Efficient Algorithms for a Mesh-Connected Computer with Additional Global Bandwidth

by

Yujie An

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2019

Doctoral Committee:

Professor Quentin F. Stout, Chair
Associate Professor Kevin J. Compton
Professor Seth Pettie
Professor Martin J. Strauss

Yujie An

anyujie@umich.edu

ORCID iD: 0000-0002-2038-8992

©Yujie An 2019

Acknowledgments

My researches are done with the help of many people. First I would like to thank my advisor, Quentin F. Stout, who introduced me to most of the topics discussed here, and helped me throughout my Ph.D. career at University of Michigan. I would also like to thank my thesis committee members, Kevin J. Compton, Seth Pettie and Martin J. Strauss, for their invaluable advice during the dissertation process. To my parents, thank you very much for your long-term support. All my achievements cannot be accomplished without your loves. To all my best friends, thank you very much for keeping me happy during my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

VLSI Very Large Scale Integration

PRAM Parallel Random Access Machine

ER Exclusive Read

EW Exclusive Write

CR Concurrent Read

CW Concurrent Write

ADT Abstract Data Type

SIMD Single Instruction Multiple Data

RAR Random Access Read

RAW Random Access Write

SRAR Sparse Random Access Read

SRAW Sparse Random Access Write

MST Minimum Spanning Tree

MSF Minimum Spanning Forest

DFT discrete Fourier transform

FFT fast Fourier transform

KMP Knuth-Morris-Pratt

HPC High Performance Computing

ABSTRACT

This thesis shows that adding additional global bandwidths to a mesh-connected computer can greatly improve the performance. The goal of this project is to design algorithms for mesh-connected computers augmented with limited global bandwidth, so that we can further enhance our understanding of the parallel/serial nature of the problems on evolving parallel architectures. We do this by first solving several problems associated with fundamental data movement, then summarize ways to resolve different situations one may observe in data movement in parallel computing. This can help us to understand whether the problem is easily parallelizable on different parallel models. We give efficient algorithms to solve several fundamental problems, which include sorting, counting, fast Fourier transform, finding a minimum spanning tree, finding a convex hull, etc. We show that adding a small amount of global bandwidth makes a practical design that combines aspects of mesh and fully connected models to achieve the benefits of each. Most of the algorithms are optimal. For future work, we believe that algorithms with peak-power constrains can make our model well adapted to the recent architectures in high performance computing.

CHAPTER 1

Introduction

In recent years, advances in Very Large Scale Integration (VLSI) technology have provided increasingly powerful cores consist of many thousands and potentially millions of processors. The layouts of cores and processors are very different from machine to machine, and they are changing very rapidly. For example, the PSC Bridges supercomputer has 28 cores on each node [1], and communication between nodes are expensive. This is not true for the same supercomputer system before 2016, not to say many other supercomputers.

From a programmer's prospective, it would be ideal to write programs for a Parallel Random Access Machine (PRAM). In a PRAM, all processors have a small local memory but can communicate with a large shared memory in constant time. The power of such global communication can significantly reduce the program's complexity, though algorithms can still be extremely hard to implement. It is a quite powerful theoretical model that is capable of solving many problems in poly-logarithmic time (i.e., $\mathcal{O}(\log^k n)$ time for some constant k) [2, 3, 4, 5]. However, the PRAM is not practical and cannot be scaled to large numbers of processors. Further, many of the optimal (in $\mathcal{O}$-notation) algorithms for it are decidedly impractical, with the most conspicuous example being sorting [6].

On the other hand, the mesh-connected computer has occurred throughout parallel computing history and is a practical, highly scalable model that has been implemented many times [7, 8, 9]. Further, many more meshes are being planned [10, 11, 12, 13, 14]. One example of a real regular mesh-connected computer is the NEO chip [15]. The basic model assumes a $n^{\frac{1}{2}} \times n^{\frac{1}{2}}$ 2-dimensional mesh of processors, where each processor can communicate only with its 4 immediate neighbors. Extensive research has been done on this model [16, 17, 18, 19, 20] and almost any algorithm takes $\Omega(n^{\frac{1}{2}})$ time since this is its the communication diameter and bisection bandwidth. Even if the mesh is modified to be a 2-dimensional torus, the communication diameter and bisection bandwidth are only reduced by a factor of 2.

In this thesis we present algorithms on a new model: a mesh-connected computer with additional global bandwidth. It combines the realistic scaling of the mesh-connected com-

puter with the speed of the theoretical, but impractical, PRAM. Only a small amount of global bandwidth is added, and the communication channels are limited to exclusive read (i.e., point-to-point) unless otherwise noted. Here we focus on the use of such bandwidth, without a specific implementation. Options include standard communication networks, buses, shared memory, cache coherence, etc.

More importantly, the new model provides a way to think about parallel algorithm structures, which is not new but always ignored previously. Many parallel computers can be abstracted as some local computational components plus some global communication power. Though the local components may not always be a mesh, our algorithms can still provide a generic way to think and design algorithms on such abstract models. With the help of the ideas inspired by our model, one can easily design new algorithms on a new supercomputer as long as it supports some kind of local and global communication, regardless of how many cores are on each node, and what these nodes' layout is.

The goal of this research is to solve fundamental problems on this new model. These solutions should be as general as possible, so that they can be easily adapted to other similar models. Chapter 2 gives a more detailed definition to our model, and also some related works. Chapter 3 provides several basic operations that will serve as building blocks in later algorithms. In Chapter 4, we solve problems related to ordered data. We give algorithms to sort, find the median, and construct the histogram. In Chapter 5, we efficiently implement the fast Fourier transform on our model, and give several applications of it. In Chapter 6, we give algorithms to solve fundamental graph problems such as finding a minimum spanning tree. Many other graph problems can be solved after a spanning tree is found. In Chapter 7, we give algorithms to solve computational geometry problems, such as finding a convex hull, finding all-nearest neighbors, finding a pair of intersecting line segments, etc. In Chapter 8, we switch to a coarse-grained model, and show that solving problems on a coarse-grained model is much more difficult than solving the same problem on a fine-grained model. Finally in Chapter 9, we give the conclusion and future works.

CHAPTER 2

Preliminaries

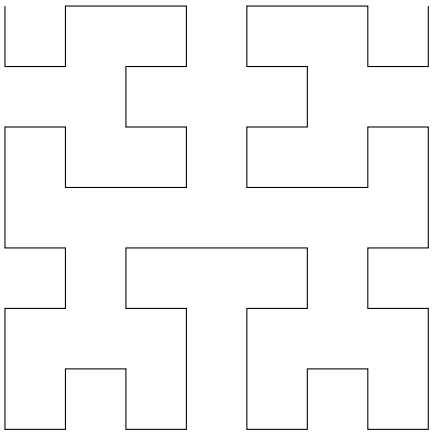
In this Chapter, notations and general parallel models of computation are discussed. Throughout this thesis, $\mathcal{O}$-notation is used. Whenever an ϵ appears in a theorem, the implied constants inside the $\mathcal{O}$-notation are a function of ϵ. This applies for all algorithms in all chapters.

2.1 Our Model

To simplify exposition, let n be a power of 4. Otherwise, one can use the closest power of 4 that is smaller than n to be the mesh size. In this way, the algorithm will have a constant factor overhead. We start with the standard mesh model of n processors on a $n^{\frac{1}{2}} \times n^{\frac{1}{2}}$ grid, where each processor is only connected to its immediate neighbors. In the *fine-grained* model, each processor has a fixed number of words of memory, each with $\Theta(\log n)$ bits, which are sufficient to store its coordinates and a constant number of other values. In the *coarse-grained* model, each processor has a much bigger size memory, each with $\Theta(K \log n)$ bits, where K could be anything, not necessarily a constant. The processors are numbered $0 \ldots n-1$ using space-filling curve ordering. Throughout, "space-filling curve ordering" means the Hilbert space-filling curve ordering (see Figure 2.1).

We add global communication channels above the mesh, creating the mesh-connected computer with additional bandwidth. In the fine-grained model, we use MCC$(n, S(n))$ to represent a computer with base mesh of size n and $\Theta(S(n))$ global communication channels. In the coarse-grained model, we use MCC$(n, p, S(p))$ to represent a computer with base mesh of size p and $\Theta(S(p))$ global communication channels, with input size n. Note that the coarse-grained representation can cover all cases of the fine-grained model with MCC$(n, n, S(n))$. We keep the fine-grained model as a different case because it can better show how to do communication.

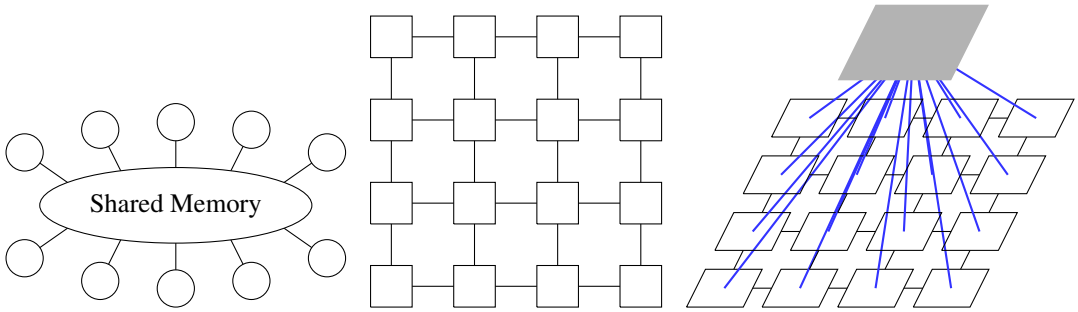
To make this practical, we want the size of the global communication channels to be



0	3	4	5	58	59	60	63
1	2	7	6	57	56	61	62
14	13	8	9	54	55	50	49
15	12	11	10	53	52	51	48
16	17	30	31	32	33	46	47
19	18	29	28	35	34	45	44
20	23	24	27	36	39	40	43
21	22	25	26	37	38	41	42

(a) Space-filling curve (b) Numbering

Figure 2.1: Space-filling curve ordering



(a) PRAM (b) Mesh (c) Our model

Figure 2.2: Mesh with additional global bandwidth

quite small, and thus $S(n) = n^{\epsilon}$ with $0 \leq \epsilon \leq \frac{1}{2}$ is desired, though our algorithms work for all $0 \leq \epsilon < 1$. But unfortunately, for problems where bisection bandwidth serves as a lower bound, the worst case time cannot be improved when $S(n) = \mathcal{O}(n^{\frac{1}{2}})$. Sorting is one of the most obvious problems limited by bisection bandwidth, and it will be used repeatedly. Thus we use $S(n) = n^{\frac{1}{2}+\epsilon}$ in those problems, with $0 < \epsilon < \frac{1}{2}$. The communication channels are numbered $0 \ldots S(n) - 1$ (see Figure 2.2). We first consider Exclusive Read (ER) Exclusive Write (EW) channels. That is, for each channel, only two processors at a time can use it to communicate. This is denoted by ER MCC$(n, S(n))$.

One way to extend the model is to give channels the power of Concurrent Read (CR), that is, to give them the functionality of a bus. Thus a CR MCC$(n, S(n))$ can do broadcast in constant time, which can result in significant speedup in algorithms for problems such as finding a spanning tree, where broadcast is used quite often. These algorithms are studied in

Section 6.3. Concurrent Write (CW) options, such as collision detection, are not examined in this thesis.

ER and CR are very different when dealing with communication problems. Even simple pairwise communication can have quite different timing on ER vs. CR. Consider the following problem: processor 0 determines that it requires a value from processor i, while processors other than 0, including i, do not initially know i. A broadcast by processor 0 can be used to solve this problem. The ER model is much weaker than the CR one, since broadcast takes $\Theta(1)$ time on a CR model and $\omega(1)$ time on an ER model. An optimal ER broadcast algorithm and the proof of its optimality is given in Chapter 3.

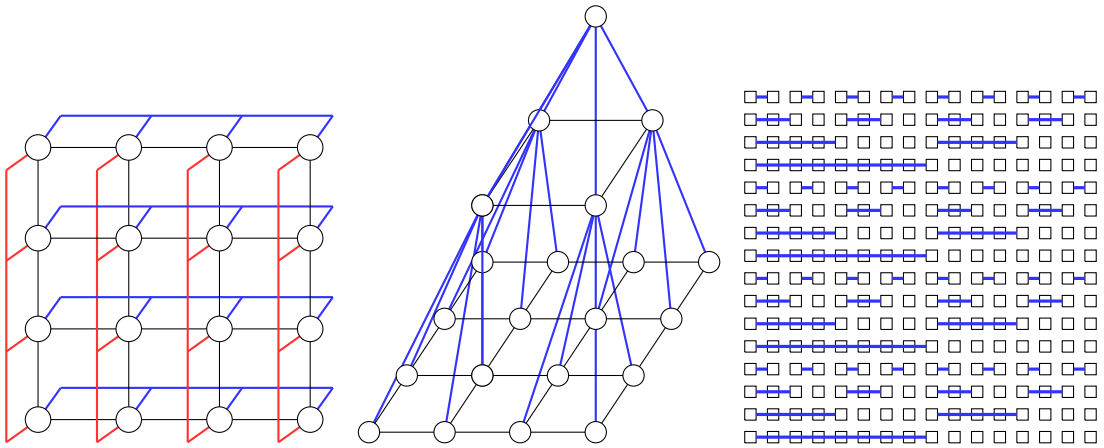
This makes a significant difference in algorithms where processors can determine which processor they need information from, but cannot easily determine where they need to send information to until they get a request. This is a general problem in distributed memory systems. We call the time difference between sending to a target processor, where the sender and receiver know which channel is being used, vs. broadcasting a request and then receiving the value the *communication gap*. A CR MCC is more powerful in resolving the communication gap than an ER MCC. How to resolve the communication gap completely using different models is an interesting topic.

Throughout, it is straightforward for a processor to determine in constant time if it is sending or receiving a message from a channel, and the channel being used. The details are described in algorithms.

2.2 Related Work

The first model that is highly relevant to our model is the PRAM. One can see that when $S(n) = \Theta(n)$, our model is essentially a PRAM. The base mesh connections cannot help since the number of global connections is already $\Theta(n)$. Also, any algorithm on an EREW PRAM that takes $\Theta(T)$ time can be stepwise simulated on our model in $\Theta(n/S(n) \cdot T)$ time. What's more, since one can always use a sort to simulate one step of concurrent read or concurrent write, any CRCW PRAM algorithm that takes $\Theta(T)$ time can also be stepwise simulated on our model in $\Theta(n/S(n) \cdot T)$ time. This is so because sort can be done in $\mathcal{O}(n/S(n))$ time (see Section 4.1). Using this fact, many problems can be trivially solved on our model in $\tilde{\mathcal{O}}(n/S(n))$ time with one or two logarithmic factors. $\tilde{\mathcal{O}}$ means that there may be additional log factors. The problems include sort [2], finding a minimum spanning tree [21], finding the convex hull [22], finding the Voronoi diagram [23], etc.

The mesh-connected computer is another model that serves as a building block of our model. When $S(n)$ is small, and the problem requires a bisection bandwidth bigger



(a) Mesh with row and column buses (b) Pyramid (c) Optical mesh of trees

Figure 2.3: Mesh augmented by global communication power

than $S(n)$, our model is essentially a mesh since the global communications cannot help. Though in general stepwise simulating a mesh algorithm on our model will not be faster since it will still give an $\Omega(n^{\frac{1}{2}})$ run time, many mesh algorithms are very inspirational, and can serve as the base case in our algorithms. Most of these algorithms can be found in [19].

In order to break the lower bound due to the diameter, many researchers augment the mesh communication structure with additional communication capabilities to create new models. These models include, but are not limited to, shared memory mesh, mesh computer with buses, reconfigurable mesh, pyramid computer, and mesh computer with optical connections (see Figure 2.3).

The mesh computer with buses is a well-known model that augments the mesh communication structure with additional communication capabilities using buses. Several researchers use row and column buses [24, 25, 26, 27, 28], while some others use global buses [29, 30] or reconfigurable buses [31, 32, 33, 34]. The major concern with buses is that they require concurrent reads, which is not always practical.

The pyramid computer is another well-known model that augments the mesh with a pyramid-like structure of processors. It has long been proposed for performing high-speed low-level image processing [35, 36, 37, 38]. A variety of such machines have been built [39, 40]. Though limited by its bisection bandwidth, many problems can still be solved significantly faster than on a mesh, e.g., finding a minimum spanning forest [41].

The mesh computer with optical connections is a very recent model published in [42]. It adds optical connections on the mesh to overcome the long communication diameter. Their results [43] provide time-power trade-off for many problems like sorting, finding a

minimum spanning tree, etc. The time-power trade-off exist not only in their model, but also in the standard mesh as well [44], though it is more natural to consider such a trade-off when you have global communication powers such like optical connections.

CHAPTER 3

Basic Operations

In designing serial algorithms, one major concern is the proper choice and implementation of data structures and their associated algorithms. A well-known design pattern is to use an Abstract Data Type (ADT), which contains an abstract data structure (e.g., list, tree, queue) and a set of basic operations to be performed on the data in the structure. In this thesis, abstract data movement operations are viewed as the parallel analogue of ADTs. That is, parallel algorithms are expressed in terms of fundamental data movement operations without worrying about their implementation or the specific interconnection of the processors. This technology was used implicitly in a lot of previous work, and was explicitly defined in [19].

In this chapter, we provide algorithms for doing basic data movement operations on mesh-connected computers with additional global bandwidth. The operations include broadcast, reduction, scan, rotation and sparse random access read/write. We also provide segmented, row and column, and block versions of some of these operations.

3.1 Broadcast, Reduction and Scan

In broadcast, one processor i starts with a value v and all other processors receive the value at the end. In reduction, each processor i starts with a value v_i, and at the end processor 0 has the value of $\oplus_{i=0}^{n-1} v_i$, where $\oplus$ is a binary associative operator such as xor, max or sum. We assume that $\oplus$ of two values can be computed in constant time. In scan, each processor i starts with a value v_i, and at the end each processor k has the value of $s_k = \oplus_{i=0}^{k} v_i$ for $0 \leq k \leq n-1$. Theorem 3.1.1 shows that these operations can be done efficiently.

Theorem 3.1.1. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, broadcast, reduction and scan can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.*

Proof. Only the algorithm for scan is given here. Broadcast and reduction can be implemented as special cases of scan (or scan in reverse order) and can be solved using a

simplified version of this algorithm. The algorithm takes four steps.

1. Let $B = n^{\frac{2}{3}(1-\epsilon)}$. Divide the mesh into n/B blocks, each of size $B^{\frac{1}{2}} \times B^{\frac{1}{2}}$. Do a scan in each block using a standard mesh algorithm.

2. In each block, only a single value, the scan value of the processor with the largest index, needs to be combined with those in the other blocks. Using channels, move these n/B values to a subset of the blocks. This step works as follow. There are $n^{1-\epsilon}/B$ rounds where in each round, simultaneously for n^{ϵ} pairs, one processor writes its value to the channel, then another processor reads the value. At the end of this step the remaining values have been reorganized such that they are saved in n/B^2 different blocks, each of size B. Solve these blocks' scan problems using a standard mesh algorithm.

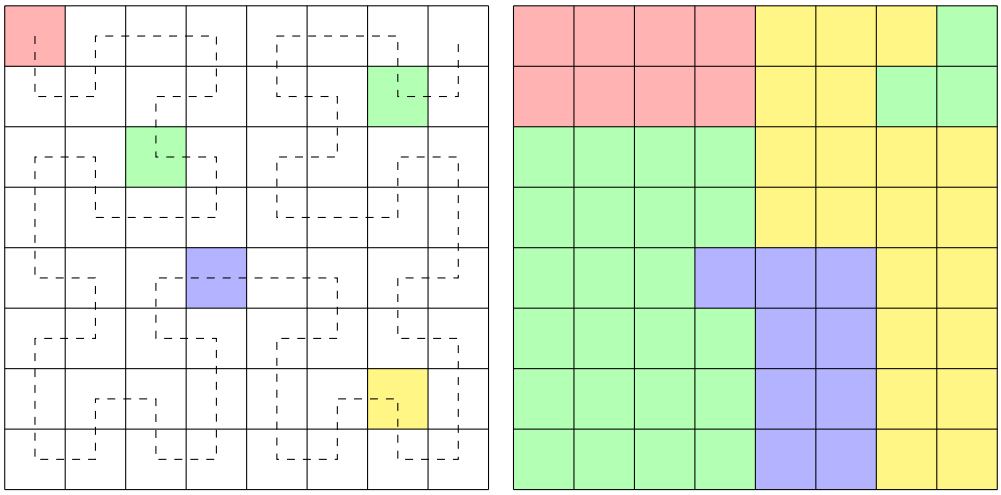
3. Repeatedly do step 2 for t times, where $t = \lceil \log_B n \rceil$. At this point only a single value remains. Note that we can let each round use different blocks to save data values, and therefore each processor in the end still contains only a constant number of data values.

4. Do step 3 in reverse order, i.e. top down, to propagate values back. With the help of the previously calculated values, each processor can get its scan value. Here we use the last step as an example. In the last step, we have n/B blocks. In each block the processor p with the smallest index has the scan value $s_{p-1} = \oplus_{i=0}^{p-1} v_i$ of processor $p-1$, which comes from the previous iteration. That is, processor 0 has 0, processor B has s_{B-1}, processor $2B$ has s_{2B-1}, and etc. Broadcast such scan values in the block, and the final scan value for each processor can be calculated in constant time, since the scan value using data values only inside the block has been calculated in step 1.

Step 1 takes $\Theta(B^{\frac{1}{2}})$ time. The ith iteration for step 2 and 3 takes $\Theta(n^{1-\epsilon}/B^i)$ time to move the data, and $\Theta(B^{\frac{1}{2}})$ time to do the scan in the block. Note that moving time decreases geometrically as the iteration goes on, but the scan time within a block stays the same. Step 4 takes the same time as step 3. Therefore the total time is $\Theta(B^{\frac{1}{2}} + n^{1-\epsilon}/B + B^{\frac{1}{2}}t)$. We know that $B = n^{\frac{2}{3}(1-\epsilon)}$, thus we have $t = \Theta(1)$ and the time is as claimed. □

Theorem 3.1.1 implies that when the number of channels is $\Theta(1)$, the time is $\Theta(n^{\frac{1}{3}})$, and when it is $\Theta(n^{\frac{1}{2}})$, the time is $\Theta(n^{\frac{1}{6}})$.

There are *segmented* versions of these operations where the processors are divided into segments $0 \ldots i_1 - 1$, $i_1 \ldots i_2 - 1$, $\ldots$, $i_k \ldots n-1$ for some $k < n$ and the operations are



(a) Before (b) After

Figure 3.1: Segmented broadcast

performed concurrently on the segments, independently. For example, the value broadcasted on the first segment might be different than that on the second (see Figure 3.1). These operations take the same time as the single global version, until all of the segments are no larger than the base mesh case, where the time is the square root of the base mesh's size. One could do better if most of the segments are very small, but such refinement is not needed here.

Theorem 3.1.2. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, segmented broadcast, reduction and scan can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.* □

Using segments does not reduce the time until the base case size is reached because while the segments may be smaller than the entire MCC, the number of channels per segment is also decreasing. Note that on an ER MCC(a, b), the segmented broadcast, reduction and scan time is always $\Theta((a/b)^{\frac{1}{3}})$, regardless of a and b.

There are also *row and column* versions of these operations. That is, each row or column does reduction, broadcast or scan independently. Theorem 3.1.3 shows that we can efficiently do these operations on different rows and columns independently in parallel. This method is widely used in graph algorithms with adjacency matrix input.

Theorem 3.1.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, broadcast, reduction and scan on rows and columns can be done in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time.*

Proof. We only give the algorithm for scan for rows here. The algorithm follows the structure in Theorem 3.1.1. Divide the mesh into blocks of size $B = n^{1-\epsilon}$. Solve the scan in

each block's rows. Then there are $n/B^{\frac{1}{2}}$ values remain, move them to a sub-mesh of size $n/B^{\frac{1}{2}}$. Note that the problem now is exactly the same as before, i.e., do a scan in each row, though the problem size is smaller. We keep dividing the sub-mesh into blocks of size B, and recursively solve the problem until the sub-mesh has size no bigger than B. Finally propagate the final results of the $n/B^{\frac{1}{2}}$ values back to their original block, and calculate the final scan values of all other values accordingly.

Solving in each block takes $\Theta(B^{\frac{1}{2}})$ time, and moving takes $\Theta(n/B^{\frac{1}{2}}/n^{\epsilon})$ time. The recursive calls have a constant depth depending upon ϵ. Thus all steps finish in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time. Note that this is slower than the results in Theorem 3.1.1 and Theorem 3.1.2. This is because we cannot exploit the properties of the space-filling curve. □

3.2 Rotation

In many problems, we need to compare two sets of values pairwise. The rotation operation does the following: given a set V of n initial values and an additional set T of m values, each stored no more than one per processor, the operation pairwise compares values in V and T. That is, for any $v \in V$ and $t \in T$, at some point in the rotation, v and t are in the same processor. This operation can be easily done by sequentially broadcasting all values in T. However, a more efficient algorithm exists.

Theorem 3.2.1. *On an ER MCC(n, n^{ϵ}) with $0 \leq \epsilon < 1$, given n initial values and additional $m \leq n$ values, rotation can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \cdot m^{\frac{1}{3}} + m)$ time.*

Proof. First assume that $m = \mathcal{O}(n^{\frac{1}{2}})$. Otherwise, we can use a standard mesh algorithm [19] to finish the problem in $\Theta(m)$ time, and the channels cannot help. Note that in this case, $\Theta(n^{\frac{1}{3}(1-\epsilon)} \cdot m^{\frac{1}{3}} + m) = \Theta(m)$.

The idea is to use divide and conquer. Divide the mesh into blocks of size $B \geq m$. Copy all m values into all blocks, and finish the problem inside each block using a standard mesh algorithm. There are n/B blocks, and making one copy of m values takes m/n^{ϵ} time. Thus the total time is $\Theta(\frac{n}{B} \cdot \frac{m}{n^{\epsilon}} + B^{\frac{1}{2}} + m)$. To minimize the time, take $B = n^{\frac{2}{3}(1-\epsilon)} \cdot m^{\frac{2}{3}}$, and we have run time as claimed.

There are two issues in the algorithm described above, but both of them can be fixed. First, we choose B to minimize the time, but we still need to guarantee that $B \geq m$. Otherwise the logic is incorrect since we cannot copy m things into a block smaller than m. This requires $n^{\frac{2}{3}(1-\epsilon)} \cdot m^{\frac{2}{3}} \geq m$, which simplifies to $m \leq n^{2(1-\epsilon)}$. If this condition holds, the algorithm above solves the problem. Otherwise we change the parameter B. Let

$B = m$ in such case, and the time becomes $\Theta(n^{1-\epsilon} + m^{\frac{1}{2}} + m)$. Since $m > n^{2(1-\epsilon)}$, the time is $\Theta(m)$.

Second, the algorithm above is designed for $m \geq n^{\epsilon}$. If this is not true, then we have $m < n^{\epsilon}$. In this case, we have $n/B \geq n^{\epsilon}/m$, because it requires $n^{1-(\frac{2}{3}-\frac{2}{3}\epsilon)}/m^{\frac{2}{3}} \geq n^{\epsilon}/m$, which simplifies to $m \geq n^{\epsilon-1} = \mathcal{O}(1)$. Therefore the $\Theta(\frac{n}{B} \cdot \frac{m}{n^{\epsilon}})$ time for data movement is still correct, because it equals $\Theta(\frac{n}{B}/\frac{n^{\epsilon}}{m})$, which is the time for the correct logic in this case. Note that we also need $\Theta(\log \frac{n^{\epsilon}}{m})$ time to copy the m values to fill the channels, but this term never dominates. □

In some cases, we need to copy m values into n/m non-overlapping blocks of size m, without rotating them. We call this operation the *block* version of broadcast. Using the same algorithm as Theorem 3.2.1, Corollary 3.2.2 shows that block broadcast can be done efficiently. Note that we can also choose the block size to be not equal to m. However, there are too many cases there. It is better to discuss them case by case when they are used in other algorithms.

Corollary 3.2.2. *On an ER MCC(n, n^{ϵ}) with $0 \leq \epsilon < 1$, given $m \leq n$ values, block broadcast can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \cdot m^{\frac{1}{3}})$ time if $T = \mathcal{O}(n^{2(1-\epsilon)})$, and $\Theta(n^{1-\epsilon})$ time otherwise.* □

3.3 Sparse Random Access Read/Write

Random access problems for Single Instruction Multiple Data (SIMD) parallel computers have been widely studied. The mesh version of many such problems were solved efficiently by Nassimi and Sahni [45]. Two versions of this problem, Random Access Read (RAR) and Random Access Write (RAW), are considered.

Each processor initially has two values: a data value v_i and an address value p_i, which specifies a processor to read from or write to. After random access read, v_{p_i} should be saved in processor i; whereas after random access write, v_i should be written to processor p_i. Note that many processors may write to the same position. There are two ways to resolve such cases. First is that the processor that reads the values must read them all before determining which value to accept. Second is that a selection preference is given beforehand to determine which one should be sent.

