergman, WH Wu, AR Johnston, R Nixon, Sadik C Esener, CC Guest, P Yu, Timothy J Drabik, M Feldman, and Sing H Lee. Holographic optical interconnects for vlsi. *Optical Engineering*, 25(10):1109–1118, 1986.

[34] Gary C Marsden, Philippe J Marchand, Phil Harvey, and Sadik C Esener. Optical transpose interconnection system architectures. *Optics letters*, 18(13):1083–1085, 1993.

[35] Michael R Feldman, Sadik C Esener, Clark C Guest, and Sing H Lee. Comparison between optical and electrical interconnects based on power and speed considerations. *applied optics*, 27(9):1742–1751, 1988.

[36] Marc A Taubenblatt. Optical interconnects for high-performance computing. *Journal of Lightwave Technology*, 30(4):448–457, 2011.

[37] Michael RT Tan, Moray McLaren, and Norman P Jouppi. Optical interconnects for high-performance computing systems. *IEEE Micro*, 33(1):14–21, 2012.

[38] Chen Sun, Mark T Wade, Yunsup Lee, Jason S Orcutt, Luca Alloatti, Michael S Georgas, Andrew S Waterman, Jeffrey M Shainline, Rimas R Avizienis, Sen Lin, et al. Single-chip microprocessor that communicates directly using light. *Nature*, 528(7583):534–538, 2015.

[39] Ian A Young, Edris Mohammed, Jason TS Liao, Alexandra M Kern, Samuel Palermo, Bruce A Block, Miriam R Reshotko, and Peter LD Chang. Optical i/o technology for tera-scale computing. *IEEE Journal of solid-state circuits*, 45(1):235–248, 2009.

[40] Min He, Xiaolong Wu, and Si Qing Zheng. An optimal and processor efficient parallel sorting algorithm on a linear array with a reconfigurable pipelined bus system. *Computers & Electrical Engineering*, 35(6):951–965, 2009.

[41] Keny T Lucas. Parallel enumeration sort on otis-hypercube. In *International conference on contemporary computing*, pages 21–31. Springer, 2010.

[42] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California . . . , 2009.

[43] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[44] Yashpalsinh Jadeja and Kirit Modi. Cloud computing-concepts, architecture and challenges. In *2012 international conference on computing, electronics and electrical technologies (ICCEET)*, pages 877–880. IEEE, 2012.

[45] Rajkumar Buyya and Srikumar Venugopal. A gentle introduction to grid computing and technologies. *database*, 2(R3), 2005.

[46] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop*, pages 1–10. Ieee, 2008.

[47] Uwe Schwiegelshohn, Rosa M Badia, Marian Bubak, Marco Danelutto, Schahram Dustdar, Fabrizio Gagliardi, Alfred Geiger, Ladislav Hluchy, Dieter Kranzlmüller, Erwin Laure, et al. Perspectives on grid computing. *Future Generation Computer Systems*, 26 (8):1104–1115, 2010.

[48] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. ISSN 00255718, 10886842. URL `http://www.jstor.org/stable/2003354`.

[49] Ping Tak Peter Tang, Jongsoo Park, Daehyun Kim, and Vladimir Petrov. A framework for low-communication 1-D FFT. *Scientific Programming*, 21(3-4):181–195, 2013.

[50] R. Al Na'mneh, D. W. Pan, and R. Adhami. Parallel implementation of 1-d fast fourier transform without inter-processor communications. In *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05.*, pages 307–311, March 2005. doi: 10.1109/SSST.2005.1460927.

[51] Jongsoo Park, Ganesh Bikshandi, Karthikeyan Vaidyanathan, Ping Tak Peter Tang, Pradeep Dubey, and Daehyun Kim. Tera-scale 1d fft with low-communication algorithm and intel® xeon phi™ coprocessors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY,

USA, 2013. Association for Computing Machinery. ISBN 9781450323789. doi: 10.1145/2503210.2503242. URL https://doi-org.proxy-um.researchport.umd.edu/10.1145/2503210.2503242.

[52] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[53] Apache Software Foundation. Hadoop. URL https://hadoop.apache.org.

[54] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD record*, 40(4):11–20, 2012.

[55] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[56] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Coded mapreduce. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 964–971. IEEE, 2015.

[57] Songze Li, Mohammad Ali Maddah-Ali, Qian Yu, and A Salman Avestimehr. A fundamental tradeoff between computation and communication in distributed computing. *IEEE Transactions on Information Theory*, 64(1):109–128, 2017.

[58] Songze Li, Sucha Supittayapornpong, Mohammad Ali Maddah-Ali, and Salman Avestimehr. Coded terasort. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 389–398. IEEE, 2017.

[59] A Yavuz Oruç. One-sided binary tree-crossbar switching for on-chip networks. In *2015 49th Annual Conference on Information Sciences and Systems (CISS)*, pages 1–5, 3 2015.

[60] A Yavuz Oruç. A self-routing on-chip network. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1229–1239, 2016.

[61] Ali Shatnawi, Yathrip AlZahouri, Mohammed A Shehab, Yaser Jararweh, and Mahmoud Al-Ayyoub. Toward a new approach for sorting extremely large data files in the big data era. *Cluster Computing*, 22(3):819–828, 2019.

[62] Ajitesh Srivastava, Ren Chen, Viktor K Prasanna, and Charalampos Chelmis. A hybrid design for high performance large-scale sorting on FPGA. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[63] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[64] Donald Ervin Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998. ISBN 9780321635785. URL https://books.google.com/books?id=cYULBAAAQBAJ.

[65] Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 2014.

[66] Leslie G Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3): 348–355, 1975.

[67] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[68] A Nico Habermann. Parallel neighbor sort. *Computer Science Report, Carnegie-Mellon University, Pittsburgh*, 1972.

[69] Ge Baudet, David Stevenson, et al. Optimal sorting algorithms for parallel computers. *IEEE Trans. Comput.*, 27(1):84–87, 1 1978. ISSN 0018-9340.

[70] Clark D Thompson and Hsiang Tsung Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, 1977.

[71] Franco P. Preparata. New parallel-sorting schemes. *IEEE Computer Architecture Letters*, 27(07):669–673, 1978.

[72] David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, (1):2–7, 1979.

[73] Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 255–263, 1986.

[74] Minze V Chien and A Yavuz Oruç. Adaptive binary sorting schemes and associated interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(6): 561–572, 1994.

[75] R Lin, S Olariu, JL Schwing, and J Zhang. Sorting in $O(1)$ time on an $n \times n$ reconfigurable mesh. In *Parallel Computing: From Theory to Sound Practice, Proceedings of the European Workshop on Parallel Computing*, pages 16–27. Citeseer, 1992.

[76] Richard P Brent and Hsiang T Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, 3 1982. ISSN 2326-3814. doi: 10.1109/TC.1982.1675982.

[77] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

[78] Ian Parberry and Georg Schnitger. Parallel computation with threshold functions. *Journal of Computer and System Sciences*, 36(3):278–302, 1988.

[79] Kai-Yeung Siu, Vwani P. Roychowdhury, and Thomas Kailath. Depth-size tradeoffs for neural computation. *IEEE Transactions on Computers*, (12):1402–1412, 1991.

[80] András Hajnal, Wolfgang Maass, Pavel Pudlák, Mario Szegedy, and György Turán. Threshold circuits of bounded depth. *Journal of Computer and System Sciences*, 46(2): 129–154, 1993.

[81] Ashok K Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984.

[82] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[83] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[84] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[85] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59 (11):56–65, 2016.

[86] Hazem M Bahig. A new constant-time parallel algorithm for merging. *The Journal of Supercomputing*, 75(2):968–983, 2019.

[87] Ahmad Salah, Kenli Li, Qing Liao, Mervat Hashem, Zhiyong Li, Anthony T Chronopoulos, and Albert Y Zomaya. A time-space efficient algorithm for parallel k-way in-place merging based on sequence partitioning and perfect shuffle. *ACM Transactions on Parallel Computing (TOPC)*, 7(2):1–23, 2020.