A random access read/write is called *sparse* if the number of values that need to be sent is small. The most common definition is as follow: initially all processors have v_i's, but only $\mathcal{O}(n^{\frac{1}{2}})$ of them have p_i's. In such case, we refer these operations as Sparse Random Access Read (SRAR) and Sparse Random Access Write (SRAW). Algorithms for both

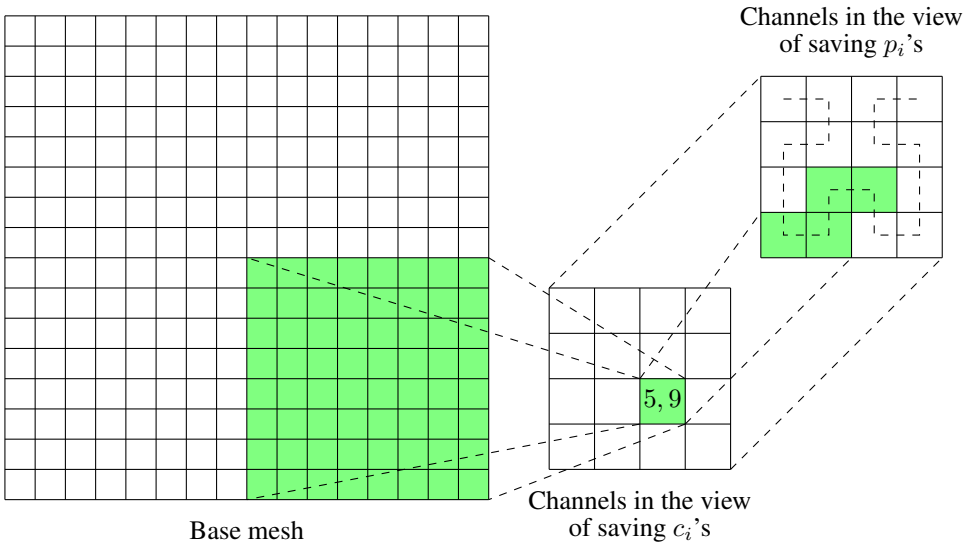


Figure 3.2: Data requesting operation with $n = 256$ and $\epsilon = 1/2$; after step 3, channel $n^\epsilon/2$ saves the range of channels that saves p_i in range $[n/2, 3n/4)$

SRAR and SRAW on mesh-connected computer with buses were developed in [26], but a sparsity condition was required in order to finish the algorithm in $\Theta(n^{\frac{1}{6}})$ time.

In our model, we slightly modify the definition to make it a better fit to our model. On an ER MCC(n, n^ϵ), a random access read/write is called sparse if the number of values that need to be sent is $\mathcal{O}(n^\epsilon)$. Our algorithms do not require any sparsity condition as in [26]. Many terminologies in our algorithm, for example "concentrate" and "distribute", follows from the standard mesh algorithm in [45]. Note that modifications can be made to our algorithms to make them work when the input is not sparse, i.e., the number of values that need to be sent is $\omega(n^\epsilon)$. However, such cases are not as interesting since the tedious data movement dominates the run time.

Theorem 3.3.1. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, sparse random access read can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.*

We first need to prove that the data requesting operation in Lemma 3.3.2 can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. It is used in both SRAR and SRAW. Note that throughout the algorithm, we will say that some values are "contained" in the channels. This can be done with the help of one processor per channel, so that from the other processors' point of view, the values are saved in the channels.

Lemma 3.3.2. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given that each channel i contains a processor ID p_i, this value needs to be replaced by the corresponding processor's data*

value v_{p_i}. Different channels may have same p_i. This operation can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

Proof. If all processors know whether they should write their v_i to the channels, and also the positions, this operation can be done in $\Theta(1)$ time if all p_i's in the channels are different. Thus the hardest part is to determine how each processor knows if its value is requested.

The algorithm takes 12 steps.

1. Sort the values in the channels using bitonic sort, and keep the original position of each value along with the value itself when sorting. We can use any n^ϵ processors to help the channels to do sorting. This step takes $\Theta(\log^2 n)$ time.

2. Concentrate the values, and thus the first K channels save K different p_i's if there are this many unique p_i's. The range of the channel positions these p_i's were previously saved is also kept in these K channels. For example, if the sorted channel values are $(1, 1, 2, 2, 3)$, they become $(1, 2, 3, -, -)$, and also $((0, 1), (2, 3), (4, 4), -, -)$, which represents the interval of positions that originally save these values. We save these values to help propagating back in later steps. Also, channel 0 knows K. This step can be done by a scan and sort, and takes $\Theta(\log n)$ time.

3. Processor 0 reads channel 0, and then it knows K. Do binary search 3 times to find channel positions c_1, c_2, c_3 that satisfy the following conditions: the p_i's saved between channels 0 and $c_1 - 1$ are greater than or equal to 0 and smaller than $n/4$, the p_i's saved between channels c_1 and $c_2 - 1$ are greater than or equal to $n/4$ and smaller than $n/2$, etc. Save all three values to channels $0, n^\epsilon/4, 2n^\epsilon/4$ and $3n^\epsilon/4$. This step takes $\Theta(\log n)$ time.

   Figure 3.2 shows an example of this step. Each channel will hold a constant number of values. There are two views of the channels, one saves all p_i's, while the other saves all necessary c_i's. These two views are quite independent, though c_i's depend on p_i's. Also note that c_i's are updated in each of the following few steps, but p_i's are never updated after step 2, until the very end.

4. Processors $0, n/4, 2n/4$ and $3n/4$ read channels $0, n^\epsilon/4, 2n^\epsilon/4$ and $3n^\epsilon/4$ respectively, and do a similar process as before. We will show what processor $n/4$ does as an example. Do binary searches between channel c_1 and $c_2 - 1$ three times, and find three new positions c_1', c_2', c_3'. Similar to before, the numbers saved between channels c_1 and $c_1' - 1$ are greater than or equal to $n/4$ and smaller than $n/4 + n/16 - 1$, the numbers saved between channels c_1' and $c_2' - 1$ are greater than or equal to $n/4 + n/16$

and smaller than $n/4 + 2n/16 - 1$, etc. Save all these three values to channels $n^\epsilon/4, n^\epsilon/4 + n^\epsilon/16, n^\epsilon/4 + 2n^\epsilon/16$ and $n^\epsilon/4 + 3n^\epsilon/16$. This step takes $\Theta(\log n)$ time.

5. Do step 4 in hierarchy $\lceil \log_4 n^\epsilon \rceil$ times, and now all channels are filled with the latest position numbers. Note that after a new iteration starts, the position numbers in the previous iterations are useless, and can be deleted. Thus we still only need $\Theta(\log n)$ bits capacity in each channel. This step takes $\Theta(\log^2 n)$ time in total for at most $\log n$ binary searches. Note that we can also divide the mesh into 2 parts instead of 4 in each iteration, and the run time will not be affected. We choose 4 to make sure that each sub-mesh is a square, instead of a rectangle.

6. After step 5, the mesh is divided into n^ϵ blocks, and each block has size $n^{\frac{1}{2}(1-\epsilon)} \times n^{\frac{1}{2}(1-\epsilon)}$. For each block, one of the following three cases must happen.

   - The values saved in this block are not requested.
   - At least one value is requested. The requested processor numbers are saved in some channels through c_1 and $c_2 - 1$, and $c_2 - c_1 = \Theta(1)$.
   - Same as the previous one, but $c_2 - c_1 = \omega(1)$.

   Note that the top left processor in each block can determine which case happens in its block in $\Theta(1)$ time by reading the corresponding channel associated with the block.

7. Broadcast which case each block is in inside the block in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time, with the help of one channel per block. For blocks in case 1, they are done now.

8. For blocks in case 2, using only channels through c_1 and $c_2 - 1$, broadcast all p_i's in the range. Processors with ID equals to the p_i's, write their values to the corresponding channels, and these blocks are done.

9. For blocks in case 3, keep dividing the channels through c_1 and $c_2 - 1$ using a similar method as described in step 4, until all positions in c_1 and $c_2 - 1$ are filled again. We can easily modify the algorithm to make it work if $c_2 - c_1$ is not a power of 4. Note that in order to synchronize, one may want each binary search to take exactly $\lceil \log n \rceil$ time. This step takes $\Theta(\log^2 n)$ time.

10. After step 9, all blocks in case 3 are now divided into some smaller sub-blocks, whose sizes depend upon $c_2 - c_1$. Now each sub-block is in the situation that is almost the same as described in step 6. Do, in hierarchy, a similar process as in step 6 through

10, until finally all sub-blocks have a constant size, or all sub-blocks are in case 1 or 2. This step takes no more than $\mathcal{O}(\log^2 n + n^{\frac{1}{3}(1-\epsilon)})$ time, because dividing blocks in case 3 takes $\mathcal{O}(\log^2 n)$ time in total, and solving case 2 takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time for the first time, and the time decreases geometrically each time the algorithm falls into case 3.

11. Now first K positions in the channels get the values required. Distribute these values to all channels according to the range information saved in step 2. This is the reverse of step 2, and takes $\Theta(\log n)$ time.

12. Sort the values in the channels again according to their original positions. This is the reverse of step 1, and takes $\Theta(\log n)$ time.

As we can see, the whole algorithm finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. □

With the help of Lemma 3.3.2, we are now able to prove Theorem 3.3.1.

Proof. Do a scan so that each processor that requests a value knows the number of processors that also request a value and have a processor ID smaller then it. Define this number to be the request ID of this processor.

Each processor that requests a value writes its p_i to the channel according to its request ID. Since processors are modifying different channel positions, they can do it in parallel. Then do the operation in Lemma 3.3.2 to collect the values. All processors that request a value then read the channels in parallel according to their request IDs, and the problem is solved. All step finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. □

Theorem 3.3.3 shows that sparse random access write can be solved basically in the same time, though there is one more subtle case that needs to be taken care of. Note that the selection preference needs to be reasonable. Otherwise one can choose a preference that is impossible to calculate in polynomial time, or even undecidable. A binary associative and commutative operation that can be done in constant time is preferred.

Theorem 3.3.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, sparse random access write can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + S)$ time, where S is the maximum number of values that one processor receives. Furthermore, if a selection preference is given beforehand, e.g., maximum value remains, SRAW can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.* □

3.4 Optimality

All operations mentioned in this Chapter are worst case optimal. The proof of optimality uses the same idea for all algorithms. Thus we give a generic proof here.

To briefly show the idea, let's first do Theorem 3.1.1 with $\epsilon = 0$. Divide the mesh into $n^{\frac{1}{3}}/4$ blocks, where each has size $2n^{\frac{1}{3}} \times 2n^{\frac{1}{3}}$. Consider broadcast operation. For the processor at the center of each block, to receive the broadcast value in $o(n^{\frac{1}{3}})$ time, the value must be sent using channels. Therefore, at least $n^{\frac{1}{3}}/4$ time is required for the value to be sent using channels, resulting in an $\Omega(n^{\frac{1}{3}})$ time for data movement. Thus $o(n^{\frac{1}{3}})$ total time is impossible, and the lower bound is $\Theta(n^{\frac{1}{3}})$.

All other algorithms' optimality, except rotation, can be proved in a similar way. In general, a way to prove a lower bound of $\Omega(T)$ is to show that every block of size $T \times T$ has at least $\Theta(B)$ information that must be combined with information from the other blocks. This gives $\Omega(n/T^2 \cdot B)$ information that must be moved by at least distance $\Theta(T)$. Moving this via the mesh establishes the lower bound, therefore it must be moved via the channels. There are only $\Theta(n^\epsilon)$ channels, hence if the information size is also $\Omega(n^\epsilon \cdot T)$ then the movement takes $\Omega(T)$ time.

The proof for rotation is slightly different. Using the same logic above, one can show that Corollary 3.2.2 is optimal, and thus the $\Theta(n^{\frac{1}{3}(1-\epsilon)} \cdot m^{\frac{1}{3}})$ part in Theorem 3.2.1 is optimal. For the $\Theta(m)$ part, consider the total number of comparisons in the algorithm. The problem requires at least $\Omega(n \cdot m)$ comparisons, and we have only n processors. Therefore, even if all processors are running all the time, it requires $\Omega(m)$ time to finish all comparisons. Therefore, $\Omega(m)$ gives another lower bound, and Theorem 3.2.1 is optimal.

CHAPTER 4

Sorting and Selecting

In this chapter, we provide algorithms related to sorting and selecting. The problems either require us to sort the data, or to answer questions related to the unsorted data. We give algorithms to sort the data in several cases, find the median in the worst/expected case, and count the number of different labels.

4.1 Sort

Sorting is one of the most fundamental problems that are required by many applications. Researchers have been developing sorting algorithms for many years, and optimal algorithms for the PRAM, taking $\Theta(\log n)$ time [2], and mesh, taking $\Theta(n^{\frac{1}{2}})$ time [20], are well-known. We study this problem on our mesh-connected computer with additional global bandwidth. Throughout, we assume that all values are different. Otherwise, one can associate each value with a processor ID to make it unique.

4.1.1 Sort in the Worst Case

We first give a worst case optimal sorting algorithm on an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$. Note that in the worst case, we have to move everything on the left half of the mesh to the right half. Thus due to the bisection bandwidth, the number of channels must be $\omega(n^{\frac{1}{2}})$ in order to reduce run time. Otherwise the time will be fixed as $\Theta(n^{\frac{1}{2}})$, which is the same as on a standard mesh. Our model MCC$(n, n^{\frac{1}{2}+\epsilon})$ has enough bandwidth to reduce the run time. The hidden multiplicative constant in the $\mathcal{O}$-notation depends upon ϵ, and is small.

The sorting algorithm uses a *redistribution* operation. In the redistribution problem setting, every processor i has a data value v_i and a destination processor ID d_i, where d is a permutation (i.e., all d_i values are different). The redistribution operation moves v_i from processor i to processor d_i, for all i in parallel. Note that processor d_i might not know the value of i.

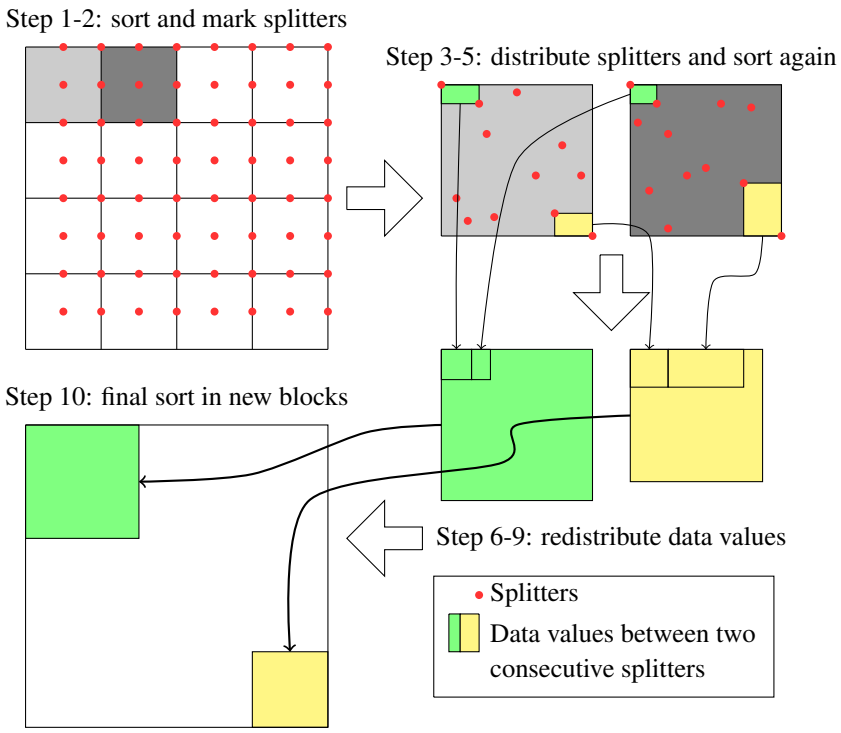


Figure 4.1: Sort using multiple splitters

Lemma 4.1.1. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, redistribution can be done in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time.*

Proof. The algorithm takes $n^{\frac{1}{2}-\epsilon}$ rounds. In the kth round ($0 \leq k < n^{\frac{1}{2}-\epsilon}$), all processors i with d_i in the range $[k \cdot n^{\frac{1}{2}+\epsilon}, (k+1) \cdot n^{\frac{1}{2}+\epsilon})$ write their data values in parallel, with processor i writing v_i to channel $d_i - k \cdot n^{\frac{1}{2}+\epsilon}$. Processors with ID $[k \cdot n^{\frac{1}{2}+\epsilon}, (k+1) \cdot n^{\frac{1}{2}+\epsilon})$ read the channels at the same time to receive their new values. □

With the help of redistribution, we can do sort. Theorem 4.1.2 gives a worst case optimal algorithm for sorting. As mentioned earlier, due to the bisection bandwidth, sort takes $\Omega(n^{\frac{1}{2}-\epsilon})$ time on an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$.

Theorem 4.1.2. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, sort can be done in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time.*

Proof. The proof contains two cases: one with $0 < \epsilon \leq \frac{1}{6}$, and the other with $\frac{1}{6} < \epsilon < \frac{1}{2}$.

We first deal with the first case, when $0 < \epsilon \leq \frac{1}{6}$. The algorithm uses multiple splitters and takes 10 steps. Recall that the processor ordering is a space-filling curve.

1. Divide the mesh into n/B blocks of size $B = n^{\frac{1}{2}-\epsilon} \times n^{\frac{1}{2}-\epsilon}$. Sort the B values within each block in parallel using standard mesh sorting [19], taking $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

2. Let $S = n^{\frac{1}{2}-\epsilon}$ and in each block select every Sth value as a splitter. The total number of splitters is $n^{\frac{1}{2}+\epsilon}$. Note that the block size is greater than the number of splitters, since $B = n^{1-2\epsilon} \geq n^{\frac{1}{2}+\epsilon} = n/S$ when $0 < \epsilon \leq \frac{1}{6}$.

3. Distribute all splitters to all blocks sequentially. The number of splitters equals the number of channels, thus this step takes $\Theta(n/B)$ time.

4. Inside each block, sort the splitters along with all the data values using standard mesh sorting. Let v'_i represent the value in processor i after sorting, not the value initially there. Note that there are $n^{1-2\epsilon} + n^{\frac{1}{2}+\epsilon}$ values inside a block of size $n^{1-2\epsilon}$. For $\epsilon \leq \frac{1}{6}$, this is no bigger than twice the block size. Thus the sort algorithm will have at most a constant factor of overhead.

5. Within each block, divide the data values into $n^{\frac{1}{2}+\epsilon}$ groups, where each group contains the values between consecutive splitters. Count the number of v'_i's in each group, and for each i calculate the rank of v'_i within its group. This step can be done by a segmented scan operation in $\Theta(n^{\frac{1}{3}(\frac{1}{2}-\epsilon)})$ time.

6. We use b to index blocks. Denote the number of data values between the ith and $(i+1)$st splitter inside block b by $C(b, i)$. We can set splitter -1 to be $-\infty$, and splitter n/S to be ∞ to fix the boundary condition. $C(b, i)$ is stored in the ith processor of block b.

7. Processor i, $0 \leq i \leq n^{\frac{1}{2}+\epsilon} - 1$, creates a variable $N(i)$ that is initially 0. The blocks sequentially utilize the channels. Do the following substeps for n/B rounds, one round for each block. In each round, let b denote the current block.

   - Processor i, $0 \leq i \leq n^{\frac{1}{2}+\epsilon} - 1$, writes $N(i)$ to channel i, and this is read by the ith processor in block b;
   - the receiving processor stores this value, call it $N(b, i)$, and sends back $N(b, i) + C(b, i)$ using the same channel;
   - processor i reads the value sent and updates $N(i)$.

   This step takes $\Theta(n/B)$ time.

8. After going through all blocks, do a scan in processors $0 \leq i \leq n^{\frac{1}{2}+\epsilon} - 1$ to compute $M(i) = \sum_{j=0}^{i-1} N(j)$. Thus $M(i)$ is the total number of values less than the ith splitter. Go through all blocks again, sending them the M values. This step takes $\Theta(n/B)$ time.

9. Using the values from step 7 and 8, every processor determines a destination ID for the value it contains. First do a segmented broadcast so that all values between the ith and the $(i+1)$st splitter know $M(i)$ and $N(b,i)$. To determine the destination for a value in block b, suppose its value has rank r in the group of processors with value between the ith and $(i+1)$st splitter. Then its destination is $M(i)+N(b,i)+r$, that is, it will go after all the values less than the ith splitter in all blocks, and after all the values between the ith and $(i+1)$st splitter in preceding blocks, and after the values with smaller rank in its group. The destination ID calculated must be unique for each value. Redistribute all values. This step takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

10. There are at most $n/B \cdot S = n^{\frac{1}{2}+\epsilon}$ values between every two consecutive splitters, and $n^{\frac{1}{2}+\epsilon}$ values in space-filling curve ordering must be covered by a block of size at most $4n^{\frac{1}{2}+\epsilon}$. Sort all of these blocks in parallel, taking $\mathcal{O}(n^{\frac{1}{4}+\frac{1}{2}\epsilon})$ time.

Figure 4.1 gives a brief overview of how this algorithm works. Since $n/B = n^{2\epsilon}$, the algorithm finishes in $\Theta(n^{\frac{1}{2}-\epsilon} + n^{2\epsilon} + n^{\frac{1}{4}+\frac{1}{2}\epsilon})$ time. When $0 < \epsilon \leq \frac{1}{6}$, this is $\Theta(n^{\frac{1}{2}-\epsilon})$.

For the second case, we have $\frac{1}{6} < \epsilon < \frac{1}{2}$. The algorithm uses the same idea as before, but it requires recursive calls.

1. Divide the mesh into n/B blocks of size $B = n^{\frac{1}{2}+\epsilon}$. Sort the data values within each block in parallel using a recursive call. When doing the recursive call, uniformly split the $n^{\frac{1}{2}+\epsilon}$ channels to n/B blocks, and thus each block gets $n^{2\epsilon}$ channels. Each block uses its own channels to help its sort. This step takes time $\Theta(T)$, where T is determined later.

2. Select every $S = n^{2\epsilon}$th value as splitters. In total there are $\Theta(n^{1-2\epsilon})$ of them. Note that $B = n^{\frac{1}{2}+\epsilon} \geq n^{1-2\epsilon}$ when $\frac{1}{6} < \epsilon < \frac{1}{2}$.

3. Do step 3 in the first case. The number of blocks here is $n/B = n^{\frac{1}{2}-\epsilon}$, and thus it takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

4. Inside each block, sort the splitters along with all v_i''s using a recursive call. Note that there are $n^{\frac{1}{2}+\epsilon} + n^{1-2\epsilon}$ values inside a block of size $n^{\frac{1}{2}+\epsilon}$, thus the sort algorithm will have a constant factor of overhead. This step takes $\Theta(T)$ time.

5. Divide all v_i''s into $n^{1-2\epsilon}$ groups, with each group contains only values between consecutive splitters. Count the number of v_i''s in each group, and calculate the rank of values in their group. This step takes $\Theta(n^{\frac{1}{3}(\frac{1}{2}-\epsilon)})$ time.

6. Do step 6 through 8 in the first case. The only modification here is that the number of blocks is $n^{\frac{1}{2}-\epsilon}$, instead of $n^{2\epsilon}$. This step takes $\Theta(n/B) = \Theta(n^{\frac{1}{2}-\epsilon})$ time.

7. Using the values from step 6, all values inside a block can calculate a destination processor ID. This step requires moving $M(i)$ and $N(b, i)$ inside each block, followed by a segmented broadcast. The time it takes is no more than a sort, thus this step finishes in $\mathcal{O}(T)$ time.

8. Redistribute all values. This step takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

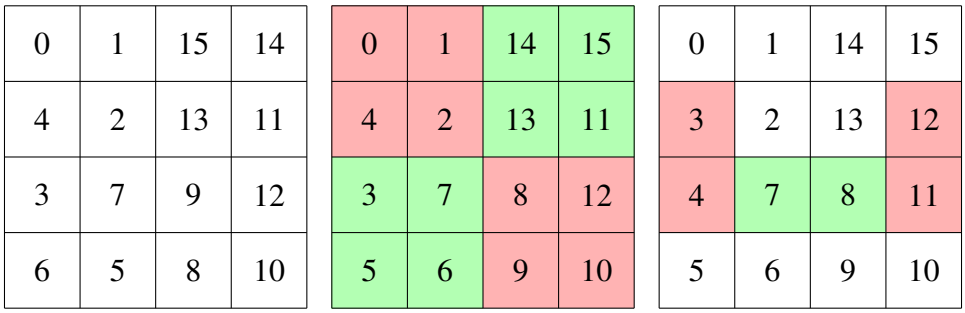
9. We know that there are at most $n/B \cdot S = n^{\frac{1}{2}+\epsilon}$ values between every two consecutive splitters. We also know that $n^{\frac{1}{2}+\epsilon}$ values in a space-filling curve ordering must be covered by a block of size at most $4n^{\frac{1}{2}+\epsilon}$. Therefore, first use a scan to determine the block sizes, and then the number of channels each block is going to use. Then sort all blocks in parallel using a recursive call. This step takes $\Theta(T)$ time.

The algorithm takes $\Theta(n^{\frac{1}{2}-\epsilon} + T)$ time. To determine T, consider step 1. Each block of size $n^{\frac{1}{2}+\epsilon}$ has $n^{2\epsilon}$ channels. When $\frac{1}{6} < \epsilon < \frac{1}{2}$, we have $\frac{2}{3} < \frac{1}{2} + \epsilon < 1$, and $\frac{1}{3} < 2\epsilon < 1$. More precisely, no matter what ϵ is, we can always do a recursive call, since we have $2\epsilon = (\frac{1}{2} + \epsilon)(\frac{1}{2} + \epsilon')$, with $0 < \epsilon' < \epsilon < \frac{1}{2}$. Thus, we know that $T = n^{\frac{1}{2}+\epsilon}/n^{2\epsilon} = n^{\frac{1}{2}-\epsilon}$. This analysis applies to step 1, 4, 7 and 9.

One more concern is that in steps 4 and 9, each recursive call results in a problem size that grows by a constant factor of 2, and thus too many recursive calls will make the problem size too large. Also, since the time does not decrease at each level, the depth of recursion cannot be bigger than a constant; otherwise we may have an extra logarithmic factor. However, when $\frac{1}{6} < \epsilon < \frac{1}{2}$ is a constant, the recursive depth is $\mathcal{O}(\frac{1}{1/2-\epsilon})$ and thus the resulting overhead is at most another constant. Thus the algorithm finishes in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

4.1.2 Sort when Only k Values are Out of Order

The worst case time for sorting cannot be improved. However, in the real world, things may not always be in the worst case. Some input arrays are nearly sorted, and one should expect that there exists a better algorithm to sort in such cases. When the sorting algorithm takes advantage of the existing order of the input, and the time taken by the algorithm is a smoothly growing function of the size of the sequence and the disorder of the sequence, we say that the algorithm is *adaptive*. Many researches have been done on adaptive sorting algorithms [46], and people also try to implement non-adaptive sorting algorithms and



(a) Initial mesh (b) Sort big blocks (c) Sort small blocks

Figure 4.2: Sort when inputs are 1 quasi-sorted

compare their performance on a nearly sorted array [47]. It is especially interesting to design an algorithm to sort a nearly sorted array on our model, since the power of moving small amount of things globally should help significantly in this case.

There are many ways to define a *nearly sorted* array. Here we choose two of them. The first one is *quasi-sorted*. If an array is k quasi-sorted, then each value is at most k away from its correct position. If this is the case, then there is an easy algorithm on a standard mesh to solve it. Let B be the smallest power of 4 that is greater than or equal to $4k$. Sort in blocks of size B. Then we can prove that for the four sub-blocks of size $B/4$ inside each block, the middle two are already correct. This is so because all correct values in the middle two sub-blocks of size $B/4$ must be initially in the big block of size B, and all values smaller than them must be either in the current block, or in the blocks before it. Thus we only need to sort the first sub-block in each block along with the last sub-block in the previous block (see Figure 4.2). Both sorts take $\Theta(k^{\frac{1}{2}})$ time. The mesh time is optimal since in the worst case, everything needs to be moved by a distance of $\Theta(k^{\frac{1}{2}})$.

The second definition is more general: there are k values out of order. That is, for a given array, we are able to take out at most k values to make this array be in an increasing order. Although at most k values are far away from their destinations, it could be that all values are still about k away from their destinations. Thus the problem still has a mesh time lower bound of $\Omega(k^{\frac{1}{2}})$.

Theorem 4.1.3 gives an adaptive algorithm to sort a nearly sorted array, given that at most k values are out of order. We do not need to know k beforehand. Technically *a value that is out of order* is not a well-defined term, because it is possible that there exists two values and one can remove any of them to make the array in an increasing order. In such case, it is hard to tell which value is out of order. However, *a set that is out of order* is a well-defined term, though there could be many such sets. To simplify exposition, we define a set with k out of order values to be a set with size k such that the array is in an increasing

order after the values in the set are completely removed. Throughout, the nearly sorted array has size n.

Theorem 4.1.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given a nearly sorted array with $k \leq n^\epsilon$ values out of order, sort can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time.*

Theorem 4.1.3 requires the following lemmas. The lemmas are step by step, i.e., the later lemmas are based on the previous ones.

Lemma 4.1.4. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given a nearly sorted array with $k \leq n^\epsilon$ values out of order, and the integer k, if each of the k processors which save an out of order value v knows the current position of the value v' that is right before it in the final sorted order after all out of order values are removed, then sort can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time.*

Proof. First note that this lemma has a condition that is not straightforward at first glance. Ideally, the processor that contains an out of order value v should know the final position of v. However, this position is very hard to get, unless all values are already sorted. Thus the current condition is a compromise of it.

The algorithm takes 3 steps.

1. Using the condition, we first move all these k values to a position that is close to their destinations. That is, move the value v to v''s position. This is almost equivalent to a sparse random access write. If more than one values are written to the same position, then in $\mathcal{O}(\log n)$ time, using a PRAM algorithm to reset the destinations of the values so that all values' positions are unique. If this results in an value going to a processor beyond the last one, fix it using another PRAM scan so that all values go into processors in the range. Do a sparse random access write. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

2. Do a scan to determine the number of processors that contain two values and the number of processors that are empty before each processor. Then each value can determine in the current unsorted order which processor it should be in, so that each processor contains only one value. Do a $\Theta(k^{\frac{1}{2}})$ time movement to spread out the values. For example, for a processor i, assume that it contains only one value. Then if the number of processors that contain two values before i is a, and the number of processors that is empty before i is b, then the value in processor i should go to processor $i + a - b$. Note that $|a - b| \leq k$, and thus the movement can be done in $\Theta(k^{\frac{1}{2}})$ time.

3. We know that at the current stage, each value is less than $2k$ away from its real destination. Do a $2k$ quasi-sort. This step takes $\Theta(k^{\frac{1}{2}})$ time.

Therefore, the algorithm finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time. □

Lemma 4.1.4 gives an algorithm in the easiest setting where basically every processor knows everything that it should know. Lemma 4.1.5 requires a weaker condition, where the processors who save out of order values only know that they are out of order, but not where to send them.

Lemma 4.1.5. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given a nearly sorted array with $k \leq n^\epsilon$ values out of order, and the integer k, if all of the k processors which save out of order values know their values are out of order, then sort can be done in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time.*

Proof. We first need to determine which values in the correct order that these k values are after, and then use Lemma 4.1.4 to finish the problem. We can use the same technique as in Lemma 3.3.2. The algorithm takes 3 steps.

1. Take out all these k values, and save them in the channels. Fill in these empty k processors with the values that are closest to them, and mark these values as dummy. This can be done by a segmented broadcast, and takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

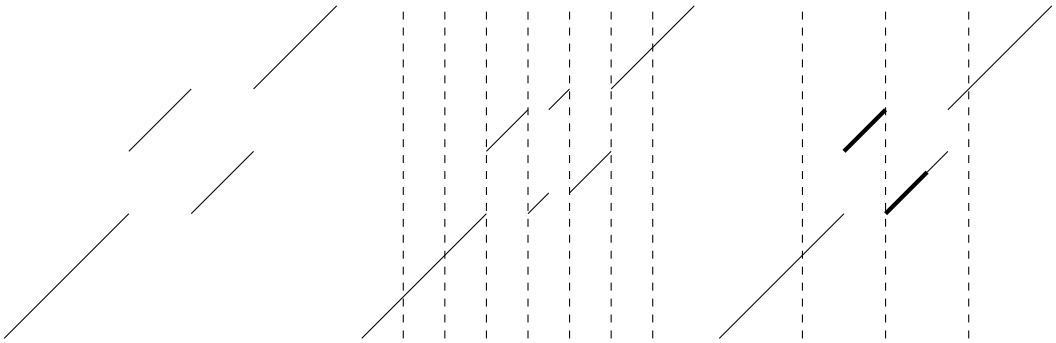
2. Sort the values in the channels. Do a similar process as Lemma 3.3.2. The only difference is that in the binary search, each leader will first get data values from the four leaders in the next level using channels, then do binary search according to these values, and then determine how to cut the channel positions. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

3. Now every value in the channel knows the position that it should go. Use the algorithm in Lemma 4.1.4 to finish the problem.

Therefore, the algorithm finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time. □

We are now able to solve the problem if we know the positions of the k out of order values. Lemma 4.1.6 shows that how we can locate these k values.

Lemma 4.1.6. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given a nearly sorted array with $k \leq n^\epsilon$ values out of order, in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time one can determine a set of at most $2k$ processors, such that the array is in an increasing order after the values saved in these $2k$ processors are taken out.*



(a) Initial values (b) k greater than the block size (c) k smaller than the block size

Figure 4.3: Determine $2k$ values to take out

Proof. First note that k is unknown beforehand. The algorithm uses a similar idea as in the serial algorithm that solves the problem and takes at most $2k$ values out. In the serial algorithm, we make the array a doubly linked list, and do a linear scan on it. Whenever we see the $(s+1)$st value is smaller than the sth value, take both of them out. This greedy approach takes at most $2k$ values out, and finishes in linear time, without knowing k.

Our algorithm works in 4 steps.

1. Divide the mesh into blocks of size $n^{\frac{1}{3}(1-\epsilon)} \times n^{\frac{1}{3}(1-\epsilon)}$. Sort in each block using a standard mesh algorithm. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. Note that the minimum number of values that need to be taken out cannot increase after this sort, because the array after these values are taken out is sorted both before and after the sort.

2. Consider blocks in space-filling curve ordering. For each two consecutive blocks, sort them together using a standard mesh algorithm. Determine if the smallest value in block i is smaller than the biggest value in block $i+1$. If the condition is true, mark block i as active. Note that one can split the checking part into two to avoid overlaps between checking for i and $i+1$, and checking for $i+1$ and $i+2$. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

3. Do a global reduction to determine if there are any blocks that are active. If so, increase the block size by 4, and do step 2 and 3 again for all blocks, not only the active blocks, until all blocks are not active. The ith iteration for this step takes $\Theta(2^i \cdot n^{\frac{1}{3}(1-\epsilon)})$ time, and the time for a later iteration will dominate the total run time. Even if we don't know k, we can still tell that the final iteration will not have block size bigger than $4k$. Otherwise there are at least $4k$ values need to be taken out. Therefore, this step can be done in $\mathcal{O}(k^{\frac{1}{2}})$ time. Note that when $k < n^{\frac{1}{3}(1-\epsilon)} \times n^{\frac{1}{3}(1-\epsilon)}$, the algorithm finishes in the first iteration and takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

4. Determine which $2k$ values need to be taken out. That is, for every two consecutive blocks i and $i+1$, find a length L such that if we remove the last L values in block i, and the first L values in block $i+1$, all remaining values in block i and $i+1$ are in an increasing order. This can be done by a modified sort, i.e., sort them by positions. This step takes $\Theta(k^{\frac{1}{2}})$ time when $k \geq n^{\frac{1}{3}(1-\epsilon)} \times n^{\frac{1}{3}(1-\epsilon)}$, and $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time otherwise.

All steps finish in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time. □

Finally, we are able to prove Theorem 4.1.3. The proof is easy based on the previous lemmas.

Proof of Theorem 4.1.3. First run the algorithm in Lemma 4.1.6 to determine at most $2k$ values to take out. We do not need to know k beforehand, but after this step, we know k, within a constant factor. Then run the algorithm in Lemma 4.1.5 to solve the problem regarding to these $2k$ values. The algorithm finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k^{\frac{1}{2}})$ time. □

One final note is that this algorithm is worst case optimal in most cases, but not all cases. The $\Omega(k^{\frac{1}{2}})$ time lower bound comes from the mesh, thus if we use the channels to move data, we can break this lower bound, and achieve another lower bound $\Omega(n^{1-\epsilon})$. This run time is the same as the worst case sort, and happens only when both ϵ and k are very big.

4.1.3 Splitter Sort

Some problems require us to classify a set T of n items that are not sortable. In such case, sometimes there is another set S of m items, which serve as references and can be sorted. Figure 4.4 gives an example of such problems: given a set of line segments that divide a horizontal slab into non-intersecting regions, and a set of points, sort the given points by regions. In this problem, the order of line segments is well-defined, and thus they are sortable, but the points are not. Points can only be sorted with the help of the given line segments.

Those problems share a same property: items in S cannot evenly divide the space, so that each processor that contains a value in T cannot determine in constant time which two values in S it is between. Note that a similar problem is named "searching and grouping" in [19]. Here we call it *splitter sort*. Definition 4.1.7 provides a formal definition.

Definition 4.1.7. *Given a set S of totally ordered values, the splitters, and a set T of other values, where values in T are comparable to values in S but not those in T, splitter sort*

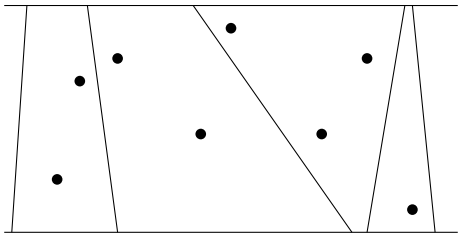


Figure 4.4: Splitter sort: put points into non-overlapping regions

orders the values in T such that for any $s \in S$, given two values $t_1, t_2 \in T$, if $t_1 < s < t_2$ then t_1 precedes t_2 in the ordering, while if $s' \in S$ is the immediate successor of s and $s < t_1, t_2 < s'$ then they can be in either order relative to each other.

Splitter sort is useful in several algorithms when the values in T cannot be compared. To construct a splitter sort algorithm for an ER MCC, first we need to do so for a mesh. An algorithm similar to Proposition 4.1.8 appears in [19], though it is not formulated this way.

Proposition 4.1.8. *On a standard mesh-connected computer of size n, given a set S of $\leq n$ totally ordered splitters, and a set T of $\leq n$ other values, splitter sort can be done in $\Theta(n^{\frac{1}{2}})$ time.*

Proof. Sort all values in S and select $n^{\frac{1}{2}}$ splitters, evenly spaced in S, unless $|S| < n^{\frac{1}{2}}$, in which case select all of them. Call this set S_1. Using a standard mesh rotation algorithm, broadcast S_1 to all processors, so that each processor that has a value in T knows which two splitters in S_1 the value is between. Create an ordered pair (i, j) as a temporary key for each value in T and each remaining splitter $S_2 = S \setminus S_1$, where i means the value is between the ith and the $(i+1)$st splitter, and j is the current processor that it is in. Note that the $(n^{\frac{1}{2}}+1)$st splitter is ∞. Sort all values according to the temporary key.

Now all values in T that are between two consecutive splitters are in a segment of processors, and each segment contains at most $n^{\frac{1}{2}}$ values in S_2. Rotate S_2 values in each segment in parallel, and sort each segment using similar keys as in the previous step to finish the splitter sort. All steps can be done in $\Theta(n^{\frac{1}{2}})$ time. □

Theorem 4.1.9. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a set S of $\leq n$ totally ordered splitters, and a set T of $\leq n$ of other values, splitter sort can be done in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.*

Proof. The proof uses the same approach as for the mesh: broadcast an evenly spaced $S_1 \subset S$ to partition T and to evenly partition $S_2 = S \setminus S_1$, sort them by their temporary keys into segments of processors, and solve the subproblems recursively. Note that the

segment sizes can be seriously unbalanced. When doing recursive calls, for problems on MCC(a, b) with $b = \mathcal{O}(a^{\frac{1}{2}})$, we solve the problem using a standard mesh algorithm, since the channels cannot help in this case. Note that this happens when $a = \mathcal{O}(n^{1-2\epsilon})$, and can serve as a proper base case.

First sort the values in S and select $n^{\frac{1}{2}+\epsilon}$ evenly spaced splitters. If there are not this many, select all of them.

Divide the mesh into n/B blocks of size $B = n^{\frac{1}{2}+\epsilon}$ and copy all splitters to all blocks in n/B time. Solve the sub-problem in each block, taking the values in set T that are originally saved in the block as the new T, and all splitters as the new S.

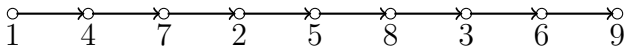
We give each value of S and T an ordered pair (i, j) as a temporary key, which has the same meaning as in the mesh algorithm. Sort all values in T and remaining values in S together. If the size of set S is smaller than or equal to $n^{\frac{1}{2}+\epsilon}$, the problem is already solved after sort. If not, do a scan to determine the number of values between each two consecutive splitters. Some of them has size smaller than or equal to $n^{1-2\epsilon}$, while others have size greater than $n^{1-2\epsilon}$. For both cases, we divide the channels evenly, and finish the problem recursively.

The only problem left is to determine the time of the recursive calls. For recursive calls on an MCC$(n^{\frac{1}{2}+\epsilon}, n^{2\epsilon})$, using the same analysis as worst case sort, the recursive depth is at most $\mathcal{O}(\frac{1}{1/2-\epsilon})$, which is a constant. For the bigger recursive calls in the last step, we know that the current problem has set T of size greater than or equal to $n' = n^{1-2\epsilon}$, and set S of size smaller than or equal to $n^{\frac{1}{2}-\epsilon}$, and channels of size greater than or equal to $n'/n^{\frac{1}{2}-\epsilon} = n^{\frac{1}{2}-\epsilon}$. This means that the number of splitters is always smaller than or equal to the number of channels, and thus the recursive depth is 1. The smaller recursive calls can be solved by a standard mesh algorithm, and thus both cases can be done within one step.

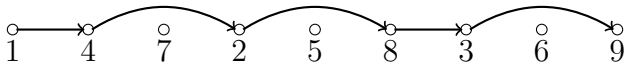
All steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

4.1.4 Chain Rank

Many problems require us to sort a list where the order is defined by the input. The *chain rank* problem is an example. Given an n-edge directed linear chain with vertices $\{s_0, s_1, \ldots, s_n\}$, and every processor contains one arc of the form (s_i, s'_i), where node s'_i is the immediate successor of node s_i, the problem asks to determine the rank r_i for all s_i, where rank is defined as the position in the chain. The problem can be easily generalized into inputs with m non-intersecting chains, and one can expect the algorithm that solves the previous problem can solve this within the same time bound as well. Note that all s'_i's must be distinct, even when the input contains m chains. Otherwise, the chains are not



(a) Input: arcs can be in arbitrary order



(b) After one iteration: for every three consecutive arcs, two of them must be merged

Figure 4.5: Chain rank

well-defined. Also note that the last s'_i in each chain does not have another arc starting with it.

This operation is especially useful when solving problems related to trees. The mesh version is solved in [48], taking $\Theta(n^{\frac{1}{2}})$ time. Theorem 4.1.10 gives a worst case optimal algorithm to solve the problem on a mesh-connected computer with addition global bandwidth. Note that we need the size of the bandwidth to be $\omega(n^{\frac{1}{2}})$, because this problem has a sort lower bound. Also, we do not require s_i's to be integers. They can be any items that can be compared in constant time, and can be sorted.

Theorem 4.1.10. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n arcs of chains in the form of (s_i, s'_i), in $\Theta(n^{\frac{1}{2}-\epsilon})$ time, each arc can determine the rank r_i of itself and the starting point of the chain that contains it.*

Proof. Using channels to move data, a stepwise simulation of the standard mesh algorithm in [48] gives an algorithm with an extra logarithmic factor. Here we provide an algorithm that gets rid of this factor, and is worst case optimal. The algorithm takes 4 steps.

1. Divide the mesh into blocks of size $B = n^{\frac{1}{2}-\epsilon} \times n^{\frac{1}{2}-\epsilon}$. Solve the problem inside each block using a standard mesh algorithm. Shrink each chain into one arc, which only contains its starting and ending positions. Note that it is possible that after this step, no arcs are shrunk, and the problem size is still $\Theta(n)$.

2. In space-filling curve ordering, give each block a block ID, starting from 0. We merge pairs of arcs (s_i, s'_i) and (s_j, s'_j) with $s'_i = s_j$ for the following four cases sequentially:

   - (s_i, s'_i) is in an even block, and (s_j, s'_j) is in an odd block;
   - (s_i, s'_i) is in an odd block, and (s_j, s'_j) is in an even block;

- (s_i, s'_i) is in an even block, and (s_j, s'_j) is in an even block;
- (s_i, s'_i) is in an odd block, and (s_j, s'_j) is in an odd block.

In each case, we call the blocks that contain (s_i, s'_i) to be a starting block, and the blocks that contain (s_j, s'_j) to be an ending block. Each remaining arc in starting blocks creates a record that contains itself and uses s'_i as the key; whereas each remaining arc in ending blocks creates a record that contains itself and uses s_j as the key. Sort all records by their keys, and combine two consecutive arcs (s_i, s'_i) and (s_j, s'_j) if $s'_i = s_j$, (s_i, s'_i) is from a starting block, and (s_j, s'_j) is from an ending block. Update the rank information accordingly.

We need to strictly follow the conditions. This means, e.g., for two consecutive arcs (s_i, s'_i) and (s_j, s'_j), even if $s'_i = s_j$, we do not combine them if their block information does not match the current even/odd condition. These conditions guarantee that each arc can be merged at most once in each four cases. Furthermore, after each case, we temporally remove all arcs that have been updated. This means that each arc can be merged at most once in all four cases, and this also explains what "remaining" means in the previous description. The removed arcs are still active in the following steps, but not in this step.

3. After step 2, the total number of arcs is reduced by $1/3$. Figure 4.5 shows an example. For each three consecutive arcs, at least one pair of them must be merged. Do a global scan to count how many arcs are left, and move them to a block of size $2n/3$. Solve the problem using a recursive call on MCC($\frac{2}{3}n, n^{\frac{1}{2}+\epsilon}$).

4. Propagate the rank information back to all arcs, using the reverse way as how they were gathered.

All steps other than step 3 takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time. For step 3, one key observation is that the number of channels does not decrease with the problem size. Therefore, the run time decreases geometrically in each iteration, and thus the first iteration dominates. Thus the total run time is $\Theta(n^{\frac{1}{2}-\epsilon})$. □

Corollary 4.1.11 follows immediately after Theorem 4.1.10. We only need to keep track of a constant number of additional values when running the algorithm.

Corollary 4.1.11. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given* n *arcs of chains in the form of* (i, s_i)*, where all* i*'s are different, and each arc has a data value* v_i*, broadcast, reduction and scan can be done on each chain in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time.*

4.2 Selection and Finding the Median

Researchers have been developing order selection algorithms for decades, with finding the median the special case of most interest. The selection problem is defined as follow: given n and k, $1 \leq k \leq n$, given an ER MCC of size n, where each processor i has a data value v_i, find the kth smallest value among all v_i's. An optimal serial algorithm is given in [49].

The communication diameter shows that on a standard mesh-connected computer selection can be solved no faster than sorting. Several researchers have given algorithms for this problem on a mesh with row and column buses [25, 24, 50], with the fastest on a square mesh taking $\Theta(n^{\frac{1}{6}} \log^{\frac{2}{3}} n)$ time [25]. An algorithm taking $\Theta(n^{\frac{1}{3}} \log^{\frac{2}{3}} n)$ time on an ER MCC$(n, 1)$ appears in [30]. Here we show that it can be solved in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \log^{\frac{1}{3}} n)$ time on an ER MCC(n, n^{ϵ}). This gives $\Theta(n^{\frac{1}{6}} \log^{\frac{1}{3}} n)$ when $\epsilon = \frac{1}{2}$. In the expected case we eliminate the logarithmic factor, giving $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. This is optimal since it takes that long just to verify that the answer is correct.

4.2.1 Worst Case

We first give an algorithm solving the problem on a smaller set of values. As usual, small changes to the algorithm enable it to find weighted medians.

Lemma 4.2.1. *On an ER MCC(n, n^{ϵ}) with $0 \leq \epsilon < 1$, suppose there are $n^{\frac{1}{3}+\frac{2}{3}\epsilon} / \log^{\frac{2}{3}} n$ disjoint blocks of size $s \times s$, where $s = n^{\frac{1}{3}(1-\epsilon)} / \log^{\frac{2}{3}} n$, and each processor has a value (thus there are $n / \log^2 n$ total values). Given any $1 \leq k \leq n / \log^2 n$, in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \log^{\frac{1}{3}} n)$ time one can find the kth smallest value.*

Proof. Divide the blocks into groups of s and assign a channel to each group. Broadcast and reduction in a group can be done in $\mathcal{O}(s)$ time, as can global broadcast and reduction. Let L denote a lower bound on the answer and U an upper bound. Initially $L = -\infty$ and $U = \infty$. Values between L and U are called *candidates*. Sort the values in each block. Repeat the following steps until $L = U$.

1. Broadcast L and U to all processors.

2. Each block determines the median m of its candidates, and assigns a weight to m which equals to the number of candidates in the block.

3. Using its channel, in each group these weighted m values are collected into a sub-block of size $s^{\frac{1}{2}} \times s^{\frac{1}{2}}$.

4. Using the mesh, find the weighted median of these values, call it x, and give it weight equals to the number of candidates in the group.

5. Use bitonic sort to find the weighted median y of the x values of all the groups.

6. Broadcast y to all processors, use reduction to determine the number of candidates with values $\leq y$, and use this to update L and U.

Step 4 takes $\Theta(s^{\frac{1}{2}})$ time, step 5 takes $\Theta(\log^2 n)$ time, and the others take $\Theta(s)$ time. Standard arguments show that the number of candidates is reduced by a fixed fraction at each iteration, and hence $\mathcal{O}(\log n)$ iterations are needed, proving the lemma. □

Theorem 4.2.2. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, with one value per processor, for any $1 \leq k \leq n$, the kth smallest value can be found in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \log^{\frac{1}{3}} n)$ time.*

Proof. Partition the mesh into $B = n^{\frac{1}{3}+\frac{2}{3}\epsilon} / \log^{\frac{2}{3}} n$ blocks. Each block has size $t \times t$, where $t = n^{\frac{1}{3}(1-\epsilon)} \log^{\frac{1}{3}} n$. We call these "large blocks". Blocks of size $s \times s$, where $s = t/\log n = n^{\frac{1}{3}(1-\epsilon)} / \log^{\frac{2}{3}} n$, are "small blocks". Divide the large blocks into groups of $n^{\frac{1}{3}(1-\epsilon)} / \log^{\frac{2}{3}} n$ and assign a channel to each group.

Let L and U be as in the lemma, and let ℓ be the number of values smaller than L. Sort the values within each large block. Repeat the following steps until $L = U$.

1. In each large block, determine the number of candidates c and let $g = \lceil c/s^2 \rceil$.

2. Select every gth candidate in the sorted order, and move them to a subblock. If the index of the largest item isn't a multiple of g then also select it. The weight of each selected candidate is g, except for the last one, where the weight is its index mod g. Each candidate is the largest in an interval of values, with the number of values equal to its weight.

3. There are at most s^2 candidates selected, so they fit into a small block.

4. Let G be the sum of the g values of the large blocks, then $G - B$ is the maximum number of values between any two selected candidates. To see this, note that $g - 1 \leq c/s^2$ in each large block. Summing over all large blocks gives the result.

5. The number of small blocks, and their size, fits the constraints of Lemma 4.2.1, so use it to find the weighted $(k - \ell)$th candidate in the small blocks. Call this result m. This is a valid upper bound for the kth smallest value. Also find the value m' which is the $(k - \ell - G + B)$th candidate in the small blocks.

6. Let $L = \max\{L, m'\}$ and $U = \min\{U, m\}$.

7. Count the number ℓ of data values smaller than or equal to L. The count is over all values, not just those in the small blocks.

We claim that the number of candidates is reduced by a factor of $\Omega(s^2) = \tilde{\Omega}(n^{\frac{2}{3}(1-\epsilon)})$ at each iteration, and hence at most $\lceil 1/\frac{2}{3}(1-\epsilon) \rceil$ iterations are needed. This is a constant depends upon ϵ.

To see this, note that m is an upper bound on the result. However, there may be more than k items smaller than it. Suppose a block has candidates q_1 and q_2 selected in step 2, where $q_1 \leq L < q_2$ and the block has no selected candidates between q_1 and q_2. The number of values in q_2's interval that are $\leq m$ may be as large as $g-1$, where g is the block's g value from step 1. Summing over all blocks shows that the number of values less than L may be as much as $k+G-B$. The same analysis insures that the number of values $\leq m'$ is $\leq k$, hence it is a lower bound, and is at least $k-G+B$. Thus the number of candidates remaining is at most $2(G-B)$.

Let C be the number of candidates at the start of an iteration. If a large block had c candidates then its g value was $\lceil c/s^2 \rceil$, hence $g-1 \leq c/s^2$. Since this is true for all blocks, $G-B \leq C/s^2$, hence the number of remaining candidates is no more than $2C/s^2$, proving the claim. □

4.2.2 Expected Case

Theorem 4.2.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, with one value per processor, for any $1 \leq k \leq n$, the kth smallest value can be found in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time.*

Proof. Assume that all values are different. The algorithm takes 8 steps.

1. Each processor flips a biased coin, with head probability $1/n^{1-\epsilon}$. If the result is a head, it is selected, otherwise not. The number of selected processors is approximately n^ϵ with high probability. This will be analyzed more carefully at the end. The values in the selected processors are called selected values.

2. Do a scan to count and label all selected values, so that each selected processor knows a channel to write to. If the number of selected values is not in between $[n^\epsilon - 2n^{\frac{\epsilon}{2}}, n^\epsilon + 2n^{\frac{\epsilon}{2}}]$, repeat step 1 and 2 until this number is. These two steps take $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time in total.

3. All selected processors write their values to the channels, taking $\Theta(1)$ time.

4. To simplify exposition, assume that exactly n^ϵ processors are selected. Processors $0 \ldots n^\epsilon - 1$ read the values from the channels and sort them using bitonic sort, using

the channels for communication. Let $K = \lfloor k/n^{1-\epsilon} \rfloor$. Find the left bound L and the right bound R. The left bound is the $(K - n^{\frac{\epsilon}{2}})$th value in the sorted order and the right bound is the $(K + n^{\frac{\epsilon}{2}})$th. This step takes $\Theta(\log^2 n)$ time.

5. Broadcast the bounds. Do a reduction to count the number of processors that contain a value $\leq L$, and count the number of processors with a value $\geq R$. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

6. In step 5, with high probability the kth smallest value is between L and R and the number of values in this range is $\Theta(n^{1-\frac{\epsilon}{2}})$. Otherwise, repeat steps 1 through 5 until the number becomes $\Theta(n^{1-\frac{\epsilon}{2}})$. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time.

7. For the $\Theta(n^{1-\frac{\epsilon}{2}})$ processors left, do a similar process as before using an updated k. Finish the problem using a recursive call. Note that we do not move anything, thus the recursive call is still on the MCC(n, n^ϵ).

Steps 1 and 2 take $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time. To see this, if we do these steps once, they take $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. The number (X) of selected processors follows a binomial distribution, with $p = 1/n^{1-\epsilon}$. Therefore, $\mu = E[X] = np = n^\epsilon$, Var$[X] = np(1 - p) = n^\epsilon - n^{2\epsilon-1}$. Thus, the standard deviation $\sigma \approx n^{\frac{\epsilon}{2}}$. Thus, according to Chebyshev's inequality, $\Pr(|X - \mu| \geq 2\sigma) \approx \Pr(|X - n^\epsilon| \geq 2n^{\frac{\epsilon}{2}}) \leq 1/4$. Therefore, the probability that the number of processors selected is within $2n^{\frac{\epsilon}{2}}$ away from n^ϵ is greater than $3/4$, and thus the expected number of times required for steps 1 and 2 are $1/(3/4) = \Theta(1)$.

Now we calculate how many times step 6 will run. Assume that the n values on mesh are k of -1's and $n-k$ of 1's, and we call this case 1. The kth smallest value will be between L and R if and only if the sum of selected values are between $(n^\epsilon - 2K) \pm 2n^{\frac{\epsilon}{2}}$. This happens with some probability P_1. Now consider a case 2. We randomly generate n^ϵ numbers, and with k/n probability the number is -1, and otherwise 1. Then the probability that the sum of these n^ϵ numbers is between $(n^\epsilon - 2K) \pm 2n^{\frac{\epsilon}{2}}$ is greater than $P_2 = 3/4$ according to Chebyshev's inequality. For case 1, if a 1 is selected, the probability of selecting -1 increases in the following steps; and if a -1 is selected, the probability of selecting 1 increases in the following steps. Therefore, case 2 dominates case 1, and thus $P_1 > P_2$. Thus the kth smallest value will be in the range with probability at least $3/4$.

What's more, we need to make sure that the number we get in step 5 is $\Theta(n^{1-\frac{\epsilon}{2}})$. We do a simplified analysis here. We know that the mesh is divided into n^ϵ parts by n^ϵ selected values, and each part has size approximately $n^{1-\epsilon}$. Assume that each part has size greater than $n^{1-\epsilon}$ with probability $1/2$ independently, then the probability that the number of values in the range between the left bound and the right bound is greater than $n^{1-\frac{\epsilon}{2}}$

is $1/2$. Thus approximately 2 runs are required to make the number we get in step 5 be smaller than $n^{1-\frac{\epsilon}{2}}$. Therefore, step 6 will go through in a constant number of runs in total in expectation.