[88] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 1–9, New York, NY, USA, 1983. Association for Computing Machinery. ISBN 0897910990. doi: 10.1145/800061.808726. URL `https://doi.org/10.1145/800061.808726`.

[89] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, 2019.

[90] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500. techreport, November 2021. URL `https://www.top500.org/`. https://www.top500.org/.

[91] Lasse Natvig. Logarithmic time cost optimal parallel sorting is not yet fast in practice! In *Conference on High Performance Networking and Computing: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, volume 12, pages 486–494. Citeseer, March 1990.

[92] Jaeyoung Choi, Jack J Dongarra, L Susan Ostrouchov, Antoine P Petitet, David W Walker, and R Clint Whaley. Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[93] Françoise Chatelin. *Eigenvalues of Matrices: Revised Edition*. SIAM, 2012.

[94] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[95] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[96] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[97] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–41. IEEE, 2017.

[98] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[99] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4): 354–356, 1969.

[100] Shmuel Winograd. On multiplication of $2\times$ 2 matrices. *Linear algebra and its applications*, 4(4):381–388, 1971.

[101] Arnold Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[102] Don Coppersmith and Shmuel Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[103] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[104] Murat Cenk and M Anwar Hasan. On the arithmetic complexity of strassen-like matrix multiplications. *Journal of Symbolic Computation*, 80:484–501, 2017.

[105] Andrew James Stothers. On the complexity of matrix multiplication. 2010.

[106] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898, 2012.

[107] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303, 2014.

[108] Josh Alman and Virginia Vassilevska Williams. *A Refined Laser Method and Faster Matrix Multiplication*, pages 522–539. doi: 10.1137/1.9781611976465.32. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976465.32.

[109] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, 1969.

[110] G.C. Fox, I.G. Angus, and J.S. Kim. *Solving Problems on Concurrent Processors*. Number v. 2 in Prentice-Hall International editions. Prentice Hall, 1988. ISBN 9780138297145. URL https://books.google.com/books?id=MYSOswEACAAJ.

[111] Jaeyoung Choi, David W Walker, and Jack J Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[112] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[113] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–121. IEEE Computer Society, 1992.

[114] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 261–272. IEEE, 2013.

[115] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017 – 1026, 2004. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2004.03.021. URL http://www.sciencedirect.com/science/article/pii/S0743731504000437.

[116] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 193–204, New York, NY, USA, 2012. Association for

Computing Machinery. ISBN 9781450312134. doi: 10.1145/2312005.2312044. URL https://doi.org/10.1145/2312005.2312044.

[117] Jacob Scott, Olga Holtz, and Oded Schwartz. Matrix multiplication i/o-complexity by path routing. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 35–45, 2015.

[118] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356181. URL https://doi.org/10.1145/3295500.3356181.

[119] Tyler Michael Smith, Bradley Lowery, Julien Langou, and Robert A. van de Geijn. A tight i/o lower bound for matrix multiplication, 2019. URL https://arxiv.org/abs/1702.02017.

[120] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. Trade-offs between synchronization, communication, and computation in parallel linear algebra computations. *ACM Trans. Parallel Comput.*, 3(1), 1 2017. ISSN 2329-4949. doi: 10.1145/2897188. URL https://doi.org/10.1145/2897188.

[121] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. ACM*, 59(6), January 2013. ISSN 0004-5411. doi: 10.1145/2395116.2395121. URL https://doi.org/10.1145/2395116.2395121.

[122] Gianfranco Bilardi and Lorenzo De Stefani. The i/o complexity of strassen's matrix multiplication with recomputation. In *Workshop on Algorithms and Data Structures*, pages 181–192. Springer, 2017.

[123] Mohammad Mahdi Javanmard, Pramod Ganapathi, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. Toward efficient architecture-independent algorithms for dynamic programs. In *International Conference on High Performance Computing*, pages 143–164. Springer, 2019.

[124] Elaye Karstadt and Oded Schwartz. Matrix multiplication, a little faster. *Journal of the ACM (JACM)*, 67(1):1–31, 2020.

[125] Y. Umuroglu, L. Rasnayake, and M. Själander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 307–3077, 2018. doi: 10.1109/FPL.2018.00059.