The time analysis for step 7 is easy: we need to do recursive calls for at most $\lceil 2/\epsilon \rceil$ times. Therefore, the expected number of times that all steps run is bounded by a constant, and thus the total run time is $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ in expectation. □

4.3 Label Counting

In many problems, we need to count how many different data values there are in the input, and relabel them into consecutive positive integers starting from 1. For example, for a graph problem, given edge inputs, we know that there are n edges, but we want to efficiently determine how many different vertices there are. Note that the vertices are not always numbered using integers starting from 1. In reality, a graph is often given with strange vertex names such as city names. Our problem here is to count how many different vertices there are in the graph, and then change the graph to a standard representation so that the computer can deal with it easily later. What's more, we can also count the number of edges from each vertex at the same time.

Theorem 4.3.1 gives an efficient algorithm on an ER MCC to solve the problem. We require the data values to be sortable. The problem cannot be solved with $o(n^2)$ comparisons in the worst case if the inputs are only comparable, but not sortable, and thus $o(n)$ time is impossible with only n processors.

Theorem 4.3.1. *On an ER MCC(n, n^ϵ) with $0 \le \epsilon < 1$, given n sortable data values, one can find the total number of different values S, the number of times that each value appears, and relabel them into positive integers starting from 1 in time*

$$\begin{cases} \Theta(n^{\frac{1}{2}}) & \text{when } \epsilon < \log_n S - \frac{1}{2}, \\ \Theta(n^{\frac{1}{3}(1-\epsilon)} \cdot S^{\frac{1}{3}}) & \text{when } \log_n S - \frac{1}{2} \le \epsilon \le 1 - \frac{1}{2}\log_n S, \\ \Theta(n^{1-\epsilon}) & \text{when } 1 - \frac{1}{2}\log_n S < \epsilon < 1. \end{cases}$$

Proof. The timing for this algorithm shows a perfect transition from a mesh algorithm to an MCC algorithm. When the number of channels is too small to improve the run time, we use a standard mesh algorithm; when the number of channels is very big, we use an MCC sort to finish the problem, regardless of the input. Otherwise, we use a combination of the mesh algorithm and the MCC algorithm.

First assume that we know S. Then the problem becomes just count the number of

times each data value appears. Let $B = n^{\frac{2}{3}(1-\epsilon)} \cdot S^{\frac{2}{3}}$. Divide the mesh into n/B blocks. Sort in each block, count how many times each value appears, and get rid of duplicated values. For each block, at most $\min\{B, S\}$ values are left. Assume that $\min\{B, S\} = S$. There are n/B such blocks, which in total contain $\Theta(n/B \cdot S)$ values. Merge these values in one step using an MCC sort and a segmented reduction. Finally we can relabel each data value according to their final sorted order. The time for solving the base case is $\Theta(B^{\frac{1}{2}})$. The time for moving and merging is $\Theta(n/B \cdot S/n^{\epsilon})$ in total. Thus the total time is $\Theta(B^{\frac{1}{2}} + n/B \cdot S/n^{\epsilon})$. When $B = n^{\frac{2}{3}(1-\epsilon)} \cdot S^{\frac{2}{3}}$, the time is as claimed in the second case.

The algorithm above requires two conditions: $S \leq B$ and $B \leq n$. The first one is required for $\min\{B, S\} = S$, and the second is always required. These two conditions are equivalent to $\epsilon \leq 1 - \frac{1}{2}\log_n S$ and $\epsilon \geq \log_n S - \frac{1}{2}$. These are the two conditions required for the second case. If $\epsilon > 1 - \frac{1}{2}\log_n S$, then B needs to be smaller than S in order to improve run time. In this case, simply choose $B = 1$, and use an MCC sort to finish the problem from the very beginning. If $\epsilon < \log_n S - \frac{1}{2}$, then B needs to be bigger than n in order to improve run time. In this case, simply use a mesh sort to finish the problem. Note that $\log_n S - \frac{1}{2}$ may be smaller than 0 if S is small enough, though it does not matter. In these cases, $\epsilon < \log_n S - \frac{1}{2}$ is automatically false, and the first case will not appear.

In all three cases, the times are as claimed. One final problem is that one might not know S in advance. However, one can determine it by a relatively simple procedure of checking if $S \leq 1$, and if not then checking $S \leq 2$, if not then trying 4, etc. Note that increasing S can only narrow down the range in case 2: if the given ϵ is in case 1 or 3 for S, then it is still in case 1 or 3 for any $S' > S$. What's more, the time for case 2 increases geometrically when S increases geometrically, thus the final run dominates the time.

Therefore, the full algorithm is as follows. Try S, starting from 1, and doubled its value after each try. Whenever the algorithm goes into case 1 or 3, finish the problem within one step because the time does not depend on S any more. The total time is as claimed. □

CHAPTER 5

Fast Fourier Transform

The Fourier transform is a well-known integral transform that was first developed by French mathematician Joseph Fourier [51]. One variance of it, discrete Fourier transform (DFT), is especially interesting to computer scientists. We use DFT to transform between a coefficient representation and a point-value representation of a function. We wish to evaluate a polynomial

$$f(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree n at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$, where $\omega_n = e^{2\pi i/n}$ is the principal nth root of unity (see Figure 5.1). Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, by $y_k = f(\omega_n^k)$. Then the vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the DFT of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$. We also write $y =$DFT$_n(a)$. For more details, the readers are referred to [52].

5.1 Fast Fourier Transform

A well-known method, fast Fourier transform (FFT), uses divide and conquer and computes DFT$_n(a)$ in $\Theta(n \log n)$ serial time [53]. It is also called one of the top 10 algorithms of the 20th century [54]. Let us introduce the serial algorithm first before going into our parallel algorithm.

Given a polynomial

$$f(x) = \sum_{j=0}^{n-1} a_j x^j,$$

a DFT gives the point-value representation of the polynomial at n different points. For FFT, these points are chosen to be $\omega_n^k = e^{2\pi ik/n}$, for $0 \leq k \leq n-1$.

In order to evaluate these points fast enough, we can rewrite the polynomial in the

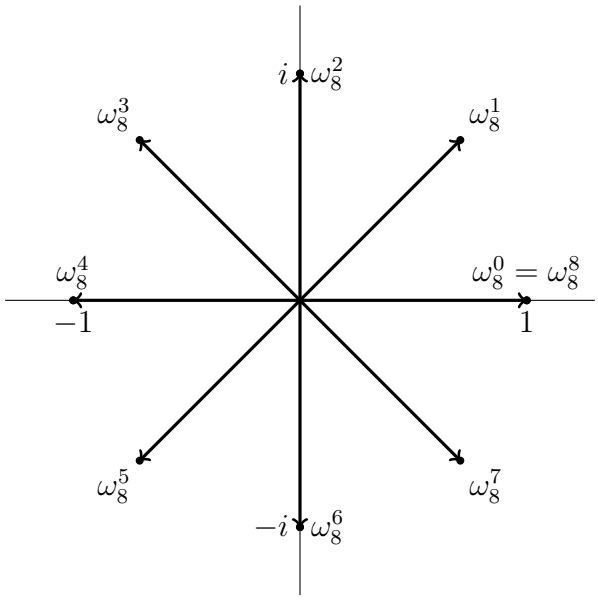


Figure 5.1: The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane

following form:

$$\begin{aligned} f_0(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}, \\ f_1(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}, \\ f(x) &= f_0(x^2) + x f_1(x^2). \end{aligned}$$

Note that $(e^{2\pi ik/n})^2 = (e^{2\pi i(k+n/2)/n})^2$, thus we only need to evaluate f_0 and f_1 at $n/2$ different points. Thus the algorithm finishes in $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ time.

This algorithm can be implemented on a mesh-connected computer. The time for each recursive call decreases geometrically, and thus the final run time is still $\Theta(n^{\frac{1}{2}})$. However, this algorithm requires a lot of data movement, and in each step, there are $\Theta(n)$ values need to be moved a non-trivial distance. In order to make the parallel algorithm on an MCC$(n, n^{\frac{1}{2}+\epsilon})$ more efficient, we need to modify the algorithm a little bit.

We divide the original polynomial into m parts instead of two, where m is a divisor of n. Without loss of generality, assume that n and m are both powers of two. We can rewrite

the polynomial in the following form:

$$\begin{aligned} f_0(x) &= a_0 + a_m x + a_{2m} x^2 + \ldots + a_{n-m} x^{n/2-1}, \\ f_1(x) &= a_1 + a_{m+1} x + a_{2m+1} x^2 + \ldots + a_{n-m+1} x^{n/2-1}, \\ &\vdots \quad\quad \vdots \\ f_{m-1}(x) &= a_{m-1} + a_{2m-1} x + a_{3m-1} x^2 + \ldots + a_{n-1} x^{n/2-1}, \\ f(x) &= f_0(x^m) + x f_1(x^m) + \ldots + x^{m-1} f_{m-1}(x^m). \end{aligned}$$

We know that $(e^{2\pi i(k+jn/m)/n})^m$ are equal for all $0 \leq j \leq m-1$. Thus each recursive call only needs to deal with n/m different points. This gives us an efficient parallel algorithm.

Theorem 5.1.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, the fast Fourier transform can be computed in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.*

Proof. The algorithm takes 5 steps.

1. Divide the given polynomial into $m = n^{\frac{1}{2}-\epsilon}$ polynomials f_i ($0 \leq i \leq m-1$), where each contains $n^{\frac{1}{2}+\epsilon}$ coefficients. Redistribute them into blocks of size $n^{\frac{1}{2}+\epsilon}$ such that all coefficients in the same f_i are saved in the same block.

2. Each block can use $n^{\frac{1}{2}+\epsilon}/m = n^{2\epsilon}$ channels. Use a recursive call to do FFT on an MCC$(n^{\frac{1}{2}+\epsilon}, n^{2\epsilon})$. If $n^{2\epsilon} \leq n^{\frac{1}{2}(\frac{1}{2}+\epsilon)}$, then the recursive call is solved using a mesh algorithm. Otherwise, using the same analysis as in sort, such recursive calls have a constant depth, and thus finishes in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

3. Redistribute the calculated values in a new way such that the $n^{\frac{1}{2}-\epsilon}$ values required by the same group of points are grouped in a block of size $n^{\frac{1}{2}-\epsilon}$. Note that points $x = e^{2\pi i(k+jn/m)/n}$ share the same x^m for $0 \leq j \leq m-1$.

4. Use a mesh rotation algorithm in each block to calculate the final values in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

5. Redistribute the values back to their desired final positions.

All steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

The algorithm on an ER MCC can be easily generated to solve a multi-dimensional FFT problem. Given a d dimensional array x, we need to calculate a d dimensional array

X such that

$$X_{k_1,k_2,\ldots,k_d} = \sum_{n_1=0}^{N_1-1} \left(w_{N_1}^{k_1 n_1} \sum_{n_2=0}^{N_2-1} \left(w_{N_2}^{k_2 n_2} \cdots \sum_{n_d=0}^{N_d-1} w_{N_d}^{k_d n_d} \cdot x_{n_1,n_2,\ldots,n_d} \right) \right),$$

where $w_{N_l} = e^{2\pi i/N_l}$ and $0 \leq k_i < N_i$ for $1 \leq i \leq d$.

Obviously, we can solve each dimension one at a time. Thus, on a regular mesh-connected computer, with $n = N_1 \times N_2 \times \cdots \times N_d$, the problem can be solved in $\Theta(dn^{\frac{1}{2}})$ time.

Theorem 5.1.2. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, the multi-dimensional fast Fourier transform can be computed in $\Theta(dn^{\frac{1}{2}-\epsilon})$ time, where d is the number of dimensions.*

Proof. Most of the time is consumed by date movement. Between each two FFTs for two consecutive dimensions, all the data values need to be moved for a non-trivial distance, which costs $\Theta(n^{\frac{1}{2}-\epsilon})$ time already.

Now we calculate the time required for the ith dimension. There are N_i values need to be grouped together, and there are n/N_i groups. Thus, if $N_i \leq n^{1-2\epsilon}$, solve it using a mesh algorithm. Otherwise, use an MCC$(N_i, n^{\frac{1}{2}+\epsilon}/(n/N_i))$ to solve. Note that each block of size N_i has $N_i/n^{\frac{1}{2}-\epsilon}$ channels to use, thus the run time should be $\Theta(N_i/(N_i/n^{\frac{1}{2}-\epsilon})) = \Theta(n^{\frac{1}{2}-\epsilon})$.

Therefore, the total run time is $\Theta(dn^{\frac{1}{2}-\epsilon})$ as claimed. □

One major concern about using FFT for integer inputs is the loss of precision. This can be fixed by using FFT under modular arithmetic [52]. Assume that all inputs are smaller than n. Let k be the smallest integer such that $p = kn + 1$ is a prime. Let g be a generator of $\mathbb{Z}_p^*$, and let $\omega = g^k \bmod p$. FFT is well-defined under such modular arithmetic, and it will not lose any precision. Also, one can expect that $k = \mathcal{O}(\log n)$ on average.

The most well-known result derived from FFT is to do polynomial multiplication efficiently. In point-value representation, polynomial multiplication can be done in $\Theta(n)$ time, whereas in coefficient representation it takes $\Theta(n^2)$ time. Thus one can first do two FFTs to transform the two polynomials from coefficient representations to point-value representations, then do the multiplication, and finally transform the result back to finish the problem in $\Theta(n \log n)$ time. Theorem 5.1.4 gives the results. The resulting coefficient vector $c = (c_0, c_1, \ldots, c_{2n-2})$ is also called the *convolution* of vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$.

Theorem 5.1.3. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given two polynomials $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$, their product $C(x) = A(x)B(x) = \sum_{j=0}^{2n-2} c_j x^j$ can be determined in $\Theta(n^{\frac{1}{2}-\epsilon})$ time, where $c_j = \sum_{k=0}^{j} a_k b_{j-k}$.* □

With the help of polynomial multiplication, one can solve big integer multiplication efficiently.

Theorem 5.1.4. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given two big integers, each represented by no more than* n *digits, with each digit no more than 9 and the most significant digits greater than 0, their multiplication can be calculated in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time.*

Proof. First use the polynomial multiplication method to find their multiplication, and save these numbers sequentially in space-filling curve ordering. The only problem left is that some numbers now can be as large as $81n$, and we want them to be all less than 10. Do the following until all numbers are no more than 10.

- For each number c_i, let $d_i = \lfloor c_i/10 \rfloor$, and $e_i = c_i - 10d_i$. Send d_i to processor $i+1$ if $i + 1 \leq 2n - 1$.
- Receive the value from processor $i - 1$ if $i > 0$, and use $c'_i = e_i + d_{i-1}$ to be the new value for the next round.
- Use a global reduction to check if all numbers are no more than 10.

Note that the above steps assume that we have $2n - 1$ processors, but it does not matter since we can simulate it using n processors with a constant overhead. We claim that the above process stops in $\Theta(\log n)$ rounds. This is so because after each round, the number in each processor i will be at most $\lfloor a_{i-1}/10 \rfloor + 9$, where a_{i-1} is the number in processor $i - 1$ at the beginning of this round. Assume that at the beginning of the current round, the biggest number in all processors is k. If $k > 30$, then at the beginning of the next round, k is reduced by at least a factor of 2. If $k \leq 30$, one can see that in constant number of rounds, k will become at most 10.

Now the problem is that k could be 10 for at most $\Theta(n)$ rounds, e.g., consider the sequence $(10, 9, 9, \ldots, 9)$. However, since we know that $k = 10$, we can solve this case within one round. Do a scan to count how many consecutive 9's are after each processor. Then for each processor that contains a 10, change itself and all the consecutive 9's after it to be 0, and plus the processor after these 9's by 1. It does not matter even if that processor saves another 10, since in this case, it will be updated to 1. This can be done by a segmented broadcast.

The update part takes $\Theta(n^{\frac{1}{3}(\frac{1}{2}-\epsilon)} \log n)$ time, thus the algorithm finishes in the time as claimed. □

Many other mathematical problems can be solved efficient using FFT. Here we give two examples: recovering a polynomial and chirp z-transform [55].

Given n positions $z_0, z_1, \ldots, z_{n-1}$ that the polynomial $P(x)$ has zeros at, one needs to recover the coefficient representation of $P(x)$. Note that $P(x)$ has degree at most n, and has zeros only at the given n points. Also, there are infinitely many solutions since one can scale $P(x)$, but we can fix the coefficient for the highest order term to be 1 to make $P(x)$ unique. That is, $P(x) = (x - z_0)(x - z_1) \cdots (x - z_{n-1})$. Let $P_j = x - z_j$ be the initial n polynomials. One can see that we can merge them stepwise, two at a time, and thus after $\log n$ iterations, $P(x)$ is found. Multiplying each two polynomials uses one FFT, and thus we need to do FFT $\log n$ rounds to finish the problem. Theorem 5.1.5 gives the result.

Theorem 5.1.5. *On an ER MCC$(n, n^{\frac{1}{2}-\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n positions $z_0, z_1, \ldots, z_{n-1}$, the coefficient representation of a polynomial $P(x)$ that is zero only at these positions can be determined in $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ time.* □

Chirp z-transformation is a generalized version of DFT. Given a vector $a = (a_0, a_1, \ldots, a_{n-1})$ and an arbitrary complex number z, the chirp z-transformation of a is a vector $y = (y_0, y_1, \ldots, y_{n-1})$ such that $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$. When $z = \omega_n$, this is the standard DFT.

The formula can be rewritten as

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right).$$

Therefore, define $P(x) = \sum_{j=0}^{2n-1} a_j z^{j^2/2} x^j$ where $a_j = 0$ when $j \geq n$, and $Q(x) = \sum_{j=0}^{2n-1} z^{-(j-n)^2/2} x^j$. Find $R(x) = P(x)Q(x)$, and we know that the coefficients of $R(x)$ when $n \leq k \leq 2n-1$ are

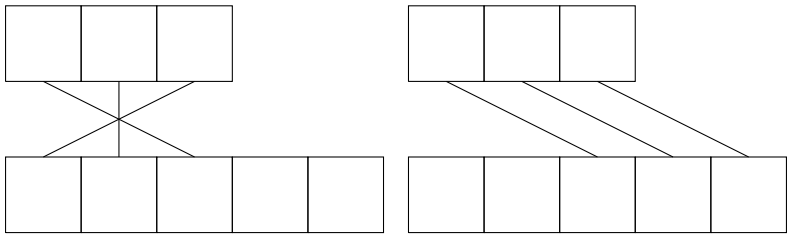
$$b_k = \sum_{j=0}^{k} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j-n)^2/2}\right) = \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-n-j)^2/2}\right).$$

Set $y_k = b_{k+n} \cdot z^{k^2/2}$, the transformation has been calculated.

Theorem 5.1.6. *On an ER MCC$(n, n^{\frac{1}{2}-\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a vector $a = (a_0, a_1, \ldots, a_{n-1})$ and an arbitrary complex number z, chirp z-transformation can be done in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.* □

5.2 Pattern Matching

String pattern matching and image pattern matching are interesting problems that appear for many years [56, 57, 58]. In serial, one can use several different techniques to do exact



(a) Convolution (b) String matching

Figure 5.2: Directions of convolution and string matching

string matching, e.g., using Knuth-Morris-Pratt (KMP) algorithm [59], Aho-Corasick algorithm [60], suffix array [61], or suffix automaton [62]. Much research has been done in parallel string matching as well [63, 64].

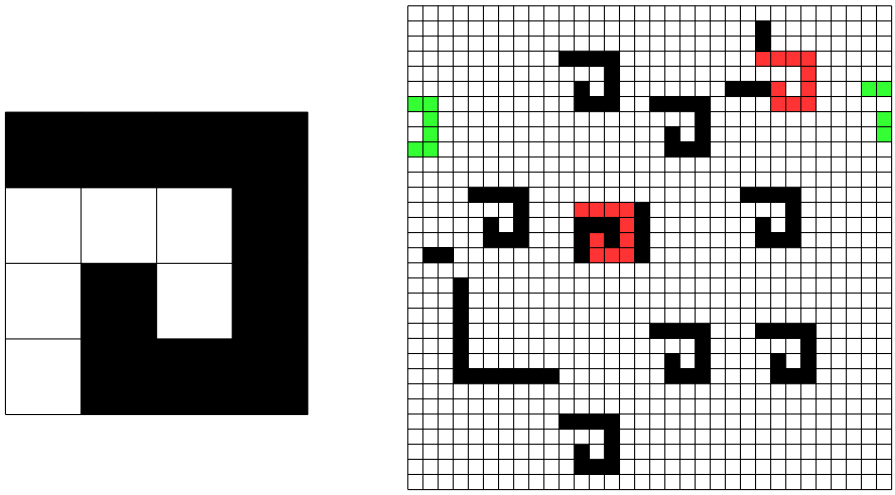
There are several different settings of this problem [56]. These include, but are not limited to, matching with k mismatches, matching with k errors, and matching with wildcards. In the setting with k mismatches, the matching is allowed to have k mismatches; whereas in the setting with k errors, the matching is allowed to have a k edit distance from the pattern string. In serial, one can solve the first case efficiently by using a suffix array, and the second case by using dynamic programming. Here we only focus on string pattern matching with wildcards. In this setting, the pattern string can have wildcards, i.e., don't cares. These positions can match with any characters.

Given a pattern string p of length m containing any number of wildcards and a text string t of length n, the wildcard matching problem is to determine if p occurs in t. Theorem 5.2.1 gives an efficient algorithm to solve the wildcard matching problem with arbitrary number of wildcards.

Theorem 5.2.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given two string p and t of lengths no more than n, where p may contain any number of wildcards, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time one can determine if p appears in t, and how many times it appears.*

Proof. First shrink the pattern string p if it contains wildcards at the beginning or at the end, since they are useless. Reverse the string p, and now the directions of convolution and string matching are the same (see Figure 5.2).

Our goal is to let the number obtained from the convolution to have a specific value if there is a matching from that position. There are many ways to achieve this, and many of them involve using complex numbers to represent each character. Here is one without using complex numbers [65]. Two numbers a and b are the same if and only if $(a - b)^2 = 0$. We use 0 to represent wildcards, and positive integers to represent each character, then



(a) Pattern (b) Image: colors are only for highlighting

Figure 5.3: Image pattern matching: set blank areas in the pattern to be wildcard to count the red regions, and white color otherwise; green region is an example of an invalid matching

for each position in the convolution we calculate $ab(a-b)^2$. This can be represented by $a^3b - 2a^2b^2 + ab^3$, and thus be calculated by three independent FFTs. Since wildcards themselves are 0, p appears at a position in t if and only if the corresponding position in the convolution is 0.

Thus we can use three FFTs to calculate the required values, and use a global reduction to determine how many times p appears in t. All steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

A solution for wildcard string matching can be easily transformed to solve the image pattern matching problem. In an image pattern matching problem, two images I_1 and I_2 are given as input. Assume that I_1 has size $n_1 \times m_1$ and I_2 has size $n_2 \times m_2$, and $n_1 \leq n_2$, $m_1 \leq m_2$. We wish to find how many times I_1 appears in I_2, where I_1 may contain pixels that are wildcards. Brute force can take as much as $\Theta(n_1 m_1 n_2 m_2)$ time in serial, and one can use FFT to do much better.

Using two strings of length $n_1 m_1$ and $n_2 m_2$ to represent two images in row major ordering, the image pattern matching problem is exactly string wildcard matching. To see this, note that in I_2, the pixel p' that is above, below, to the left of and to the right of p is always in the $-m_2, +m_2, -1, +1$ position in the string of I_2 relative to p. Resize I_1 to $n_2 \times m_2$ and fill all outside positions with wildcards, a wildcard string matching gives the result for the image pattern matching problem. Though this matching may involve invalid states, one can simply ignore the boundary conditions when doing the wildcard matching, and remove all impossible positions in the end (see Figure 5.3). In this way, the image

pattern matching problem can be efficiently solved. Theorem 5.2.2 gives the result.

Theorem 5.2.2. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given two rectangular pixel images I_1 and I_2 of size no more than n, where each pixel value is an integer, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time one can determine if I_1 appears in I_2, and how many times it appears.* □

A solution for string matching can also be used to solve many other problems that one might not expect. Here we give one example: determining congruence of polygons. For two polygons, they are *congruent* if they have the same number of sides, and all corresponding sides and interior angles are equal. This means that the two polygons will have the same shape and size, but one may be a rotated, or be the mirror image of the other. Note that in this definition, the polygons are not necessarily simple.

In order to determine whether the given two polygons are congruent, one first needs to check that if they have the same number of vertices. If not, return false. Then we can represent each polygon using a vector of $2n$ numbers, where n of the odd positions are edge lengths, and n of the even positions are the angles. The problem left is to check if these two vectors are equal if they are circular. This is essentially a string matching problem, and can be easily done by using an FFT. One final check is to reflect one polygon, and do the previous checks again. This will finish the problem.

The same method can also be used to check if the two given polygons are *similar*, which means that they might be in different scales. One can first find the length of the longest edges in two polygons, then scale them to the same level, and check if they are congruent.

Theorem 5.2.3. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given two polygons represented by two ordered lists of points, where each list contains no more than n points, congruence of polygons can be determined in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.* □

CHAPTER 6

Graph Algorithms

In this chapter we look at several graph problems. The Minimum Spanning Tree (MST) problem has been well studied for years [66, 67]. Given an undirected graph $G = (V, E)$, where each edge $(u, v) \in E$ has an associated weight $w(u, v)$, we wish to find a spanning tree $T \subset E$ that connects all of the vertices and whose total weight $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized. Technically, for a general undirected graph one needs a Minimum Spanning Forest (MSF) with one tree per connected component. On a mesh, if a graph of $n^{\frac{1}{2}}$ vertices is given as a weighted adjacency matrix, then Atallah and Kosaraju [17] showed that an MST can be found in $\Theta(n^{\frac{1}{2}})$ time. For a collection of n unsorted weighted edges a $\Theta(n^{\frac{1}{2}})$ mesh time algorithm was given by Stout [68]. A time-work optimal randomized algorithm for finding an MSF on an EREW PRAM is given by Pettie and Ramachandran [69].

Once a spanning tree, not necessarily minimum, has been found, many other graph problems can be solved efficiently. Here we consider the following four: determine if G is bipartite, find all the bridge edges of G, find all the articulation points of G, and find all the biconnected components of G. In these problems, without loss of generality, assume G is connected. Otherwise slight modifications are required to make the problem definitions precise, though the algorithms remain the same.

Serial version of these problems are mainly solved by Hopcroft and Tarjan [70, 71], whereas the parallel versions are solved on mesh by Atallah and Kosaraju [17], and independently by Stout [72], and on PRAM by Tarjan and Vishkin [73].

6.1 Graph Algorithms with Matrix Input

In this section the input is an adjacency matrix stored one entry per processor. We first give an algorithm for finding a minimum spanning tree. Our MCC algorithm is based on an approach used in several MST algorithms for distributed memory systems: one finds minimal spanning forests of subsets of edges and then merges the forests to find an MSF

of the full graph. This depends on the following well-known fact:

Fact 6.1.1. *For a graph $G = (V, E)$, let $E' \subseteq E$ and let $G_1 = (V, E')$ and $G_2 = (V, E \backslash E')$. If F_1 and F_2 are arbitrary minimum spanning forests of G_1 and G_2, respectively, then there exists a minimum spanning forest of G that uses only edges in F_1 and F_2.* □

Based on Fact 6.1.1, Theorem 6.1.2 provides a worst case optimal algorithm for finding an MST using divide and conquer.

Theorem 6.1.2. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given the adjacency matrix of an undirected graph of $n^{\frac{1}{2}}$ vertices, a minimum spanning tree (or forest) can be found in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time.*

Proof. The following algorithm returns a minimum spanning forest, or a minimum spanning tree if it exists. We deal with the whole mesh here, but one can easily modify the algorithm to deal with the upper or lower triangle only, which is slightly more energy efficient.

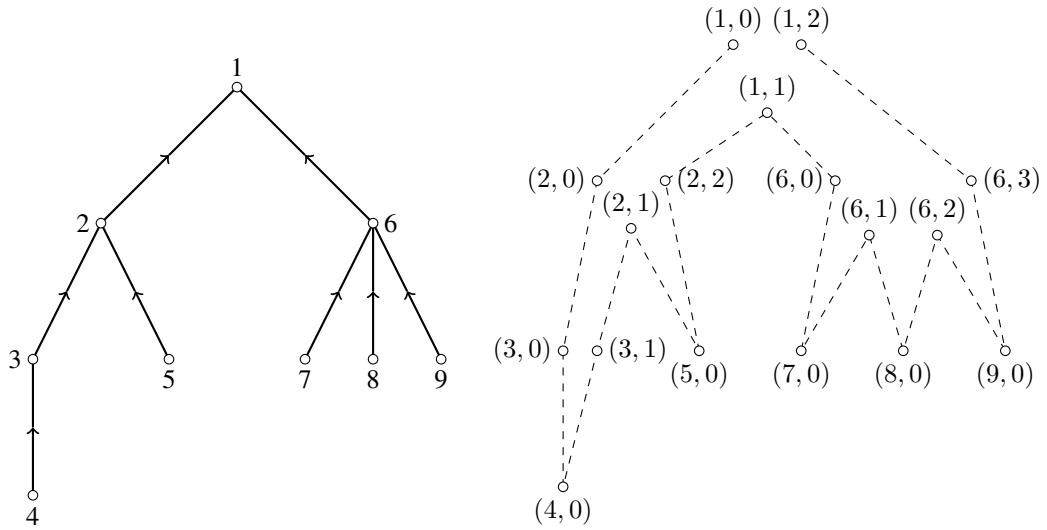
Let $B = n^{1-\epsilon}$. Divide the mesh into n/B blocks. Each block has size $B^{\frac{1}{2}} \times B^{\frac{1}{2}}$, and is a subgraph of the original graph. Find a minimum spanning forest in each block using a standard mesh algorithm [17, 68].

For each block, at most $2B^{\frac{1}{2}} - 1$ edges will be on its minimum spanning forest since there are at most $2B^{\frac{1}{2}}$ different vertices in it. There are n/B blocks, which combined contain $\Theta(n/B^{\frac{1}{2}})$ edge information. These edges will be merged stepwise, four at a time. We always use the channels to move edges, and the moving time is $\Theta(n/B^{\frac{1}{2}}/n^\epsilon)$ in total since the time for the first move dominates.

For solving, in the first few merges, use a standard mesh algorithm to merge the edges from the four consecutive blocks. This will take $\Theta((2^i \cdot B^{\frac{1}{2}})^{\frac{1}{2}})$ for the ith merge. The time increases geometrically, and thus the last merge dominates. Do this until all blocks have been merged, or $2^i = B^{\frac{1}{2}}$. If later, then the current blocks that need to be merged have size B^2, and thus the total number of edges is $n/B^2 \cdot B = n/B$, which equals the number of channels when $B = n^{1-\epsilon}$. Simulate a PRAM algorithm to find a minimum spanning forest [21], and finish the problem in one step. This step takes $\mathcal{O}(B^{\frac{1}{2}} + \log n)$ time. Thus the total time is as claimed by choosing $B = n^{1-\epsilon}$. □

Theorem 6.1.3 shows that many other graph problems can be solved efficiently after a minimum spanning tree is found.

Theorem 6.1.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given the adjacency matrix of a graph G, in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time one can*



(a) Original tree (b) Decomposed chain

Figure 6.1: Creating a linear chain from a tree

1. *determine if G is bipartite,*

2. *find all the bridge edges of G,*

3. *find all the articulation points of G, and*

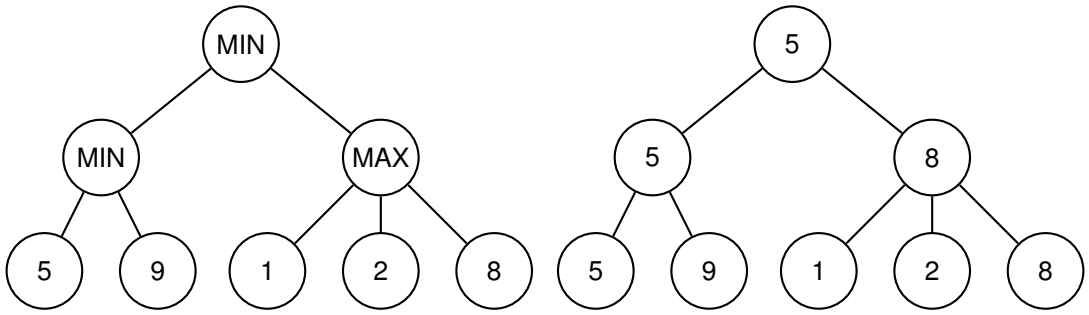
4. *find all biconnected components of G.*

Our proof requires the following lemmas. The algorithms for these lemmas on a standard mesh-connected computer appear in [48], and some of them are solved on a CRCW PRAM in [74]. A stepwise simulation of them on our model would result in an extra logarithmic factor, but our algorithm uses Theorem 4.1.10, and can eliminate the logarithmic factor. These lemmas serve as basic building blocks in our later algorithm.

Lemma 6.1.4. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given an undirected n-node tree, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time the edges can be directed to create a rooted directed tree.* □

Lemma 6.1.5. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given an n-node directed tree, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time the depth, the height, the number of descendants, the pre-order, the post-order, and the in-order numbers of every node can be computed.* □

Lemma 6.1.4 follows from Atallah and Hambrusch [48], and we only need to change the standard mesh version of sort and chain rank to our MCC version. Lemma 6.1.5 can be proved using the well-known technique of creating a linear chain from a tree by Tarjan and Vishkin [74], followed by a scan using Corollary 4.1.11 (see Figure 6.1).



(a) Before evaluation (b) After evaluation

Figure 6.2: Game tree

The following lemma requires a more serious proof. A *game tree*, also known as *min-max tree*, is a rooted tree, such that each leaf node is a data value, and each non-leaf node is either MIN or MAX. Given an arbitrary n-node game tree, the value for each node can be computed efficiently if all internal nodes are of the same type, either MIN or MAX. Figure 6.2 gives an example of a game tree. Note that the tree is not necessary binary.

Atallah and Hambrusch [48] gave an algorithm to evaluate the tree in the general case. However, the algorithm can only find the value of the root, not all nodes in the tree, which is required in our algorithms. Lemma 6.1.6 gives an algorithm to find the value of all nodes in the tree in a special case. This is enough for our later algorithms.

Lemma 6.1.6. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given an* n*-node directed game tree* T *with root* r*, where every leaf has a real number and every internal node is of a same type, either MIN or MAX, in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time the value of the root and all internal nodes can be determined.*

Proof. Here we solve a modified version of the problem. For each non-leaf node, besides a type, it also contains a real number by itself. We still need to evaluate the tree, and find final values for all non-leaf nodes. Note that this version fits our later requirements better, though the version written in the lemma is more common. The new version can be transformed to the original version by adding a dummy leaf node below each non-leaf node, and the original version can be transformed to the new version by adding positive/negative infinities to MIN/MAX nodes. Both transformation will change the problem size by at most a constant factor of 2.

Assume that each node has at least two children. Otherwise, shrink the chains in the tree and update the value in the top node of each chain using only values in the chain. The shrunk chains are saved at the bottom right corner of the mesh, and the final values of nodes in the chain, other than the top one, will be evaluated at the end using a segmented scan

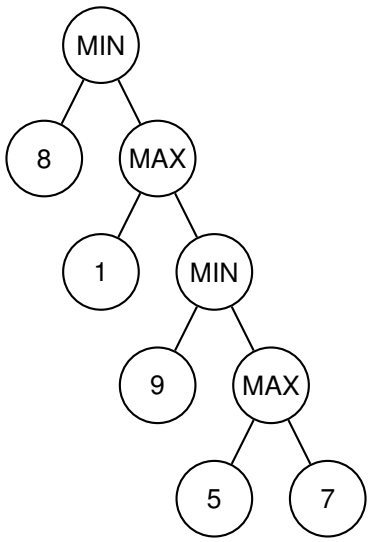


Figure 6.3: A game tree that is hard to evaluate

when all values outside the chains are evaluated.

Now each node has at least two children. This means that at least half of all nodes are leaves. Find the pre-order numbering of all nodes, and make sure that leaves always go before other subtrees in the pre-order numbering. This can be done by first finding the sizes of subtrees of each node, then using it as a key when generating the chain using [74]. Sort all nodes by their pre-order numbering. This step takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

Now we know that all leaf children of a node, if there is any, are immediately after their parents in the sorted order. Do a segmented reduction and update the node's value accordingly. This step takes $\Theta(n^{\frac{1}{2}-\epsilon})$ time. After this step, the problem size is shrunk by $1/2$, because all leaves are gone. Do a recursive call on MCC$(\frac{1}{2}n, n^{\frac{1}{2}+\epsilon})$ to solve the subproblem. Note that the run time decreases geometrically since the number of channels does not decrease with the problem size.

Finally all values outside the chains are calculated. Take out the chains from the bottom right corner of the mesh, and evaluate the chains using a segmented scan. Note that chains will not affect each other, because they are non-intersecting. This step takes $\mathcal{O}(n^{\frac{1}{3}(\frac{1}{2}-\epsilon)})$ time.

All steps finish within $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

One may be wondering that why we cannot run this algorithm for a general tree. The answer is that we cannot efficiently update the value in the top node of each chain. In this special version, it is just a reduction. However, in the general version, this cannot be done easily. This operation is not associative, e.g., (1 MIN 8) MAX 5 $\neq$ 1 MIN (8 MAX 5). Figure 6.3 gives an example of a game tree that is hard to evaluate. It is essentially doing

a scan on a list of values, and the operator is either MIN or MAX. How efficiently we can solve this min-max scan is still open.

There are four different problems in Theorem 6.1.3. Though the solution to these problems are similar, each problem has its own property. We are going to divide the proof into four parts, each for one of the four problems.

Proof of Part 1). We first give our algorithm for determining if G is bipartite, which serves as a good example of how these lemmas work. The following algorithm determines whether the graph is bipartite, broadcasting the decision to all processors, in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time.

1. Find an arbitrary spanning tree. Save it in a block of size $n^{\frac{1}{4}} \times n^{\frac{1}{4}}$. This step takes $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time to find a spanning tree, and $\Theta(\max\{n^{\frac{1}{2}-\epsilon}, 1\})$ time to move the tree. This is so because when $\epsilon \geq \frac{1}{2}$, the moving time is $\Theta(1)$.

2. Arbitrarily determine a root, and make the tree a directed rooted tree. Find the depth of every node. This step takes $\mathcal{O}(n^{\frac{1}{2}(1-\epsilon)})$ time. To see this, consider the following three cases. When $\epsilon \geq \frac{1}{2}$, our MCC algorithm can achieve $\Theta(n^{\alpha})$ run time for any arbitrarily small positive constant α. This is so because there are only $n^{\frac{1}{2}}$ tree nodes, and the number of channels is no less than $n^{\frac{1}{2}}$ when $\epsilon \geq \frac{1}{2}$. When $\frac{1}{4} < \epsilon < \frac{1}{2}$, the MCC algorithm takes $\Theta(n^{\frac{1}{2}-\epsilon}) = \mathcal{O}(n^{\frac{1}{2}(1-\epsilon)})$ time. When $\epsilon \leq \frac{1}{4}$, use a mesh algorithm, and the time is $\Theta(n^{\frac{1}{4}}) = \mathcal{O}(n^{\frac{1}{2}(1-\epsilon)})$.

3. Do row and column broadcasts, and each processor gets the depths of the two endpoints of the edge saved in it. Each processor checks whether the two endpoints' depths have the same parity. This step takes $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time to broadcast, and $\Theta(1)$ time to check.

4. Use a global reduction to collect all results, and the decision of whether G is bipartite is then broadcast to all processors. This step takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time.

The algorithm finishes in $\Theta(n^{\frac{1}{2}(1-\epsilon)} + \max\{n^{\frac{1}{2}-\epsilon}, 1\} + 1) = \Theta(n^{\frac{1}{2}(1-\epsilon)})$ time. □

Proof of Part 2). We now give the algorithm for finding all bridge edges. This requires using a necessary and sufficient condition for an edge to be a bridge edge [19]. Given an undirected graph $G = (V, E)$, let T be a rooted spanning tree and let p_i be the parent of vertex i in T. Let $q_1(i)$ be the index of vertex i in the pre-order numbering of T and let $q_2(i)$ be the largest pre-order index of any vertex in its subtree. Let $l(i)$ be the smallest index of a neighbor of any vertex in i's subtree, and $r(i)$ the largest such index. There is no vertex in i's subtree adjacent to a vertex not in the subtree if and only if $q_1(i) \leq l(i)$ and $r(i) \leq q_2(i)$. If this is true, then edge (i, p_i) a bridge edge (see Figure 6.4).

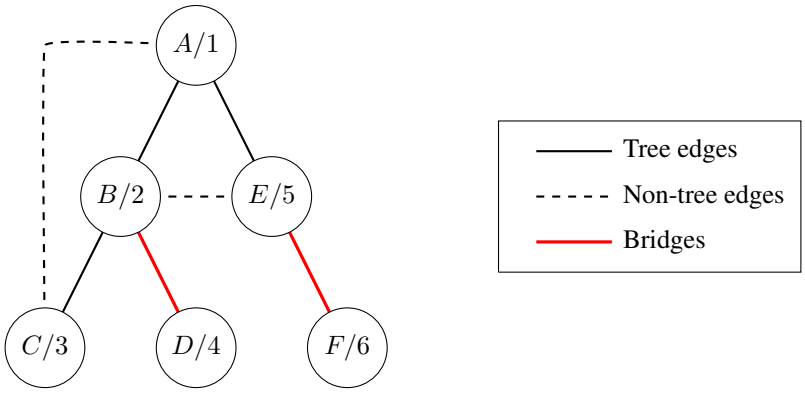


Figure 6.4: Bridges in G, the number on the node represents its pre-order numbering

The following algorithm finds all bridge edges in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time using this necessary and sufficient condition.

1. Find an arbitrary spanning tree. Save it in a block of size $n^{\frac{1}{4}} \times n^{\frac{1}{4}}$.

2. Arbitrarily determine a root, and make the tree a directed rooted tree. Find the pre-order numbering of each node.

3. Solve the min-max tree problem using the directed rooted tree, with all internal nodes marked by MAX, and the pre-order numbering for each node as input. Note that in this case each non-leaf node has a value too. We can transform the current case to the standard one by adding a dummy leaf vertex below each non-leaf node.

   Now the range of the pre-order numbers in the subtree rooted by i is calculated, since the smallest number is the pre-order number of itself, and the largest number is the one just calculated.

4. Do column broadcast, and each processor gets the pre-order number of j if the edge saved in it is (i, j).

5. Do row reduction twice to get the minimum and maximum values in each row.

6. Solve the min-max tree problem using the directed rooted tree, with all internal nodes makred by MIN, and the collected minimum values as input. Do a similar process to find the MAX using the max values. Compare these two numbers with the range obtained in step 3, then whether edge (i, p_i) is a bridge can be determined.

All steps except step 3 and 6 share a same time analysis as in Part 1). For step 3 and 6, the time analysis for solving a min-max tree is the same as the one for rooting a tree, which is described in step 2 in part 1). All steps finish in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time. □

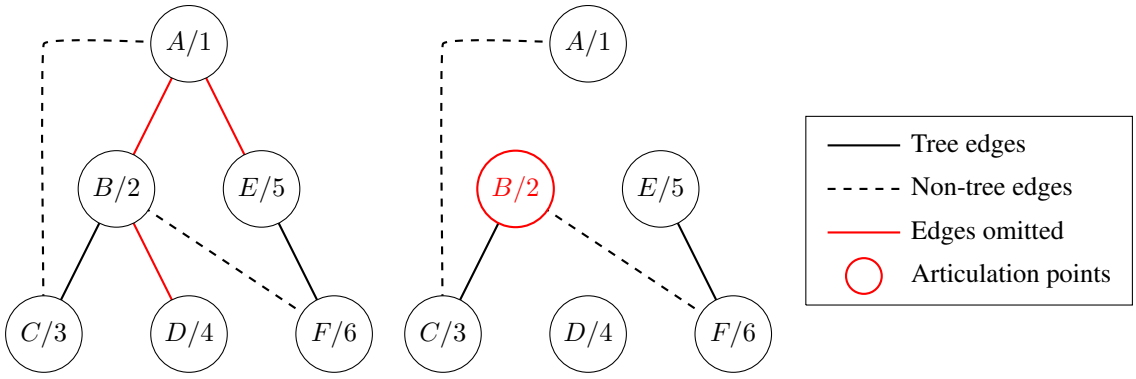


Figure 6.5: Articulation points in G, the number on the node represents its pre-order numbering

Proof of Part 3). The algorithm for finding all articulation points is more complicated than the previous ones and utilizes the MST algorithm for matrix inputs again in Theorem 6.1.2. Using same notation $l(i), r(i), q_1(i), q_2(i)$ as before, let G' be the undirected graph with vertices V and for edges has

- every edge in $E - T$.
- the edge in T between u and a child v if $l(v) < q_1(u)$ or $r(v) > q_2(u)$. That is, a node in v's subtree is adjacent to a vertex not in u's subtree. Omit the edge in T between u and v if such condition does not hold.

Find the connected components in G', then vertex i is an articulation point if and only if i is in a different component than at least one of its children (see Figure 6.5).

The following algorithm can find all articulation points on an ER MCC(n, n^ϵ) with adjacency matrix input in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time.

1. Run the algorithm for finding all bridge edges to gather $l(i), r(i), q_1(i), q_2(i)$ for each i.
2. Broadcast each node's parent to the node's row. For each row i, each processor (i, j) knows the parent of i, and each processor (i, j) can decide if it contains the edge from i to its parent. All processors (i, p_i) are marked as special processors.
3. The edge in T between u and v such that v is a child of u and $q_1(u) \leq l(v)$ and $r(v) \leq q_2(u)$ should be omitted. This can be determined in constant time since we know all required values. There are at most $n^{\frac{1}{2}} - 1$ such edges.

4. All special processors concurrently read the channels in $\Theta(\max\{n^{\frac{1}{2}-\epsilon}, 1\})$ time and determine whether the edge needs to be omitted. When $\epsilon < \frac{1}{2}$, this can be done in $n^{\frac{1}{2}-\epsilon}$ rounds, and in each round n^{ϵ} values are read.

5. Find connected components on the new graph using the MST algorithm with matrix input in Theorem 6.1.2.

6. Save the labels of connected components in the channels. All special processors concurrently read the channels to get the label of i. Similar as in step 4, this part takes $\Theta(\max\{n^{\frac{1}{2}-\epsilon}, 1\})$ time. Do column broadcast, and thus each special processor knows the label of p_i. Determine whether the labels of i and p_i are the same.

7. Do column reduction to collect information for all edges from node i and all its children. Save the collected decisions at the center of mesh.

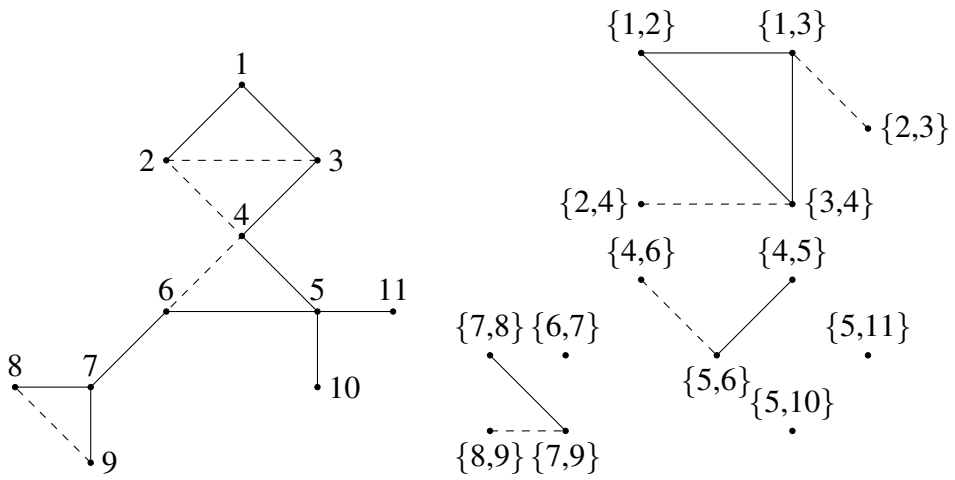
All steps finish in $\Theta(n^{\frac{1}{2}(1-\epsilon)} + \max\{n^{\frac{1}{2}-\epsilon}, 1\} + 1) = \Theta(n^{\frac{1}{2}(1-\epsilon)})$ time. □

Proof of Part 4). A biconnected graph is a connected graph such that if any one vertex were to be removed, the graph will remain connected. If one changes the definition to one edge is removed, the graph is referred as an edge biconnected graph. Such biconnected components can be found by removing all bridge edges. From now on, we only consider the traditional biconnected components. A biconnected component is also referred as a *block*. Note that each articulation points will appear in more than one block, which makes the problem hard.

Our algorithm follows the basic structure of Tarjan and Vishkin's result [73]. Without loss of generality, assume the graph is connected. Otherwise, we can deal with each connected component independently. First we need to find an arbitrary rooted spanning tree T. We denote the edges of T by $v \to w$, where v is the parent of w. Let the vertices of T be numbered from 1 to $n^{\frac{1}{2}}$ in pre-order and identify each vertex by its number in the analysis before the algorithm.

We define an auxiliary graph G' of G whose connected components correspond to the blocks of G. The vertices of G' are the edges of G. We do this because an edge in G can appear in only one block, but a vertex may appear in many blocks. Thus edges are more natural to represent a block. The edges of G' are in the following forms (see Figure 6.6).

1. $\{\{u, w\}, \{v, w\}\}$, where $u \to w$ is an edge of T and $\{v, w\}$ is an edge of $G - T$ such that $v < w$.



(a) Original graph G (b) New graph G': dashed edges are in case 1

Figure 6.6: Biconnected component: graph transformation

2. $\{\{u, v\}, \{x, w\}\}$, where $u \to v$ and $x \to w$ are edges of T and $\{v, w\}$ is an edge of $G - T$ such that v and w are unrelated in T. That is, v is not in w's subtree, and w is not in v's subtree.

3. $\{\{u, v\}, \{v, w\}\}$, where $u \to v$ and $v \to w$ are edges of T and some edge of G joins a descendant of w with a nondescendant of v.

Note that in the analysis, all edges $\{u, v\}$ are undirected since G is an undirected graph. However, given adjacency matrix input, all edges are indeed directed. In the algorithm, we will show that for each pair of edges ((u, v) and (v, u)), only one of them will be used. Even if the algorithm does not take care of that, it is still fine since there will be at most a constant factor of overhead.

Let S be a set of edges in G. S induces a block of G if and only if S induces a connected component of G'. However, G' may have $\Theta(n)$ vertices, which will force us to use algorithms for edge input in Theorem 6.2.1. However, this is not necessary. We can use only edges in condition 2 and 3, which is enough to identify all connected components of vertices that represent edges in T. After that, we can extend the equivalence relation on the edges of T to the edges of $G - T$, which can be done efficiently in parallel. In this way, there are only $\mathcal{O}(n^{\frac{1}{2}})$ vertices and $\mathcal{O}(n)$ edges in G', since now only tree edges can be a vertex in G', and all edges in the original graph can induce at most one edge in G'. Furthermore, we can make use of the original adjacency matrix to make the problem even simpler, though vertices in G' look very different from vertices in G.

The following algorithm finds and marks all edges in a same block by the same label in

$\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time.

1. Run the algorithm for finding bridge edges.

2. Do row broadcast for $p_i, q_1(i), q_2(i), l(i), r(i)$. For each row i, each processor (i, j) knows the parent of i, and processor (i, p_i) can decide if it contains the edge from i to its parent. All processors (i, p_i) are marked as tree edges.

3. Do column broadcast for $p_i, q_1(i), q_2(i), l(i), r(i)$. For each column i, each processor (i, j) knows the parent of j. Processor (p_j, j) knows that its edge should be ignored, since we only need one edge for each tree edge.

4. Now every processor that contains an edge (i, j) knows all five values for both i and j. From now on, let i' represent the edge (i, p_i), which is a directed edge in the spanning tree, and also a vertex in G'. Let's consider edges in G' induced by case 2 and 3.

   - For case 2, consider an edge (a, b) in $G-T$ such that $q_2(a) < q_1(b)$. We need to add edge $((a, p_a), (b, p_b))$ to G'. According to our representation, this is (a', b'). Thus we can use the original edge (a, b) for this case.
   - For case 3, consider an edge (i, p_i) that has been marked as a tree edge. Given $q_1(p_i) \neq 1$, if $l(i) < q_1(p_i)$ or $r(i) > q_2(p_i)$, then we need to add an edge $((p_i, p_{p_i}), (i, p_i))$ to G', which is (i', p_i'). Thus we can use the original edge (i, p_i) for this case.

5. Now we have marked all edges in G'. Find connected components of G' using the MST algorithm with matrix input in Theorem 6.1.2.

6. Collect the labels of each vertex i (same as i'), which is the label for vertex (i, p_i) in G'. Do row and column broadcast so that all processors (i, j) know the labels of both i and j.

7. For each edge (a, b) in $G - T$ in the original graph, the label of it is equivalent to (b, p_b) if $a < b$, and otherwise (a, p_a) obviously. For each edge (p_i, i), the label is equivalent to (i, p_i). After this step, all blocks are labeled.

All steps finish in $\Theta(n^{\frac{1}{2}(1-\epsilon)})$ time. □

6.2 Graph Algorithms with Edge input

In this section we give graph algorithms where the graph $G = (V, E)$ has n edges, stored in an arbitrary order, one per processor, on a mesh of size n. This representation will ignore all vertices that have no edge connected to them. We assume that G does not have such vertices. Otherwise, we will first ignore them, and trivially add them back after the algorithms finish.

If the input is an adjacency matrix, then $|V| = \Theta(n^{\frac{1}{2}})$ and an MST has only $\Theta(n^{\frac{1}{2}})$ edges. However, if the input is unsorted edges, then both $|V|$ and the MST could be as large as $\Theta(n)$. In this case we need the number of channels to be $\omega(n^{\frac{1}{2}})$ in order to improve run time, since all problems have a bisection bandwidth lower bound.

However, moving everything is not always required if $|V|$ is not strictly $\Theta(n)$. An important aspect is the average number of edges per vertex, that is, $|E|/|V|$. This is also known as the *edge density*. If there are n edges then the density ranges from 1 to $n^{\frac{1}{2}}$. For a regular mesh with unsorted edge input there is an optimal algorithm that runs in $\Theta(n^{\frac{1}{2}})$ time [68] no matter what the density. An MCC(n, n^{ϵ}) is sometimes enough to solve this problem more efficiently than the mesh even when $\epsilon \leq \frac{1}{2}$, but the time depends upon the density.

We start by finding an MST in the worst case. Theorem 6.2.1 gives an algorithm for finding an MST given edge input. There are many ways to solve the problem with an extra logarithmic factor. One can either simulate a PRAM algorithm [21], or do a similar process as described in [68]. Note that for the later one, we only need to do coarsening $\lceil \log n \rceil$ times, and reducing the number of edges is not necessary.

Theorem 6.2.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given an undirected graph represented by n unsorted edges, a minimum spanning tree (or forest) can be found in $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ time.*

We do not know if the extra logarithmic factor can be removed or not. If it can be, then that will give a worst case optimal algorithm for finding a minimum spanning tree with edge inputs.

Now we solve the problem again, assuming that the edge density is larger. Theorem 6.2.2 gives an algorithm for finding a minimum spanning tree with density equal to d. Note that d does not have to be larger than 1, though the run time becomes $\Theta(n^{\frac{1}{2}})$ in that case, which equals the solving time on a standard mesh.

Theorem 6.2.2. *On an ER MCC(n, n^{ϵ}) with $0 \leq \epsilon < 1$, for an undirected graph $G = (V, E)$ given as unsorted edge data with n edges, let $d = |E|/|V|$. Then a minimum*

spanning tree (or forest) can be found in time

$$\begin{cases} \Theta(n^{\frac{1}{2}}) & \textit{when } \epsilon < \frac{1}{2} - \log_n d\,, \\ \Theta(n^{\frac{1}{3}(2-\epsilon)}/d^{\frac{1}{3}}) & \textit{when } \frac{1}{2} - \log_n d \leq \epsilon < \frac{1}{2} + \frac{1}{2}\log_n d, \\ \Theta(n^{1-\epsilon}\log n) & \textit{when } \epsilon \geq \frac{1}{2} + \frac{1}{2}\log_n d. \end{cases}$$

Proof. If we do not know d in advance, then first use the algorithm for label counting to count how many vertices there are. The algorithm in Theorem 4.3.1 has the same boundary conditions as this one except the equals sign, and the time for all three cases are within the time limit. After this step, we know d. The algorithm given here is basically the same as the one described in Theorem 6.1.2, except that the base case size and the time analysis are different due to the increasing size of the MST. The algorithm takes 3 steps.

1. Divide the mesh into n/B blocks. Find a minimum spanning tree in each block using a standard mesh algorithm.

2. For each block, at most $\min\{B, n/d\}$ edges will be in its minimum spanning forest because there are only n/d vertices in the graph. Assume that $\min\{B, n/d\} = n/d$, and other cases will be taken care of later. There are n/B such blocks, which in total contain $\Theta(n/B \cdot n/d)$ edge information. Merge these edges stepwise using the same method as before.