[126] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002. doi: 10.1126/science.298.5594.824. URL https://www.science.org/doi/abs/10.1126/science.298.5594.824.

[127] Chidanand Apte, Bing Liu, Edwin P. D. Pednault, and Padhraic Smyth. Business applications of data mining. *Commun. ACM*, 45(8):49–53, 8 2002. ISSN 0001-0782. doi: 10.1145/545151.545178. URL https://doi.org/10.1145/545151.545178.

[128] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. doi: 10.1073/pnas.122653799. URL https://www.pnas.org/doi/abs/10.1073/pnas.122653799.

[129] Fan Chung and Linyuan Lu. *Complex graphs and networks*. Number 107. American Mathematical Soc., 2006.

[130] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 16–24, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581934. doi: 10.1145/1401890.1401898. URL https://doi.org/10.1145/1401890.1401898.

[131] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, page 623–632, Philadelphia, PA, United States, 2002. Society for Industrial and Applied Mathematics. ISBN 089871513X.

[132] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Trans. Knowl. Discov. Data*, 8(1), 2 2014. ISSN 1556-4681. doi: 10.1145/2556609. URL https://doi.org/10.1145/2556609.

[133] Jonathan W. Berry, Bruce Hendrickson, Randall A. LaViolette, and Cynthia A. Phillips. Tolerating the community detection resolution limit with edge weighting. *Phys. Rev. E*, 83:056119, 5 2011. doi: 10.1103/PhysRevE.83.056119. URL https://link.aps.org/doi/10.1103/PhysRevE.83.056119.

[134] Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8 (2):e1226, 2018. doi: https://doi.org/10.1002/widm.1226. URL https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1226.

[135] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538, 2013.

[136] Mahmudur Rahman and Mohammad Al Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, pages 127–133. IEEE, 2013.

[137] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814–818, 2005.

[138] Thomas Schank. Algorithmic aspects of triangle-based network analysis. 2007.

[139] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[140] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.

[141] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. Opt: A new framework for overlapped and parallel triangulation in large-scale graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 637–648, 2014.

[142] Andrew Lumsdaine, Luke Dalessandro, Kevin Deweese, Jesun Firoz, and Scott McMillan. Triangle counting with cyclic distributions. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2020.

[143] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[144] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015.

[145] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[146] Seher Acer, Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, and Ümit V Catalyürek. Scalable triangle counting on distributed-memory systems. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019.

[147] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, page 837–846, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584959. doi: 10.1145/1557019.1557111. URL `https://doi.org/10.1145/1557019.1557111`.

[148] Rasmus Pagh and Charalampos E Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[149] Roohollah Etemadi, Jianguo Lu, and Yung H Tsin. Efficient estimation of triangles in very large graphs. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1251–1260, 2016.

[150] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[151] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM international conference on data mining*, pages 10–18. SIAM, 2013.

[152] Jakub Tětek. Approximate triangle counting via sampling and fast matrix multiplication. *arXiv preprint arXiv:2104.08501*, 2021.

[153] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 539–548, 2013.

[154] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160. IEEE, 2015.

[155] Shumo Chu and James Cheng. Triangle listing in massive networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–32, 2012.

[156] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 325–336, 2013.

[157] Ilias Giechaskiel, George Panagopoulos, and Eiko Yoneki. Pdtl: Parallel and distributed triangle listing for massive graphs. In *2015 44th International Conference on Parallel Processing*, pages 370–379. IEEE, 2015.

[158] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Disttc: High performance distributed triangle counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[159] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[160] Terence Kelly. Programming workbench: Compressed sparse row format for representing graphs. *login Usenix Mag.*, 45(4), 2020.

[161] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, January 1987. ISSN 0004-5411. doi: 10.1145/7531.7532. URL `http://doi.acm.org/10.1145/7531.7532`.

[162] Yen-Cheng Chen and Wen-Tsuen Chen. Constant time sorting on reconfigurable meshes. *IEEE Transactions on Computers*, 43(6):749–751, 6 1994. ISSN 0018-9340.