3. Finally the minimum spanning forest edges are saved at the center of the mesh, using $\Theta(n/d)$ processors.

Step 1 takes $\Theta(B^{\frac{1}{2}})$ time. For the first iteration of step 2, moving requires $\Theta(n/B \cdot n/d \cdot 1/n^{\epsilon})$ time, and solving requires $\Theta((n/d)^{\frac{1}{2}})$ time in the worst case. As the iterations proceed, the time for moving decreases geometrically as usual, but the time required for solving within blocks stays the same. Using channels for solving would not help much here because the problem size does not change. Either you use it at the beginning, and the time will be $\log n$ times the moving time; or you wait for $\Theta(\log n)$ iterations in order to shadow the logarithmic factor in solving, but in this case the solving time for these iterations is already $\Theta((n/d)^{\frac{1}{2}} \log n)$. We can get rid of a $\log^{\frac{2}{3}} n$ factor when ϵ is very close to $\frac{1}{2} + \frac{1}{2}\log_n d$, but this case is omitted here because it is too specific.

There are $\log(n^{\frac{1}{2}}/B^{\frac{1}{2}})$ iterations, and thus the total time is $\Theta(B^{\frac{1}{2}} + n^{2-\epsilon}/(Bd) + (n/d)^{\frac{1}{2}} \log(n^{\frac{1}{2}}/B^{\frac{1}{2}}))$. Using $B = n^{\frac{2}{3}(2-\epsilon)}/d^{\frac{2}{3}}$, gives $\Theta(n^{\frac{1}{3}(2-\epsilon)}/d^{\frac{1}{3}} + (n/d)^{\frac{1}{2}} \log n)$ time in the second case. Note that $n^{\frac{1}{3}(2-\epsilon)}/d^{\frac{1}{3}} = (n/d)^{\frac{1}{2}}$ when $\epsilon = \frac{1}{2} + \frac{1}{2}\log_n d$, thus when there is a constant gap between ϵ and $\frac{1}{2} + \frac{1}{2}\log_n d$, the $(n/d)^{\frac{1}{2}} \log n$ term is gone, and the time is as claimed in the second case.

The algorithm above requires two conditions: $n/d \leq B$ and $B \leq n$. The first one is required for $\min\{B, n/d\} = n/d$, and the second is always required. These two conditions are equivalent to $\epsilon \leq \frac{1}{2} + \frac{1}{2}\log_n d$ and $\frac{1}{2} - \log_n d \leq \epsilon$. These are the two conditions required for the second case. If $\epsilon > \frac{1}{2} + \frac{1}{2}\log_n d$, then B needs to be smaller than n/d in order to improve run time. In this case, simply choose $B = 1$, and use a worst case MCC algorithm for edge input to finish the problem. If $\frac{1}{2} - \log_n d > \epsilon$, then B needs to be bigger than n in order to improve run time. In this case, simply use a mesh algorithm to finish the problem. In all three cases, the times are as claimed. □

Just as for matrix input, finding a minimum spanning tree given edge input is an important step for many other problems.

Theorem 6.2.3. *On an ER MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, for an undirected graph $G = (V, E)$ given as unsorted edge data with n edges, let $d = |E|/|V|$. Then in*

$$\begin{cases} \Theta(n^{\frac{1}{2}}) & \textit{when } \epsilon < \frac{1}{2} - \log_n d\,, \\ \Theta(n^{\frac{1}{3}(2-\epsilon)}/d^{\frac{1}{3}}) & \textit{when } \frac{1}{2} - \log_n d \leq \epsilon < \frac{1}{2} + \frac{1}{2}\log_n d, \\ \Theta(n^{1-\epsilon}\log n) & \textit{when } \epsilon \geq \frac{1}{2} + \frac{1}{2}\log_n d \end{cases}$$

time one can

1. *determine if G is bipartite,*
2. *find all the bridge edges of G,*
3. *find all the articulation points of G, and*
4. *find all biconnected components of G.*

□

The algorithms used here are almost the same as in Theorem 6.1.3, though they require a different time analysis. One needs to know the following facts.

1. The time for a global broadcast and reduction is always in the time limit.
2. The time for building a rooted tree, finding pre-order numbering, and solving the min-max tree is always $\mathcal{O}(\min\{(n/d)^{\frac{1}{2}}, n^{1-\epsilon}\})$. The first one uses only mesh, and the second one uses MCC(n, n^ϵ). These two terms cannot dominate in their cases.
3. Broadcast the numbering of each vertex to all edges is no more than a reverse step of finding an MST, which is always in the time limit.

The other details are omitted.

6.3 Graph Algorithms on a CR MCC

In this section we look at graph problems where the input is the adjacency matrix of an undirected graph stored one entry per processor in a CR MCC, where a vertex's edges are stored in squares of size $n^{\frac{1}{4}} \times n^{\frac{1}{4}}$, instead of by rows or columns. In this way, we can fully utilize the power of concurrent read. Note that this section and its coarse-grained counterpart are the only places where concurrent read appears. The algorithms in this section are used to illustrate the power of CR in some situations. Note that when the matrix is in a serial computer, in C/C++ the rows are stored in concatenated order. If this ordering is mapped onto the MCC's space-filling ordering the rows are mapped to subsquares. We call this a *block format*.

Given an undirected graph $G = (V, E)$, we wish to find a spanning tree $T \subset E$ that connects all of the vertices. More correctly, we find a spanning tree for each connected component, and hence are finding a forest. Note that this spanning tree is not necessarily minimum. Our CR MCC(n, n^ϵ) algorithm makes full use of the block format layout and finds a spanning tree in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time when ϵ is big enough.

We first provide a data movement operation to solve the following problem. Suppose each processor p_i contains a label l_i, where there may be duplicates and the labels are in random order. Given an integer $k \geq 1$, the *k-label selection problem* is to select k different labels, or all of the labels if there are no more than k different ones. There are no restrictions on which k are selected.

Note that the following lemma is for an MCC$(n, 1)$, not MCC(n, n^ϵ), and that CR plays a critical role.

Lemma 6.3.1. *On a CR MCC$(n, 1)$, given $1 \leq k \leq n$, the k-label selection problem can be solved in $\Theta(n^{\frac{1}{3}} + k)$ time.*

Proof. Divide the mesh into n/B blocks of size $B^{\frac{1}{2}} \times B^{\frac{1}{2}}$, with $B = n^{\frac{2}{3}}$. In each block, sort the labels and eliminate duplicates, then compress the remaining labels, keeping them in sorted order, in space-filling curve ordering. The processors that do not save a label keep null values. This takes $\Theta(n^{\frac{1}{3}})$ time.

Now go through the blocks one by one. The following is repeated until either k values have been broadcast or the last block has broadcast its last value. Each iteration will be called a round.

- The first processor in the current block broadcasts its label if the label is not null, or else broadcasts that the next block should start.

- If a label was broadcast and is the same as the label in a processor then the processor replaces its label with null. This is true even for the current block, i.e., its first processor becomes null.

- If processor p_i has a null value and is not the last processor in its block, then it sends a request to processor p_{i+1} for that processor's label, and replaces null with the label received. Note that the label received could be a null as well. Processor p_{i+1} then sets its label to null, unless it too requested a label, in which case it uses the one received. Note that p_{i+1} is always a neighbor of p_i since a space-filling ordering is being used. We call this step a *move*.

- The previous step is repeated a second time.

Roughly speaking, the null values are being moved towards the end at twice the rate of the broadcasts. The only thing that needs to be proven is that when a block is broadcasting its labels it broadcasts every label in it that has not been broadcast earlier. That is, at the start of each round the first processor never has a null value if there are non-null labels remaining in the block. This can be proven by showing that the following invariant holds at the beginning of each round: for any prefix of the space-filling curve ordering processors in a block before the last processor that contains a non-null value, the number of processors that have a valid label is no less than the number of processors that have a null value.

From now on, a processor is represented by an X if the data value it contains has already been broadcast. Otherwise, it is represented by an O. Given a sequence that contains only O's and X's, we define it to be valid if we can use it to do label selection. That is, we can arbitrarily choose an O and change it to an X, and do two moves after each change, and we can guarantee that before each change, the value at the beginning of the sequence is always an O, unless there is no remaining O's. Though changing the O at the beginning of the sequence to an X seems to be the only case, we want it to work for changing an O at an arbitrary position because we want sequences in other blocks to be valid as well when broadcasting the current block.

We now prove that a sequence is valid if and only if our previous invariant holds. That is, for every position i, the number of O's before i is greater than or equal to the number of X's before i. Same as in the invariant, we ignore all positions after the last O. We only need to prove the "if" part in order to show that the label selection works. The "only if" part is there only to show that this condition is tight.

For the "if" part, we only need to show that given a sequence such that the invariant holds, after changing an O to an X and performing two moves, the invariant still holds. This can guarantee that the sequence is always valid. Consider a prefix $p_0, \ldots, p_i$ of the

sequence. Let O_i be the number of O's in such sequence, and X_i be the number of X's. First note that for any i, each move can only make $X_i - O_i$ smaller, or unchanged. It is impossible that after a move, $X_i - O_i$ becomes bigger for any i.

After one change, we know that $X_i - O_i \leq 2$, since before the change this sequence is valid, which means $X_i - O_i \leq 0$ for all i, and we only change one O to one X. Now we have four cases. In all cases, we assume that there is at least one O after position i. Otherwise we do not care its value.

1. $X_i - O_i \leq 0$, then nothing needs to be done, and two moves would not hurt.

2. $X_i - O_i = 2$, then we know that p_{i+1} is an O. Otherwise $X_{i+1} - O_{i+1} = 3$, which is impossible. Thus, after one move, $X_i - O_i = 0$, and a second move would not hurt.

3. $X_i - O_i = 1$, and p_i is an X. There are two possible cases here. If p_{i+1} is an O, then this can be fixed within one move. Otherwise, p_{i+2} must be an O, and this can be fixed within two moves.

4. $X_i - O_i = 1$, and p_i is an O. We know that p_{i-1} must be an X, otherwise $X_{i-2} - O_{i-2} = 3$. Thus, after one move, p_i becomes an X. Also, we know that there must be at least one O in either p_{i+1} or p_{i+2}, otherwise $X_{i+2} - O_{i+2} = 3$. Thus after the first move, p_{i+1} must be an O, and a second move can fix this case.

Therefore, we can fix all four cases in two moves, and the "if" part is proved. This is enough to show that the label selection algorithm works.

For the "only if" part, we prove the contrapositive statement. We use induction to show that for a sequence that the condition does not hold, it is not valid.

Given a sequence such that the condition does not hold, we know that there exists an i such that $X_i - O_i > 0$. Consider the smallest such i. We define such position to be the smallest invalid point of the sequence, which is represented by S. If $S = 0$, then the proof is done, since the broadcast fails in the first step. The serves as the base case. Note that for all invalid sequences, S must be an even number, regardless of the sequence.

Assume that for any sequence with $S \leq i$, the sequence is invalid. We now prove that for a sequence with $S = i + 2$, it is also invalid. We know that p_{i+2} is an X. Otherwise it is impossible that $S = i + 2$, because at least $i + 1$ is already an invalid position. For a similar reason, p_{i+1} is an X as well.

We only consider the case for changing the O at p_0 to an X, because a valid sequence needs to remain valid for changing an O at any position. One can see that if we always change the last O, it is possible to make an invalid sequence valid after the moves.

Consider the new sequence at positions from 0 to p_i after two moves. We know that $p'_0, \ldots, p'_i$ is a sequence such that the condition does not hold, with $S \leq i$. This is so because $X_i - O_i \geq -1$ before the change, and $X'_i - O'_i \geq 1$ after the change. Since both p_{i+1} and p_{i+2} are X's, two moves are not enough to fix the position i. By induction assumption, we know that it is not valid. Thus the current sequence with $S = i + 2$ is also not valid. This finishes the proof for the "only if" part. □

The data movement in the lemma is quite different from that in the other algorithms in that a leader does not know in advance when it will start broadcasting. Lemma 6.3.2 gives a more general version of k-label selection, using Lemma 6.3.1.

Lemma 6.3.2. *On a CR MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given $1 \leq k \leq n$, the k-label selection problem can be solved in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k)$ time.*

Proof. Divide the mesh into blocks of size $B = n^{1-\epsilon}$. Giving each block a channel, solve the k-label selection problem in each block in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k)$ time, and save the result at the top left corner in each block in consecutive processors. Stepwise merge the results, with each time using one channel to merge $n^{\frac{1}{3}(1-\epsilon)}$ blocks group. In each merge, keep only k different labels, and save them at the top left corner of each group. In $\lceil \epsilon / \frac{1}{3}(1-\epsilon) \rceil$ rounds, the merge is done. Note that this is a constant. Each merge takes $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k)$ time, and thus the total time is as claimed. □

With the help of Lemma 6.3.2, we can significantly reduce the run time for finding a spanning tree.

Theorem 6.3.3. *On a CR MCC(n, n^ϵ), $\frac{1}{4} < \epsilon < 1$, given the adjacency matrix of an undirected graph G of $n^{\frac{1}{2}}$ vertices stored in block format, in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time one can find a spanning tree, or a spanning forest. For $0 < \epsilon \leq \frac{1}{4}$, the time becomes $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$.*

Proof. The algorithm uses the common Boruvka approach. We first show that how to solve for $\epsilon = \frac{1}{2}$. The algorithm takes 4 steps.

Define the label of a tree to be the smallest index of any vertex in the tree. If a processor in the block for vertex u has an edge to vertex v it maintains labels $L(u)$ and $L(v)$, which are the labels of the trees u and v are currently in. Initially each vertex belongs to its own connected component, so initially $L(u) = u$ for all vertices u.

1. In each vertex's block, use a single channel to do $n^{\frac{1}{8}}$-label selection to find $n^{\frac{1}{8}}$ edges that connect to different connected components. There might not be that many such edges.

2. Move the selected edges to the top left block of size $\Theta(n^{\frac{5}{8}})$. Simulate a hypercube or PRAM algorithm stepwise to find the connected components of these edges. Note that the simulation has $\Theta(n^{\frac{1}{8}})$ time delay in each step since there at most $n^{\frac{5}{8}}$ edges but only $n^{\frac{1}{2}}$ channels. After this step, the number of unfinished connected components remaining is at most $\mathcal{O}(n^{\frac{3}{8}})$.

3. Using one channel per vertex, broadcast the new labels for vertices so that all processors can update their vertices' labels. Note that each processor only needs to read from at most two channels.

4. Do step 1 through step 3 four times. At this point there is only one tree per connected component.

Step 1 takes $\Theta(n^{\frac{1}{6}} + n^{\frac{1}{8}})$ time. Step 2 takes $\tilde{\Theta}(n^{\frac{1}{8}})$ time. Step 3 takes $\Theta(1)$ time. Thus the algorithm finishes in $\Theta(n^{\frac{1}{6}})$ time.

For all $\epsilon > \frac{1}{4}$, the only change in the algorithm is that how to do the k-label selection, and what is k. Also, more rounds are necessary. When $\epsilon > \frac{1}{2}$, choose $k = n^{\frac{1}{6}(1-\epsilon)}$ and do a k-label selection in each vertex block using $n^{\epsilon-\frac{1}{2}}$ channels. When $\frac{1}{4} < \epsilon < \frac{1}{2}$, each channel is assigned to more than one vertex. Note that the time for solving base blocks in Lemma 6.3.2 does not require using channels. Set $k = n^{\frac{1}{3}\epsilon-\frac{1}{12}}$, and the base block size to be $n^{\frac{2}{3}(1-\epsilon)}$. Merging k blocks in each round, and thus the total run time is $n^{\frac{1}{2}}/n^{\epsilon} \cdot k = o(n^{\frac{1}{3}(1-\epsilon)})$ for merging in all vertices blocks. In both cases, the k-label selection still takes $\Theta(n^{\frac{1}{3}(1-\epsilon)} + k)$ time, and $k = o(n^{\frac{1}{3}(1-\epsilon)})$ and thus the time for finding a spanning tree is within the time limit.

However, when $\epsilon = \frac{1}{4}$ each channel is assigned to $n^{\frac{1}{4}}$ vertices, and using only the mesh connections can find a single label per vertex in the same time. Collecting $\omega(1)$ per vertex would increase the time needed by the channel to move them, so it is better to just collect one label per vertex and use a logarithmic number of iterations. Another change is that decreasing ϵ does not change the fact that $n^{\frac{1}{2}}$ labels need to be collected in each iteration, taking $\Theta(n^{\frac{1}{2}-\epsilon})$ time. Therefore, the algorithm finishes in $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ time in this case. □

The algorithm is optimal when $\epsilon > \frac{1}{4}$ since it takes that long just to verify that it is a spanning forest even if the graph is known to be two cycles. It is unknown if the logarithmic factor can be eliminated in the worst case when $\epsilon \leq \frac{1}{4}$.

A minimum spanning tree can be found using a similar algorithm as above, though in both cases only one edge will be selected for each vertex in each round. Corollary 6.3.4 gives the result.

Corollary 6.3.4. *On a CR MCC(n, n^ϵ), $\frac{1}{4} < \epsilon < 1$, given the adjacency matrix of an undirected graph G of $n^{\frac{1}{2}}$ vertices stored in block format, in $\Theta(n^{\frac{1}{3}(1-\epsilon)} \log n)$ time one can find a minimum spanning tree, or a minimum spanning forest. For $0 < \epsilon \leq \frac{1}{4}$, the time becomes $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$.*

Note that the time for finding an arbitrary spanning tree and for finding a minimum spanning tree are the same when $\epsilon \leq \frac{1}{4}$. Still, it is unknown if the logarithmic factor can be eliminated in the worst case for both cases. However, in the expected case, when $\epsilon > \frac{1}{4}$, the logarithmic factor can be eliminated.

Lemma 6.3.5 gives a modified version of the label selection algorithm, which is required in our expected case minimum spanning tree algorithm. In this problem, each value is associated with a label, and we need to find k smallest values with different labels. Assume that all values are different. Otherwise we can associate each value with its processor ID to make it unique. The algorithm returns a small amount of smallest labeled values with different labels.

Lemma 6.3.5. *On a CR MCC(n, n^ϵ) with $0 \leq \epsilon < 1$, given $k = \mathcal{O}(n^{\frac{1}{7}(1-\epsilon)})$, one can find k smallest labeled values in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time.*

Proof. The algorithm uses the following fact. Given n distinct numbers, in each round one can randomly select a number and remove all numbers that are smaller than it. Then in $\log n$ rounds, the expected number of numbers left is $\mathcal{O}(1)$. Also, the probability that one can do $2 \log n$ rounds and there are still more than one number left is less than $1/n^3$. This can be proved as follow. Note that in the proof we assume that all numbers are real, and all functions are continuous, instead of discrete. The difference is negligible when n is sufficiently large.

Let $k = 2 \log n$. Suppose $x_1, \ldots, x_k$ are the fractions of numbers we'd like to keep after each random selection. Since there is more than one number left, we know that $x_1 \cdots x_k \geq \frac{2}{n}$. In order to achieve this, in the ith random selection, we need to have at least a fraction of x_i numbers left. This happens with probability $\prod_{i=1}^{k}(1 - x_i)$. Thus we have

$$\prod_{i=1}^{k}(1 - x_i) \leq \frac{1}{k^k}\left(k - \sum_{i=1}^{k} x_i\right)^k \leq \frac{1}{k^k}\left(k - k\sqrt[k]{\frac{1}{n}}\right)^k \leq \left(1 - \frac{1}{\sqrt{2}}\right)^k \leq \frac{0.5^k}{n} = \frac{1}{n^3}.$$

For label L, we define the L value to be the smallest value that has been broadcast so far with label L. The algorithm takes 4 steps.

1. Divide the mesh into blocks of size $B = n^{\frac{1}{3}(1-\epsilon)} \times n^{\frac{1}{3}(1-\epsilon)}$. Sort all values in each block, and find k smallest different labeled values it contains. All other values are

ignored. Note that there could be more than one value for each label that remains, since they could be in different blocks.

2. Each channel is in charge of $n/B/n^\epsilon = n^{\frac{1}{3}(1-\epsilon)}$ blocks. For each channel, go through the blocks in random order. This can be done as follow. Each block generates a random number, and then all numbers are broadcast, so that each block knows how many blocks have numbers greater than or equal to it. If there is a tie, use processor ID as a tie-breaker. After that, all blocks only need to broadcast in the order of the counters.

   Each block broadcasts the smallest labeled value it has, or broadcasts that it has none. When an L value is broadcast, it becomes L's current value and all other L values that are greater than or equal to this are deleted using two moves defined in Lemma 6.3.1. However, smaller L values are kept. If a block broadcasts a value then it remembers what the label and value were. Also, we count how many times each label value has been broadcast. If one of such number is greater than $2\log\frac{n}{B} = (\frac{2}{3}+\frac{4}{3}\epsilon)\log n$, we mark this label as bad.

   After this step, if there is at least one bad label, stop the current algorithm and use an MCC(n, n^ϵ) sort to finish the problem. This happens with probability less than $(1/B)^3 = 1/n^{2(1-\epsilon)}$. Note that this probability is counted as if all blocks are together. Since they are divided by channels, and each group only has $n^{\frac{1}{3}(1-\epsilon)}$ blocks, the real probability will be even smaller.

3. After the previous step, two things may happen for each channel. Either more than k different labels are found, or there are less than or equal to k labels in total. This is so because $n^{\frac{1}{3}(1-\epsilon)}/\left((\frac{2}{3}+\frac{4}{3}\epsilon)\log n\right)$ is far away from k, and thus there must be at least k labels selected if there is this many.

   If the first case is true, then we only consider the smallest k labels. There are at most $(\frac{2}{3}+\frac{4}{3}\epsilon)k\log n < 2k\log n$ blocks that have ever broadcast these labels. If the second case is true, then we consider all blocks that have ever broadcast anything. In both cases, there will be at most $2k\log n$ blocks to consider. Using channels, move all k smallest L values in all these blocks to a sub-mesh. There are in total $\mathcal{O}(k^2\log n)$ of them. Sort them to find the real k smallest labels.

4. Collect k values for each channel to a sub-mesh, and sort them using an MCC(n, n^ϵ) algorithm to find the global k smallest labels.

Step 1 finishes in $\Theta(B^{\frac{1}{2}})$ time. Step 2 finishes in $\Theta(n/B/n^\epsilon + n^{1-\epsilon}\cdot 1/n^{2(1-\epsilon)})$ expected time. Step 3 takes $\Theta(k^2\log n)$ time. Therefore, the algorithm finishes in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$

expected time as claimed when $k = \mathcal{O}(n^{\frac{1}{7}(1-\epsilon)})$. □

With the help of Lemma 6.3.5, Theorem 6.3.6 gives an algorithm to find the minimum spanning tree in expected case.

Theorem 6.3.6. *On a CR MCC(n, n^ϵ) with $\frac{1}{4} < \epsilon < 1$, given the adjacency matrix of an undirected graph G of $n^{\frac{1}{2}}$ vertices stored in block format, in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ expected time one can find a minimum spanning tree, or a minimum spanning forest.*

Proof. We first solve for $\epsilon \geq \frac{1}{2}$. First find the $k = n^{\frac{1}{7}(1-\epsilon)}$ smallest labeled values in each vertex block using the algorithm in Lemma 6.3.5. This step runs on ER MCC$(n^{\frac{1}{2}}, n^{\epsilon-\frac{1}{2}})$ for each vertex, and takes $\Theta((n^{\frac{1}{2}}/n^{\epsilon-\frac{1}{2}})^{\frac{1}{3}} + k) = \Theta(n^{\frac{1}{3}(1-\epsilon)})$ time. Note that for an edge (u, v) in a vertex block, the label is defined as v's label, since all u's are the same in the current block, and their labels are temporarily ignored. Then move all these values to the center of the mesh and solve the minimum spanning tree problem using these edges by simulating any PRAM algorithms [75]. Do it $\lceil 7/(1-\epsilon) \rceil$ times, and the minimum spanning tree is found. All steps finish in $\Theta(n^{\frac{1}{3}(1-\epsilon)} + n^{\frac{1}{7}(1-\epsilon)} \log^{\frac{3}{2}} n)$ expected time.

When $\frac{1}{4} < \epsilon < \frac{1}{2}$, each channel is assigned to more than one vertex. The time for using Lemma 6.3.5 changes slightly since broadcasting now has an $n^{\frac{1}{2}-\epsilon}$ overhead. The time for solving base cases in step 1 does not have such overhead, though we need to make the block size bigger to accommodate the broadcast time. Choose the block size to be $B = n^{\frac{2}{3}(1-\epsilon)}$, then solving in each base block takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time, going through all blocks takes $\Theta(n^{\frac{1}{2}}/B \cdot n^{\frac{1}{2}-\epsilon}) = \Theta(n^{\frac{1}{3}(1-\epsilon)})$ time, and finding global k smallest label for each vertex takes $\Theta(n^{\frac{1}{2}-\epsilon} k^2 \log n)$ time. Choose $k = \sqrt[3]{n^{\frac{1}{2}}/B} = n^{\frac{2}{9}\epsilon - \frac{1}{18}} < B$, the label selection time is $\Theta(n^{\frac{1}{3}(1-\epsilon)})$.

Finding an MST takes $\Theta(n^{\frac{1}{2}} k/n^\epsilon \cdot \log^{\frac{3}{2}} n)$ time. Run the algorithm for $\lceil 1/(\frac{2}{9}\epsilon - \frac{1}{18}) \rceil$ iterations. Thus the total time is as claimed. □

Once a spanning tree has been found, several graph problems can be solved efficiently. Without loss of generality, assume G is connected. Otherwise slight modifications are required to make the problem definitions precise, though the algorithms remain the same. Theorem 6.3.7 gives algorithms to solve several graph problems in the worst case.

Theorem 6.3.7. *On a CR MCC(n, n^ϵ), given the adjacency matrix of an undirected graph $G = (V, E)$ stored in block format, one can*

1. *determine if G is bipartite,*

2. *find all the bridge edges of G,*

3. *find all the articulation points of G, and*

4. *find all biconnected components of G*

in $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time when $\frac{1}{4} < \epsilon < 1$. For $0 < \epsilon \leq \frac{1}{4}$, the time becomes $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$.

Proof. There are three key observations about timing.

1. Broadcasting $n^{\frac{1}{2}}$ values takes $\Theta(1)$ time when $\epsilon \geq \frac{1}{2}$, and $\Theta(n^{\frac{1}{2}-\epsilon})$ time otherwise. When $\epsilon > \frac{1}{4}$, $n^{\frac{1}{2}-\epsilon} = \mathcal{O}(n^{\frac{1}{3}(1-\epsilon)})$; and when $\epsilon \leq \frac{1}{4}$, $n^{\frac{1}{2}-\epsilon} = \mathcal{O}(n^{\frac{1}{2}-\epsilon} \log n)$. Thus such broadcasting is always within the time bound.

2. A global reduction, or a reduction in each vertex block, takes $\Theta(n^{\frac{1}{3}(1-\epsilon)})$ time, which is within the time bound.

3. According to Lemma 6.1.6, solving a min-max tree with $n^{\frac{1}{2}}$ nodes, with all internal nodes of the same type, takes $\mathcal{O}(n^{\frac{1}{2}-\epsilon})$ time with the help of n^{ϵ} channels when $\epsilon < \frac{1}{2}$, and $\mathcal{O}(n^{\alpha})$ for any arbitrarily small constant $\alpha > 0$ otherwise. This is within the time limit. We can do better when $\epsilon < \frac{1}{4}$, but it is not necessary.

Throughout, we use T to represent the time limit. That is, $T = \Theta(n^{\frac{1}{3}(1-\epsilon)})$ when $\frac{1}{4} < \epsilon < 1$, and $T = \Theta(n^{\frac{1}{2}-\epsilon} \log n)$ when $0 < \epsilon \leq \frac{1}{4}$.

We use the following notation: processor i represents the ith processor in space-filling curve ordering, whereas processor (i, j) represents the processor that contains edge (i, j).

First is the algorithm for determining if G is bipartite. The algorithm takes 4 steps.

1. Find a spanning tree.

2. Given the spanning tree, arbitrarily select a node to be the root. Simulate a hypercube or PRAM algorithm to determine the depth of each node.

3. Broadcast the depth of each node. All processors concurrently read the channels, and determine if its edge connects two vertices of the same parity.

4. Do reduction to determine if any processor decides that it is false.

Steps 1, 3 and 4 take $\mathcal{O}(T)$ time. Step 2 takes poly-logarithmic time when $\epsilon \geq \frac{1}{2}$, and $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ time otherwise. Therefore the algorithm finishes in $\Theta(T)$ time.

Second is the algorithm for finding all bridge edges of G. Recall that there is a necessary and sufficient condition for an edge to be a bridge edge, described in Theorem 6.1.3. The algorithm takes 4 steps.

1. Find a spanning tree.

2. Given the spanning tree, arbitrarily select a node to be the root. Simulate a hypercube or PRAM algorithm to determine the pre-order numbering of each node.

3. Broadcast the pre-order numbering of each node. All processors concurrently read the channels, then do reduction in each vertex block to collect minimum and maximum labels each vertex is adjacent to.

4. Using an MCC$(n^{\frac{1}{2}}, n^{\epsilon})$ algorithm to solve the min-max tree problem to find all bridge edges.

All steps finish in $\Theta(T)$ time.

Third is the algorithm for finding all the articulation points of G. We use same notation $p_i, l(i), r(i), q_1(i), q_2(i), G'$ as in Theorem 6.1.3. The algorithm takes 6 steps.

1. Run the algorithm for finding bridge edges.

2. Use channels to broadcast the numbers got from step 1. Now the processor that contains edge (i, j) knows the parent of j, and processor (p_j, j) can decide if it contains the edge from j's parent to j. All processors (p_j, j) are marked as special processors.

3. The edge in T between u and v such that v is a child of u and $q_1(u) \leq l(v)$ and $r(v) \leq q_2(u)$ should be omitted. After the algorithm for finding bridge edges, this can be determined in constant time. There are at most $n^{\frac{1}{2}} - 1$ such edges. Use channels to send the information to all special processors in parallel, so that each special processor can determine whether the edge it contains needs to be omitted.

4. Find a spanning tree on the new graph, after some edges are removed.

5. Use channels to broadcast the labels of connected components to special processors concurrently. Each special processor get the labels of j and p_j. Determine whether the label of j and p_j are the same.

6. Do a reduction in each vertex block to collect information for all edges from a node to all its children, then we can tell whether this vertex is an articulation point or not.

All steps finish in $\Theta(T)$ time.

Fourth is the algorithm for finding all biconnected components of G. The algorithm takes 4 steps.

1. Run the algorithm for finding bridge edges.

2. Using channels to broadcast the numbers got from step 1. Now the processor that contains edge (i, j) knows the parent of i, and processor (i, p_i) can decide if it contains the edge from i to its parent p_i. All processors (i, p_i) are marked as tree edges. They can tell whether their edges should be ignored or not in constant time. The edges in (p_j, j) are automatically ignored.

3. Using the same transformation as in Theorem 6.1.3, find the new graph G', and find all connected components in it.

4. Collect the labels of each vertex, and broadcast them all. For each edge (a, b) in $G - T$, determine its label in constant time.

All steps finish in $\Theta(T)$ time. □

CHAPTER 7

Computational Geometry

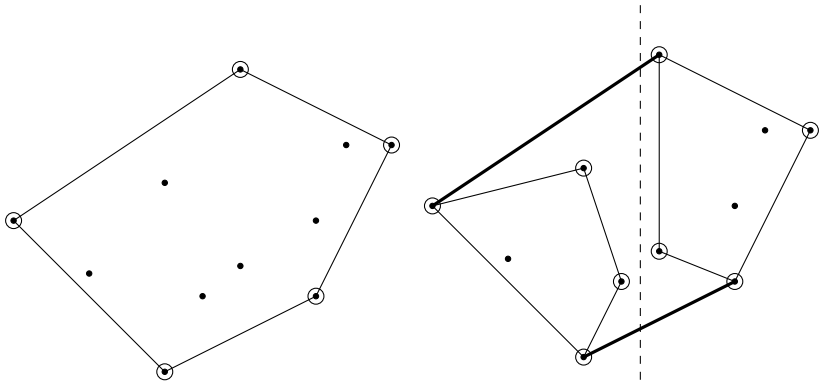
The growing field of computational geometry has provided elegant and efficient serial computer solutions to a variety of problems [76, 77, 78, 79]. Particular attention has been paid to determining geometric properties of planar figures, such as determining the convex hull, and to determining a variety of distance, intersection, and area properties involving multiple figures. For a description of classic problems, efficient serial solutions, and applications of properties in computational geometry, the reader is referred to [80, 81].

Many researchers are also interested in solving geometry problems using parallel computers. Most mesh algorithms can be found in [19]. On an EREW PRAM, one can find a convex hull [22], find all-nearest neighbors [82], and determine visibility from a point [83] in $\Theta(\log n)$ time. For more details, the reader is referred to Goodrich's review of parallel algorithms in geometry [84].

7.1 Convexity

One of the most fundamental problem in computational geometry is to find the convex hull of a point set. Extensive research has been done for serial and parallel algorithms: Googling "convex hull algorithm" returns over a million references. Given a point set S, the convex hull of S, denoted hull(S), is the smallest convex polygon containing S. A point $p \in S$ is an *extreme point* of hull(S) if and only if $p \notin$ hull$(S - \{p\})$. A convex hull can be uniquely represented by the set of its extreme points. Figure 7.1 shows an example of a convex hull of a given point set.

There is a useful divide-and-conquer algorithm paradigm for parallel models where the broadcast/reduction/scan time is significantly faster than the sorting time. The MCC model has this property, as do the pyramid, reconfigurable mesh, mesh with row and column buses, and others. There is an initial sort, and then a poly-logarithmic sequence of broadcast, reduction, and scan operations.



(a) A convex hull with 5 extreme points

(b) The support lines

Figure 7.1: A convex hull example

A well-known application of this is to find the convex hull of set S. First sort the points by x-coordinate. Find the leftmost point, where if there are ties for leftmost pick the one with largest y-coordinate. Do the same for the rightmost point, and then broadcast these points. Let $T \subset S$ be the points above the line connecting these edge points. The upper portion of hull(S) consists of points in T. To find the upper hull, start with consecutive pairs of points in T, forming initial segments. Throughout, each point remaining in its segment records the neighboring points remaining, and its index in an ordering of them.

Repeatedly join consecutive pairs of segments, finding the upper support line joining them (see Figure 7.1). This can be accomplished via binary search, where each step involves broadcasting a line segment of the partial hull on the left, and then a reduction is used on the remaining points on the right segment to determine if any of them lie above the extension of this segment. Once the support lines have been determined, some formerly active points from each side may now become inactive. Broadcasting the index and coordinates of the rightmost point remaining on the left side, and leftmost on the right side, allows every remaining point to adjust its index and neighbor information. This process continues until only 1 segment is left, containing the upper hull.

There are $\lceil \log n \rceil$ rounds of combining neighbors, each of which involves $\mathcal{O}(\log n)$ iterations to find the support line, so the total time is $\mathcal{O}(\mathrm{Time}(\mathrm{Broadcast}/\mathrm{Reduce}/\mathrm{Scan}) \cdot \log^2 n)$. This can be reduced in some parallel models [22], but in the MCC and others the time is dominated by the initial sort. For the MCC this gives:

Theorem 7.1.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n or fewer planar points, the extreme points of their convex hull can be found in $\Theta(n^{(\frac{1}{2}-\epsilon)/3} \log^2 n)$ time if the points are pre-sorted, and $\Theta(n^{\frac{1}{2}-\epsilon})$ time otherwise.* □

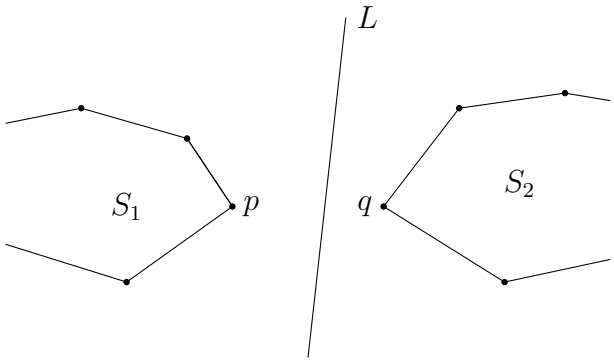


Figure 7.2: Line L separates S_1 and S_2

The following two corollaries are straightforward consequences of Theorem 7.1.1. Corollary 7.1.2 merely requires some additional broadcast and reduction operations, and hence its time can be significantly reduced if the data is presorted. Corollary 7.1.3 requires an additional sort-based operation after the convex hull has been determined since for each edge information about points on the opposite side of the convex hull is needed. See [19] for details.

Corollary 7.1.2. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a set S of n or fewer planar points, the area, perimeter and centroid of the convex hull of S can be determined in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.* □

Corollary 7.1.3. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a set S of n or fewer planar points, a smallest enclosing box of S can be identified in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.* □

There are many other well-known problems related to convexity. Theorem 7.1.4 and Theorem 7.1.5 serve as examples of how these problems could be solved efficiently.

Sets of planar points S_1 and S_2 are *linearly separable* if there exists a line in the plane such that all points in S_1 lie on one side of the line, and all points in S_2 lie on the other side. S_1 and S_2 are linearly separable if and only if their convex hulls do not intersect.

Given the convex hulls of S_1 and S_2, or determining them via Theorem 7.1.1, a binary search as above can determine a separating line, if it exists, or their intersection otherwise. The details are omitted. Note that the intersection of convex sets is a convex set.

Theorem 7.1.4. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n or fewer labeled planar points, representing sets S_1 and S_2, determining whether or not the convex hulls of S_1 and S_2 intersect can be done in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. Further, if they do not intersect then a separating line can be found, while if they do intersect the extreme points of their intersection can be determined in the same time bound.* □

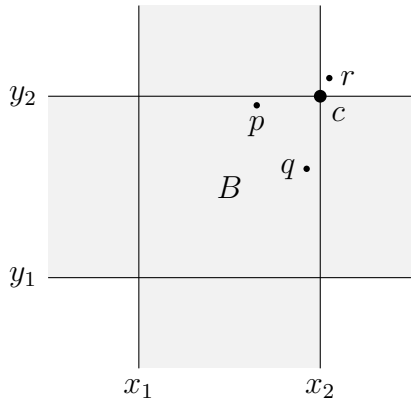


Figure 7.3: Nearest neighbor in a corner, p and q may have closer points outside the current vertical and horizontal slabs because currently $d(p) > \overline{pc}$ and $d(q) > \overline{qc}$; this condition will be true for at most two points for each corner in each block

The following problem can also be solved using a straightforward bottom-up merging algorithm [81].

Theorem 7.1.5. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given the description of n or fewer half-planes, their intersection can be determined in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.* □

For both of these theorems the time can be reduced to $\tilde{\Theta}(n^{(\frac{1}{2}-\epsilon)/3})$ if the points are presorted by x-coordinate, or the half-planes are sorted by slope, where $\tilde{\Theta}$ means that there may be additional logarithmic factors.

7.2 Nearest Neighbor Problems

In this section, problems are considered that involve nearest neighbors of planar points. A variety of nearest neighbor problems have been explored in serial [85, 86, 87]. For planar points x and y, $d(x, y)$ denotes the Euclidean distance between them.

The first problem is the all-nearest neighbor problem for planar points. Given a set S of points, a *nearest neighbor* of point $p \in S$ is a point $q \in S$, $q \neq p$, such that $d(p, q) = \min\{d(p, r) : r \in S, r \neq p\}$. The solution of the all-nearest neighbor problem for points consists of finding, for every point $p \in S$, a nearest neighbor of p. Note that a nearest neighbor is not necessarily unique, but the distance to it is. For a point p, we use $d(p)$ to represent such distance. Theorem 7.2.1 gives an optimal algorithm for solving all-nearest neighbor problem.

Theorem 7.2.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a set S of n or fewer planar points, the all-nearest neighbor problem can be solved in $\Theta(n^{\frac{1}{2}-\epsilon})$ time.*

Proof. The proof uses the same idea as the standard mesh algorithm [19]. We divide the plane into several vertical and horizontal slabs, solve the problem in each slab, and finally solve the remaining problem with at most 8 points in each block (see Figure 7.3), which is the intersecting region of a vertical and a horizontal slab.

First sort all points by their x-coordinates. Divide the plane into $n^{\frac{1}{4}-\frac{1}{2}\epsilon}$ vertical slabs, where each contains $n^{\frac{3}{4}+\frac{1}{2}\epsilon}$ points. Solve the all-nearest neighbor problem in each slab using a recursive call. Then sort all points by their y-coordinates, and do the same.

The $n^{\frac{1}{4}-\frac{1}{2}\epsilon}$ vertical slabs and the $n^{\frac{1}{4}-\frac{1}{2}\epsilon}$ horizontal slabs divide the plane into $n^{\frac{1}{2}-\epsilon}$ blocks, and within each block, there are at most 8 points for which the optimal answers are still unknown. After the previous step, each point knows which vertical and horizontal slab it is in, and what the boundaries are. In constant time, each point p determines if its distance to the closest block boundary c is smaller than its current answer. If not, then it is impossible for another point r outside the current vertical and horizontal slab to form a better answer $d(p, r)$ than p's current answer. Otherwise the point is not done yet (see Figure 7.3). Use a global reduction to count the number of points that are not done, collect all these points, and use the rotation operation to finish the problem. Note that the total number of points that are not done is no more than $8n^{\frac{1}{2}-\epsilon}$.

It is clear that all steps other than the recursive calls finish in $\mathcal{O}(n^{\frac{1}{2}-\epsilon})$ time. For the recursive calls, assume that we are currently solving a problem on MCC(a, b). Using our algorithm, we divide the plane into $\sqrt{a/b}$ slabs, and thus the subproblem size in the next step will be $a/\sqrt{a/b} = \sqrt{ab}$. Also, we know that the number of channels we have is $b = a/n^{\frac{1}{2}-\epsilon}$, because we will give each $n^{\frac{1}{2}-\epsilon}$ values a channel, regardless of a. Therefore, in the next iteration, the problem size becomes $\sqrt{ab} = a/n^{\frac{1}{4}-\frac{1}{2}\epsilon}$. Therefore, in $K = \lceil 2\epsilon/(\frac{1}{4}-\frac{1}{2}\epsilon)\rceil$ iterations, the problem size will become less than $n^{1-2\epsilon}$, and we can use a standard mesh algorithm to finish the problem at this stage. Note that K is a constant depends upon ϵ. Therefore, all recursive calls finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

Next we consider problems where the points have labels. A pair of labeled points with different coordinates are a *dissimilar pair* if and only if their labels differ. Given a set S of labeled planar points, the *closest dissimilar pair* problem is to determine a dissimilar pair of points in S that have the minimum distance among all dissimilar pairs. We first solve a special case, which is then used for solving the general case.

For sets of points A and B, the distance between A and B is $\min\{d(a, b) : a \in A, b \in B\}$. Recall that A and B are linearly separable if there is a line L such that all points of A are on one side of L and all points of B are on the other. Points in A are labeled A, and similarly for B.

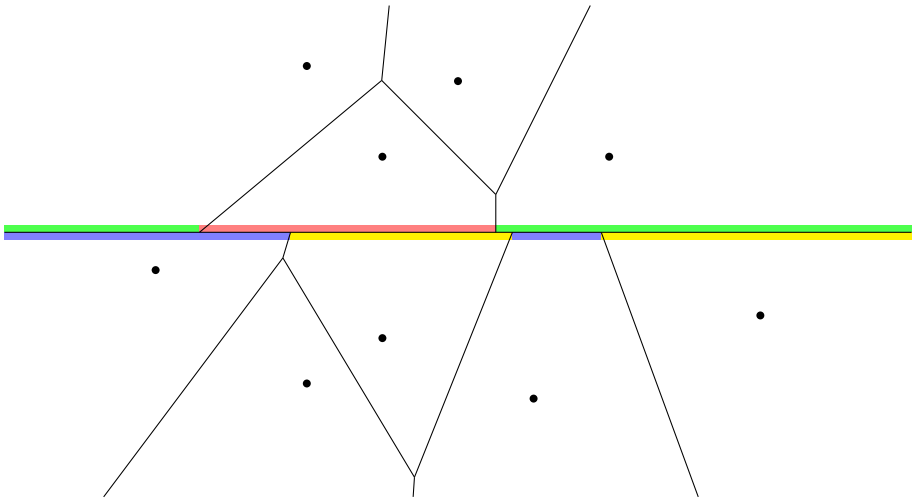


Figure 7.4: Voronoi partition

Proposition 7.2.2. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given linearly separable sets of planar points A and B with a total of at most n points, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time the distance between A and B can be determined.*

The proof is based on the following fact: suppose $a \in A$ and $b \in B$ are such that $d(a, b)$ equals the distance from A to B. Since A and B are separated by a line L, L must intersect the line segment $\overline{ab}$ at some point p. Since $d(a, b)$ is the minimum distance between points in A and points in B, it must be that a is a closest point in A to p, and b is a closest point in B to p. To locate a (and similarly b) we utilize a *Voronoi partitioning* (see Figure 7.4), which is the intersection of L and the Voronoi diagram of A.

It can be described as follows. First, assume that L is the x-axis. Otherwise one can rotate the line and all points so that L becomes the x-axis. Given a set of planar points S, the Voronoi partition of the x-axis induced by S is a set of intervals $(-\infty, x_1], [x_1, x_2], \ldots, [x_{k-1}, x_k], [x_k, \infty)$, where $x_1 < x_2 \cdots < x_k$. Technically it is not a partition since the endpoints overlap. Let x_0 denote $-\infty$ and x_{k+1} denote ∞. There are points $s_0, s_1, \ldots, s_k \in S$ such that s_i is the unique closest point in S to all points in (x_i, x_{i+1}) on L. At x_i, both s_i and s_{i-1} are closest, and there may be others as well.

To find the Voronoi partition, note that the x-coordinate of s_i is less than that of s_{i+1} for all i. Suppose the points in S have been sorted by x-coordinate, and let S_1 denote the first half and S_2 the second. Let $a_0 < a_1 \cdots < a_k$ be the points of S_1 corresponding to the Voronoi partition induced by S_1, and $b_0 < b_i \cdots < b_\ell$ the points of S_2 corresponding to the Voronoi partition induced by S_2. It is easy to show that the Voronoi partition induced by S is of the form $a_0, \cdots, a_\alpha, b_\beta, \cdots b_\ell$. Using this, the Voronoi partition can be found using the paradigm in Section 7.1.

Lemma 7.2.3. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given* n *or fewer planar points, and given the equation of a line* L *such that all points are on one side of* L*, the Voronoi partition of* L *can be determined in* $\Theta(n^{(\frac{1}{2}-\epsilon)/3} \log^2 n)$ *time if the points are pre-sorted, and* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time otherwise.* □

We are now able to prove Proposition 7.2.2, using the following algorithm:

1. For a separating line L, find the Voronoi partition of L induced by A. Let a_I denote the point in A closest to interval I.
2. Do the same for B.
3. Perform an intersection operation on the Voronoi partitions: for each interval I in A's partition, and interval J in B's, such that $I \cap J \neq \emptyset$, determine $d(a_I, b_J)$.
4. The answer is $\min\{d(a_I, b_J)\}$.

Proof. First find a line L that separates A and B using the algorithm in Theorem 7.1.4, and then find the Voronoi partitions of L induced by A and B. For step 3, there are at most n intervals in these partitions. Sort them by their starting points. If tied, the interval from A comes the first. Each interval from A determines the length of the following processors it covers, i.e., the region before the next interval in A. This can be done by breaking each interval into two endpoints, sort all points, use a scan to determine the number of starting points from B between each two endpoints of A, and sort again using interval ID so that the starting points of A know the number of starting points from B they cover. Then do a segmented broadcast, and update the answer for each interval according to these intersections. This step will cover all intersections that starting points of the interval in B lies in the interval in A.

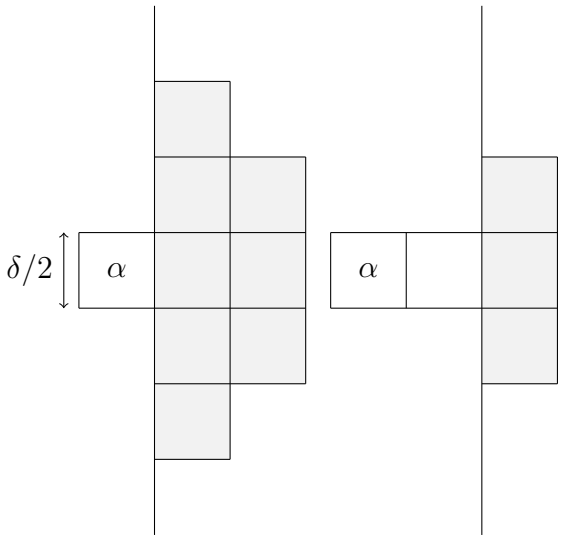
Then swap A and B. Do the previous steps again. This will cover all intersections that starting points of the interval in A lies in the interval in B. All previous steps together will cover all possible intersections. A global scan finds the answer. All steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

Proposition 7.2.2 allows us to prove Theorem 7.2.4.

Theorem 7.2.4. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given a set* S *of* n *or fewer labeled planar points, a closest dissimilar pair in* S *can be found in* $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ *time.*

Proof. We follow a well-known serial closest pair algorithm.

1. Sort the points by their x-coordinates.



(a) α is adjacent to the line (b) α is one block further

Figure 7.5: All squares that possibly contain a global closest dissimilar point from α

2. Evenly divide the points into a left and right half and in each half recursively find a closest dissimilar pair. Let δ be the closest distance found.

3. Consider a vertical strip of squares of edgelength $\delta/2$ along the dividing line, and squares at distance $\delta/2$ from the line, on both halves.

4. For each square α on the left which contains points from S, for each square β on the right side which has any part of its square within δ of α, determine the closest dissimilar pair, if any, with one point in α and the other in β.

5. Either this answer or the pair results in δ is the closest dissimilar pair found.

Note that if two points are within the same square they must have the same label since they are less than δ apart. For step 4, for any α along the line, note that β must be one of 8 squares (see Figure 7.5), and if it is in the strip further from the line then there are only 3. Consider a stencil of, say, a square adjacent to the dividing line on the left, and one on the right one square lower. For all pairs of such stencils, where the square on the left has at least one point of S, determine the closest dissimilar pair, if any exist, where if the square on the right has points of the same label on the right then it is ignored. Proposition 7.2.2 shows that computing this simultaneously for all pairs obeying such a stencil can be done in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. There are only 11 such stencils needed for all possible choices of α and β, and hence the time for this part of the recursion is $\Theta(n^{\frac{1}{2}-\epsilon})$. There are only a logarithmic

number of levels of recursion, and the sorting time remains the same at each level, and hence the total time is as claimed.

One subtle consideration is how the points are put into squares. Determine the minimum y-coordinate, y_{min}, of any point, and the maximum y_{max}. The squares are numbered upward from y_{min}, enumerated $0, 1, \ldots, \lfloor 2(y_{max} - y_{min})/\delta \rfloor$. Squares' x-coordinates go from 0 to 3. Let the concatenation of these coordinates be the square's index. For each point, create a record containing the point, with a key equal to its square's index. Sorting by the key puts the points in the same square into segments of the space-filling curve ordering, and then segmented broadcasts, reductions, and scans can be performed efficiently. □

It is unknown if the logarithmic factor can be eliminated in the worst case. However, in the expected case it can be.

Theorem 7.2.5. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given a set S of n or fewer labeled planar points, a closest dissimilar pair in S can be found in $\Theta(n^{\frac{1}{2}-\epsilon})$ expected time.*

Proof. The algorithm is based on Rabin's randomized closest pair algorithm [88], though there are some important differences. Throughout, keep track of the closest dissimilar pair found so far, and their distance.

1. Let x_{min} and x_{max} be the minimum and maximum x-coordinates of any point in S, and y_{min}, y_{max} be the corresponding y-coordinates.

2. Randomly pick a subset $R \subset S$ with $1/2$ of the points and recursively find a closest dissimilar pair in R. Let δ be the distance between this pair.

3. Partition the plane into squares of edgelength $\delta/2$, where (x_{min}, y_{min}) are the lower left coordinates of square $(0, 0)$, and put point $(x, y) \in S$ into square $(\lfloor 2(x - x_{low})/\delta \rfloor, \lfloor (2(y - y_{low})/\delta \rfloor)$.

4. For each square α with at least one point, for each neighboring square β which is within δ of α, find the closest dissimilar pair, if any, where one point is in α and the other is in β.

The steps are quite similar to the worst-case algorithm, but here a square α might have dissimilar pairs, something that could not happen in the previous algorithm. However, with high probability there are very few dissimilar pairs in α. Suppose the points in α are partitioned into sets T_1 and T_2 where all the labels in T_1 are different than those in T_2, and $k = |T_1| \leq |T_2|$. Since the edgelength of α is $\delta/2$, all pairs in α are closer than δ to each

other, and hence any pair with one point in T_1 and the other in T_2 is a closer dissimilar pair than any in R. The probability that no point in T_1 is in R is $1/2^k$ (technically the probability is slightly less), and that no point of T_2 is in R is no larger than this. Hence the probability that one or both occurred is no more than $1/2^{k-1}$, and this can be generalized to show that the probability that there are $2\log(n)+2$ points that have a label that differs from at least one other label is no more than $1/n$.

To exploit this, within each square broadcast the points which do not have the most common label. Do this one at a time, and use reduction to find the nearest point of a different label. There are at most n squares involved, so the probability that more than $2\log(k)+3$ broadcasts occur in any square is less than $1/2$, with the probability of additional broadcasts being needed decreasing geometrically. Using segmented broadcasts for the squares, and a global broadcast to determine if all squares are done, the expected time is $\mathcal{O}(n^{(\frac{1}{2}-\epsilon)/3}\log n)$.

Here are some additional details.

- The recursion in step 2 involves $n/2$ items, so sorting-based operations take half the time of sorting n items. Hence each step of the recursion takes half as long as the preceding one until the base mesh case is reached, which can be solved in time the square root of the mesh size. Thus the total expected time is $\Theta(n^{\frac{1}{2}-\epsilon})$.

- R is selected by each processor generating a random fair bit, independent of any other bits. Thus R may not be exactly $1/2$ the points, but the probability that it does not have at least 0.4, or more than 0.6, of them is tiny, and the analysis holds.

Therefore, the algorithm finishes in $\Theta(n^{\frac{1}{2}-\epsilon})$ expected time as claimed. □

A special case of Theorem 7.2.4 is that points are linearly separable. That is, given two sets, S_1 and S_2, of labeled points, find two points $a \in S_1$ and $b \in S_2$ such that $d(a,b)$ are minimized, and the labels for a and b differ. If S_1 and S_2 cannot have points with a same label, then the problem is the same as in Proposition 7.2.2. However, if they can, the problem becomes much harder. One can see that this problem can serve as a base case, and it is easy to get a divide and conquer algorithm with an extra logarithmic factor to solve Theorem 7.2.4 after this problem is solved. However, this problem is actually more complicated than the general one. This is so because any local information may be useless since points are separated by a line, and dissimilar pairs on the same side do not count. Theorem 7.2.6 gives a worst case optimal algorithm to solve it.

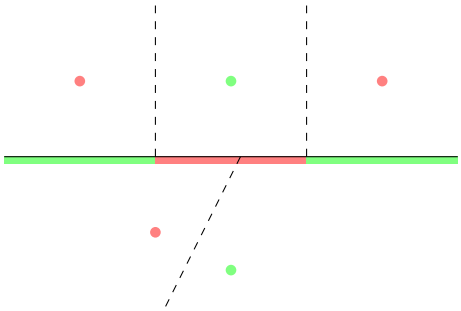


Figure 7.6: Modified version of Voronoi partition: intervals below with the same label must be ignored

Theorem 7.2.6. *On an ER MCC*$(n, n^{\frac{1}{2}+\epsilon})$ *with* $0 < \epsilon < \frac{1}{2}$*, given a two sets,* S_1 *and* S_2*, of* n *or fewer labeled planar points, a closest dissimilar pair from two sets can be found in* $\Theta(n^{\frac{1}{2}-\epsilon})$ *time.*

One key observation is that if we use the algorithm for Proposition 7.2.2 to solve the problem directly, and ignore the overlapping intervals from two sides with the same label, we should be able to find an answer pretty close. The only problem is as follows. We call an interval's label the label of the point that is closet to it. Assume that there is an interval I from S_1 with label s. In S_2, a point with label s may create an interval I_s that intersects with I on the line, and shadow another point with label t that is also very close to I_s. In this case, we may lose the best answer. However, this cannot happen on both side at the same time.

Consider a global answer (a, b) with $a \in S_1$ and $b \in S_2$. Assume that line segment $\overline{ab}$ intersects with line L at point p. It is impossible that both a and b are not the closest point to p on each side. This can be proved by contradiction. Assume both of them are not, then $c \in S_1$ and $d \in S_2$ are closer to p than a and b respectively. If c and d are of different labels, then (c, d) is closer than (a, b), and it contradicts with that (a, b) is the global answer. If c and d are of a same label, then either (a, d) or (b, c) is a better answer than (a, b). Therefore, this is impossible.

Therefore, the only necessary change is that we need to modify the algorithm for Voronoi partitioning so that it can ignore points with a specific label for covering specific intervals. These intervals are the Voronoi partition on the other side, but the algorithm does not need to know that (see Figure 7.6). This problem cannot be solved using the same method as in Lemma 7.2.3 because the monotonicity does not hold. Even if an interval I on the left side is not long enough to cover an interval I' on the right side, it is still possible that I has to cover intervals on the right of I' due to the label issue.

Apparently this problem has not yet been solved on a standard mesh-connected com-

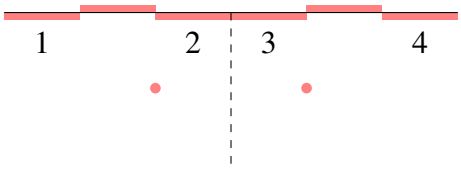


Figure 7.7: New intervals generated: a points and b intervals can generate at most $a + b$ intervals, intervals without labels above can have any label below

puter, thus we first give the algorithm on a mesh in Lemma 7.2.7, then give the algorithm on an MCC$(n, n^{\frac{1}{2}+\epsilon})$ in Lemma 7.2.8.

Lemma 7.2.7. *On a mesh-connected computer of size n, given n or fewer labeled planar points, the equation of a line L such that all points are on one side of L, and n or fewer non-intersecting labeled intervals on L, in $\Theta(n^{\frac{1}{2}})$ time one can find the Voronoi partition of L such that all resulting intervals do not have the same label with any given interval it intersects with. That is, if the closest point is of the same label, find one or more closest points with different labels.*

Proof. Assume that L is the x-axis. Otherwise, one can broadcast the equation of L, rotate and move all points, given intervals and L so that L becomes the x-axis.

Sort all points and intervals by their labels. If tied, use the coordinates as a tie-breaker. Do a scan to count how many points and intervals there are for each label. From now on, we define a label's size to be the number of points and intervals of this label. Sort all points and intervals by their label's size. If tied, use label as a tie-breaker, then coordinates. Note that a points and b intervals of a same label will result in at most $a + b$ intervals in the final answer (see Figure 7.7).

Consider the label l_i saved in processor $p_{n/2}$. Divide the mesh into three parts. All labels that go before l_i are in the same part, and all labels go after l_i are in another part, and l_i itself fills one part. Solve three parts in parallel on mesh. Note that it is almost impossible to keep the block format of the mesh, even if we put them in a space-filling curve ordering. However, it does not matter. On a mesh-connected computer of size $a \times b$, the problem can always be solved in $\Theta(a + b)$ time, if it can be solved in $\Theta((ab)^{\frac{1}{2}})$ time on a square mesh [89]. As long as in each iteration, we always divide the long edge of the current rectangular mesh, we can always decrease the run time $\Theta(a+b)$ by a factor of at least $1/4$. If a subproblem size is more than half of the current size, then it must be that all points and intervals in this subproblem are of a same label. In this case the problem can still be solved in $\Theta(a + b)$ time, using the algorithm for one label [19].

Finally there are three sets of intervals that need to be merged. Sort all resulting intervals by their left endpoints. For each interval, count the number of intervals that go after it and intersect with it. This can be done by finding the position of its right endpoints, and use sort to send such values. For each interval in p_i that covers a region of size L bigger than $n^{\frac{1}{2}}$, make $\lfloor L/n^{\frac{1}{2}} \rfloor$ extra copies of it, and use sort to move these values to processors $p_{i+n^{1/2}}$, $p_{i+2n^{1/2}}$, etc. Do a standard mesh rotation, so that each processor in range $[p_{i+1}, p_{i+n^{1/2}-1}]$ sees the intervals in p_i once, and they can update each other. Note that in the whole process, each processor will contain at most 4 different intervals at a time, because there are only three sets to merge, and thus each processor can have at most 3 of the moving copies, and one non-moving copy.

The total run time is $T(n) = T(\frac{3}{4}n) + \Theta(n^{\frac{1}{2}}) = \Theta(n^{\frac{1}{2}})$. □

Lemma 7.2.7 serves as a base case to solve the same problem on an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$.

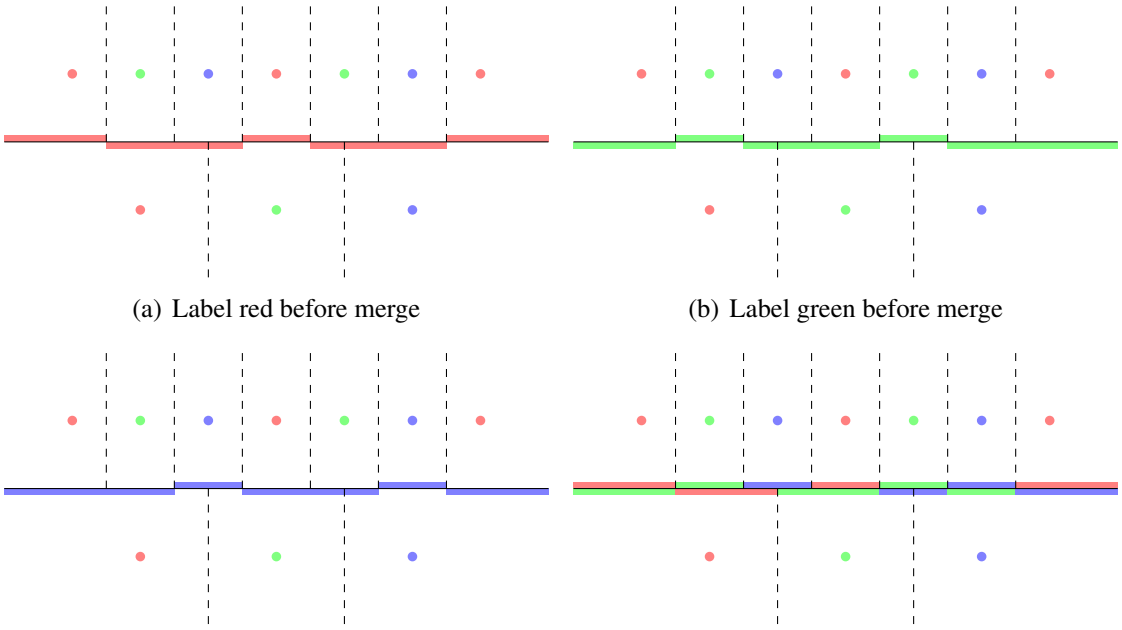
Lemma 7.2.8. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n or fewer labeled planar points, the equation of a line L such that all points are on one side of L, and n or fewer non-intersecting labeled intervals on L, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time one can find the Voronoi partition of L such that all resulting intervals do not have the same label with any given interval it intersects with. That is, if the closest point is of the same label, find one or more closest points with different labels.*

Proof. Assume that L is the x-axis. The algorithm solves the problem recursively. The boundary condition is that when solving the problem on an MCC(a, b) with $b \leq a^{\frac{1}{2}}$, use the standard mesh algorithm to finish the problem.

Sort all points and intervals by their labels. If tied, use the coordinates as a tie-breaker. Do a scan to count how many points and intervals there are for each label. Sort all points and intervals again by their label's size. If tied, use labels as a tie-breaker, then coordinates.

Divide the mesh into blocks, and try to make each block size as close to $n^{\frac{1}{2}+\epsilon}$ as possible. This can be done as follow. For all labels that have size greater than or equal to $n^{\frac{1}{2}+\epsilon}$, give each $n^{\frac{1}{2}-\epsilon}$ values a channel, and use a recursive call to solve them independently. We know that each of these blocks contain a single label, so the problem can be solved in parallel in $\Theta(n^{\frac{1}{2}-\epsilon})$ total time using the algorithm in Lemma 7.2.3, regardless of the block size. Note that slight modifications are required to accommodate the non-intersecting interval requirement. This is easy since all of them have the same label. One can just find the Voronoi partition without considering such requirement, and cut all intervals at the end.

For those labels which have size smaller than $n^{\frac{1}{2}+\epsilon}$, we use the same idea as in the standard mesh algorithm. Consider the first label saved in processors $p_{n^{1/2+\epsilon}}$, $p_{2n^{1/2+\epsilon}}$, etc.



(a) Label red before merge

(b) Label green before merge

(c) Label blue before merge

(d) Final Voronoi partition

Figure 7.8: Updates in the Voronoi partitioning

One can either divide the mesh using these labels, which will result in all block sizes strictly smaller than $n^{\frac{1}{2}+\epsilon}$, or put these labels into the previous block, which will result in all block sizes strictly smaller than $2n^{\frac{1}{2}+\epsilon}$. In any case, solve the problem using a recursive call on an MCC$(n^{\frac{1}{2}+\epsilon}, n^{2\epsilon})$.

After we calculate all the intervals in all blocks, we need to merge. Note that if two intervals do not intersect, they will have no effect on each other. All intersected intervals can only make each other shorter or unchanged, and no intersections can ever break an interval into two. Thus we only need to loop through all pairs of possible intersections between intervals from different blocks, and each update takes only constant time (see Figure 7.8).

We know that there are at most $n^{\frac{1}{2}-\epsilon}$ blocks. For two blocks of k_i and k_j intervals, we know that there are at most $k_i + k_j$ intersections. Thus, for the block that has k_i intervals, the total number of intersections it has is $\sum_{j\neq i}(k_i + k_j) = \mathcal{O}(n^{\frac{1}{2}-\epsilon} \cdot k_i + n)$. Thus the total number of intersections for all blocks is $\sum_i (n^{\frac{1}{2}-\epsilon} \cdot k_i + n) = \mathcal{O}(n^{\frac{1}{2}-\epsilon} \cdot n)$. This means that we should be able to use a standard mesh rotation to finish the problem within $\Theta(n^{\frac{1}{2}-\epsilon})$ time.

Here is how to do it. Sort all intervals by their starting positions, and break ties using their original processor IDs. For each interval, find the last interval that goes after it and intersects with it. This can be done by finding its right endpoint. For an interval that covers

K later intervals, make $\lfloor K/n^{\frac{1}{2}-\epsilon} \rfloor$ extra copies of it. This is nontrivial since one processor may have more than a constant number of values if this step is not done carefully. We can use a scan to count the total number of copies, then a sort followed by a segmented broadcast to make copies, since the total number of copies is $\mathcal{O}(n^{\frac{1}{2}-\epsilon} \cdot n/n^{\frac{1}{2}-\epsilon}) = \mathcal{O}(n)$.

We give each copied interval an ordered pair (p_d, p_o), which means that it should goes to processor p_d, and it is a copy of the interval in processor p_o. Note that all $p_d = p_o + k \cdot n^{\frac{1}{2}-\epsilon}$ for some k. We also give each non-copied interval an ordered pair, where both p_d and p_o are its position. Sort all copied intervals along with the original intervals together by their ordered pair. Note that each processor now contains at most two intervals. At this stage, use a mesh rotation algorithm to deal with all possible intersections. Although each interval can still intersect with more than $n^{\frac{1}{2}-\epsilon}$ other intervals, one can easily see that we can ignore such cases. We only need to go for a distance of $2n^{\frac{1}{2}-\epsilon}$, and this will cover all possible intersections.

Finally sort all intervals using their left endpoints, and do a segmented reduction to gather the final answer for each interval. All steps other than the recursive calls finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. Using the same analysis as in sort, we know that the depth of recursive calls is a constant. Therefore, all steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

The linearly separable closest dissimilar pair problem is easy to solve with Lemma 7.2.8.

Proof of Theorem 7.2.6. First find the Voronoi partition on both side without caring about the labels. The problem now can be solved as follow. Mark each interval generated by a point in S_1 using the label of the point that is closest to it. There are at most n of such intervals. Using these intervals as the interval inputs, use Lemma 7.2.8 to find the Voronoi partition from points in S_2. At most $2n$ intervals will be generated. Find all possible intersections between the two sets of intervals, and find the global answer.

Swap S_1 and S_2, and do the same process as above again. The global answer is now found. All steps finish in $\Theta(n^{\frac{1}{2}-\epsilon})$ time. □

One final concern is that the Voronoi partition is not a partition, since the intervals intersect at the endpoints. What's more, for the endpoints, it is possible that more than two points have a same closest distance to it. However, they cannot give global optimal answers. See Figure 7.9 for more details.

Extending the above to points in d-dimensional space, $d \geq 3$, is quite easy for all-nearest neighbors, taking the same time though the implied constants are a function of d and ϵ. Closest dissimilar pair is more complicated because it requires extending Proposition 7.2.2. For 2 dimensions, the intervals on the line correspond to a cut through the

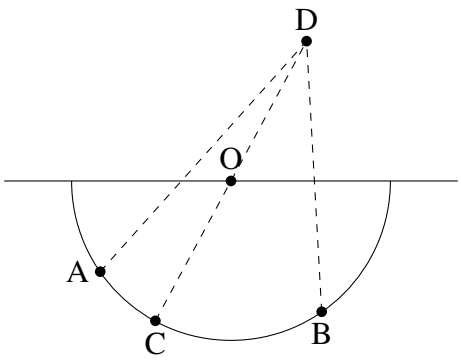


Figure 7.9: More than two points are closest to a point on L: C must be in between A and B, and $\overline{CD}$ must go through O; but in this case $\overline{CD}$ cannot be the optimal answer, because both $\overline{AD}$ and $\overline{BD}$ are shorter than it

Voronoi diagram of the points. Extending to higher dimensional points the cut needs to be a $(d-1)$-dimensional cut of a d-dimensional Voronoi diagram. As d increases the complexity of finding this rapidly increases for the fastest algorithms known, with one of the problems being that for $d \geq 4$ the size of the Voronoi diagram can be superlinear in n. The time and space required for MCC algorithms for this problem are open questions, and are open even for PRAMs and meshes.

7.3 Line Segment Intersection

The problems considered in this section involve planar line segment intersections. This is a fundamental problem in computational geometry [81], and there are many output sensitive parallel algorithms that can find all intersections [90, 91]. Here we only consider the problem of deciding if at least one intersection exists. Theorem 7.3.1 gives an algorithm to efficiently solve the problem, using splitter sort from Section 4.1.3.

Theorem 7.3.1. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n or fewer labeled line segments, if no two line segments with the same label intersect other than at endpoints, then in $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$ time it can be determined if there are any intersections of line segments with different labels.*

Proof. The proof uses divide and conquer. First each processor with a labeled line segment $\overline{ab}$ creates two line segment records, one has the x-coordinate of a as key, while the other has the x-coordinate of b as key. Both records save $\overline{ab}$ as data. Sort all records by their key values. Note that each processor will save two records. Divide the mesh into four blocks, where each block contains a vertical region of the plane. The keys of the first record in

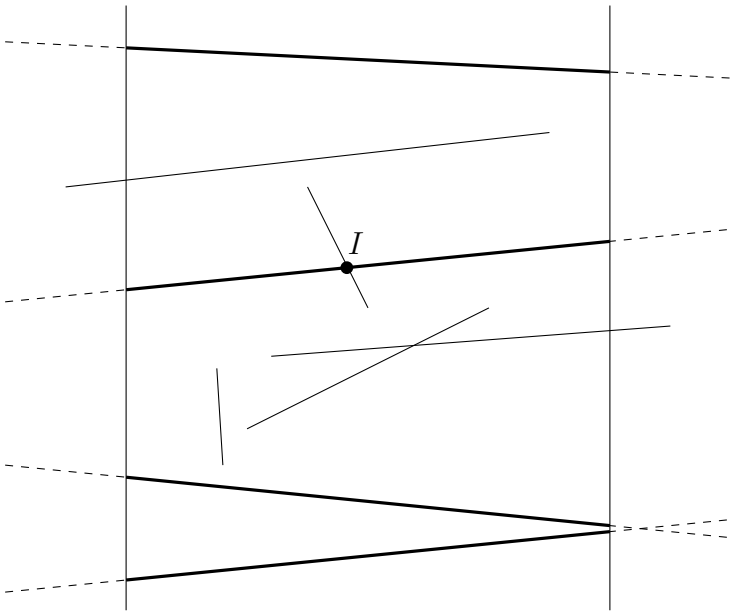


Figure 7.10: Sort line segments using partial order sort, bold lines are in S_1 and others are in S_2; only one intersection detected is I, and intersections between two line segments in S_2 cannot be detected at this stage

processors $P_{n/4}$, $P_{n/2}$, and $P_{3n/4}$ are used as region boundaries. They divide the plane into four vertical slabs. Broadcast the four slabs' information to all processors.

If a line segment completely goes across a slab, we call this line segment a *spanning line segment*. For each spanning line segment, create a *spanning line record* that is the part of the line segment lies inside the slab. Note that each line segment can create at most four spanning line records. There will be at most $4n$ spanning line records in total, and n in each slab. Do the following for four slabs in sequence.

For each slab, we need to first determine if there are any intersections between all spanning line records. This can be done by two sorts and one check. Sort the spanning line records according to their left endpoints' y-coordinates, then sort them by the right endpoints' y-coordinates. If there is a tie, use the initial processor ID that the line segment came from as a tie-breaker. After that, we only need a simple check to see if these two sorted orders are the same. If not, there must be an intersection between two line segments with different labels, since we assume that there are no intersections between segments with the same label other than at endpoints. In this case, the algorithm is finished.

We now consider intersections between spanning line records and other line segments. The spanning line records divide the slab into non-overlapping regions. In the slab, using splitter sort, sort all spanning line records and line segments. Note that spanning line records have total ordering, but regular line segments do not since they are represented by

points in the slab (see Figure 7.10). After the sort, we can use a segmented broadcast to broadcast spanning line records, so that all line segments know the spanning lines immediately below and above it. Each segment checks if it intersects with any of these spanning line records. If any do, the algorithm finishes by returning true. This is correct because all line segments that have the same label cannot intersect with each other, other than at the endpoints.

After the previous step, all possible intersections with the current spanning line records are found. If no intersections were found, remove all spanning line records and solve the subproblems in the four slabs.

Since each iteration reduces the recursive call by a fixed fraction, there are $\Theta(\log n)$ iterations. Each uses sort, so the total time is $\Theta(n^{\frac{1}{2}-\epsilon} \log n)$. □

A special case is when the points are unlabeled. Interpreting the point as its own label, the algorithm determines if there are any segment intersections. With minor changes, the algorithm can be modified to ignore intersections involving the endpoints.

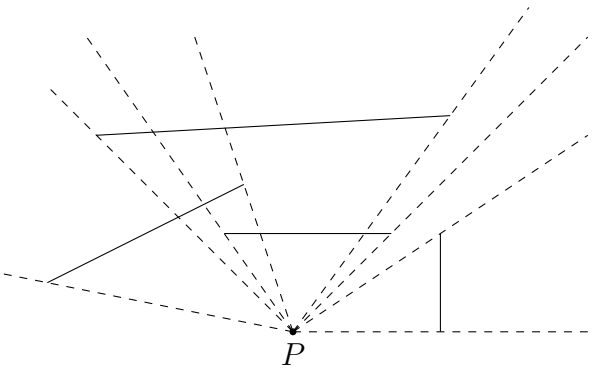
Related problems concern visibility. In the 2D plane, given a set of line segments and two points P and Q, Q is *visible* by P if and only if the line segment $\overline{PQ}$ does not intersect any other line segments. Given n non-intersecting line segments and a point P, the *visibility problem* is to determine which parts of each line segment are visible to P. The visibility problem has been well-studied, and is especially useful for planar shortest path problems [83, 92].

Theorem 7.3.2. *On an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ with $0 < \epsilon < \frac{1}{2}$, given n or fewer line segments which can only intersect at endpoints, and given a point P, in $\Theta(n^{\frac{1}{2}-\epsilon})$ time all pieces of the line segments that are visible from P can be found.*

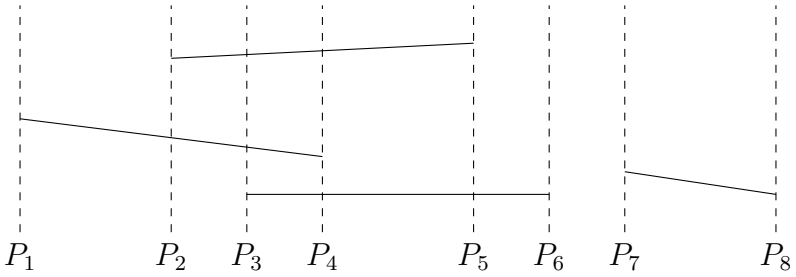
Proof. There are at most $2n$ endpoints from n line segments. If we draw a ray from P to each endpoint, we can see that the problem is very similar to the following one (see Figure 7.11): given n planar line segments above the x-axis, which can only intersect at endpoints, find the portions of the segments visible from the x-axis. We don't need to do any real transformation, but it is used to simplify exposition.

From now on, consider the visibility problem from the x-axis. The x-coordinates of the segments create at most $2n$ vertical slabs (there may be ties of x-coordinate). In each, find the lowest line segment in the slab. Since line segments can intersect only at endpoints, their ordering is well-defined in each slab. Also note that there could be n line segments in each slab.

Make two copies of each line segment, where one uses the left endpoint as a key and the other uses the right endpoint. Sort these by their keys. Divide the mesh into blocks of



(a) Visibility from a point



(b) Visibility from the bottom

Figure 7.11: Two visibility problems

size $n^{\frac{1}{4}} \times n^{\frac{1}{4}}$. Solve the problem recursively in each block using only line segments in the block.

We now need to deal with the updates caused by spanning line segments, which are line segments that completely cross a block. Broadcast the $n^{\frac{1}{2}}$ block boundaries to all blocks, taking $\Theta(n^{\frac{1}{2}-\epsilon})$ time. In the block, create a record representing the right block boundary.

In each block, sort all line segments along with the block boundaries by their rightmost x-coordinates, with block boundaries preceding segments in case of ties. Using a scan, each line segment can determine the index of the rightmost block that it spans. In each block, sort by y-coordinate of where segments cross the right edge of the block. If segment

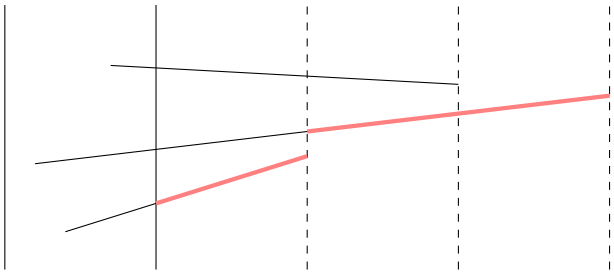


Figure 7.12: Blocking visibility

I precedes segment J then, since they don't intersect, I is always below J and obstructs its visibility in any block they both span (see Figure 7.12). Now perform a scan of the rightmost block indices. If only this block's segments are considered, a segment J is only visible from the block to the right of the one it received from the scan, through to J's rightmost block. Discard intervals that aren't visible in any block they cross, and resort the interval and block boundary records by their block index, with intervals preceding blocks. A scan can now be used to inform each block boundary record of the lowest line segment, if any, from this block that crosses the block the record represents. Use a global sort, move the calculated bottom spanning line segments in each block to the block that they should be in. Do a reduction in each block to find the global bottom spanning line segment, and use a broadcast to update the values in the block.

All steps finish in $\mathcal{O}(n^{\frac{1}{2}-\epsilon})$ time. The recursion stops when the block size m is smaller than $n^{1-2\epsilon}$, which is when there is less than $m^{\frac{1}{2}}$ channels per block. At this point a mesh algorithm is used, taking $\Theta(m^{\frac{1}{2}}) = \Theta(n^{\frac{1}{2}-\epsilon})$ time. Thus the total time is as claimed. □

CHAPTER 8

Coarse-Grained Algorithms

Coarse-grained problems have much harder communication problems to deal with, and some of them are not observed in the fine-grained problems at all. For instance, consider the redistribution operation. The fine-grained version is almost trivial. However, in the coarse-grained model, we observe two more circumstances: the algorithm needs to guarantee that in each iteration, only one value is read/written from/to each processor. Another example lies in the sort algorithm, when collecting information from each block. The number of values that we need to maintain may be greater than the number of processors, thus special considerations are required.

Initially we believed that after we solved the fine-grained problems, the coarse-grained ones are just a balance between communication and local work, since the fine-grained algorithms should have taken care of all communication details. Later we found that this is not true. In this Chapter, we give several algorithms in different areas to show how coarse-grained algorithms are different from their fine-grained counterparts, and why they are hard. Throughout, we assume that the number of processors p is smaller than or equal to the input size n.

8.1 Broadcast, Reduction and Scan

We first solve these three basic operations on a coarse-grained model. These are the easy problems whose solutions are essentially the same as their fine-grained solutions. One can see that we only need to first solve the serial problem in each processor, and then the problem left is a fine-grained problem on an MCC(p, p^ϵ). The segmented version is the same. The results are given in Proposition 8.1.1. Recall that we use MCC$(n, p, S(p))$ to represent a computer with base mesh of size p and $\Theta(S(p))$ global communication channels, with input size n.

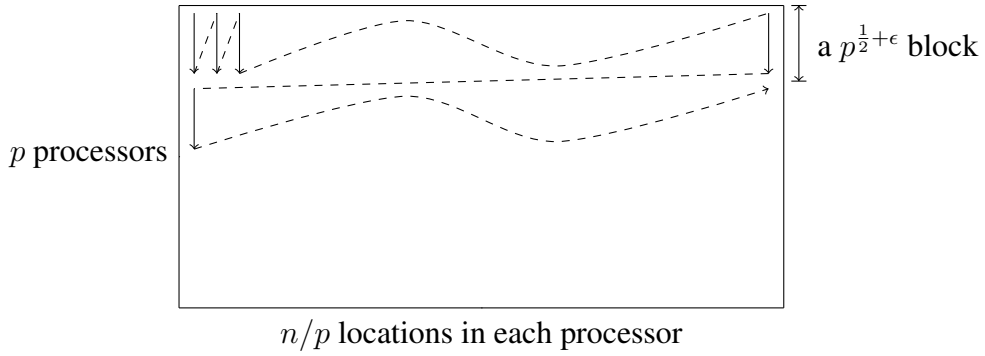


Figure 8.1: Destination ordering

Proposition 8.1.1. *On an ER MCC(n, p, p^ϵ) with $1 \leq p \leq n$ and $0 < \epsilon < 1$, broadcast, reduction and scan can be done in $\Theta(n/p + p^{\frac{1}{3}(1-\epsilon)})$ time.* □

8.2 Sort

We now do sort on a coarse-grained model. The mesh algorithm is given in Theorem 8.2.1. Though not described in this way, we want $p = \Omega(\log^2 n)$. Otherwise, the serial sort in each processor takes $\Theta(n/p \cdot \log n)$ time and will dominate the run time. This is true for all algorithms that require an extra logarithmic factor in serial. The coarse-grained mesh sort algorithm is essentially the same as the fine-grained algorithm [19], and the proof is skipped here.

Theorem 8.2.1. *On a mesh-connected computer of size p with $1 \leq p \leq n$, given n/p numbers in each processor, sort can be done in $\Theta(n/p^{\frac{1}{2}} + n/p \cdot \log n)$ time.* □

On an ER MCC, recall the sort algorithm in fine-grained model described in Theorem 4.1.2 that used multiple splitters. First divide the mesh into several small parts, and sort all parts in parallel. Then we choose some splitters, and divide the data values into several groups according to splitters. We know that all values in the same group should be close to each other at the end, and we use redistribution to do that. Finally we sort in each group in parallel to finish the problem. Theorem 8.2.2 gives an algorithm to sort in a coarse-grained model.

Theorem 8.2.2. *On an ER MCC$(n, p, p^{\frac{1}{2}+\epsilon})$ with $1 \leq p \leq n$ and $0 < \epsilon < \frac{1}{2}$, sort can be done in $\Theta(n/p^{\frac{1}{2}+\epsilon} + n/p \cdot \log n)$ time.*

Our algorithm sorts the values and saves them in the standard ordering. That is, filling the first processor first, then the second, and so on. However, in some intermediate steps, it

requires a non-standard ordering. There are p processors, and each processor contains n/p values. We use (i, j) to represent a value in processor $0 \leq i < p$ and position $0 \leq j < n/p$. We define the *destination ordering* to be as follows: $(0,0) < (1,0) < (2,0) < \ldots < (p^{\frac{1}{2}+\epsilon} - 1, 0) < (0,1) < (1,1) < \ldots < (p^{\frac{1}{2}+\epsilon} - 1, 1) < \ldots < (p^{\frac{1}{2}+\epsilon} - 1, n/p - 1) < (p^{\frac{1}{2}+\epsilon}, 0) < (p^{\frac{1}{2}+\epsilon} + 1, 0) < \ldots$ (see Figure 8.1). That is, first we focus on processors from 0 to $p^{\frac{1}{2}+\epsilon} - 1$, and the ordering is as shown above. After all values from the first $p^{\frac{1}{2}+\epsilon}$ processors are in the ordering, we move on to the next $p^{\frac{1}{2}+\epsilon}$ processors, and so on. This ordering is used in redistribution to avoid concurrent read/write to the same processor.

Note that the new ordering can be used to put in any other desired orderings within a sort time, and vice versa. Each position in our ordering can calculate its position in any other regular ordering in $\mathcal{O}(\log n)$ time. After that, we only need an extra sort, i.e., sort using the positions in the desired ordering as keys.

We still need to solve redistribution before sort. Since we are using a coarse-grained model, we define an extended redistribution operation. On a mesh-connected computer of size $p^{\frac{1}{2}} \times p^{\frac{1}{2}}$, where each processor has n/p data values, and each value is associated with a destination position (i, j), assume that all the destination positions are different, the redistribution operation moves all data values to their destinations in parallel.

Lemma 8.2.3. *On an ER MCC*$(n, p, p^{\frac{1}{2}+\epsilon})$ *with* $1 \leq p \leq n$ *and* $0 < \epsilon < \frac{1}{2}$*, redistribution can be done in* $\Theta(n/p^{\frac{1}{2}+\epsilon} + n/p \cdot \log n)$ *time.*

The fine-grained version of Lemma 8.2.3 is easy to prove: we just need $n^{\frac{1}{2}-\epsilon}$ rounds, and in each round we move $n^{\frac{1}{2}+\epsilon}$ values. The coarse-grained version is not as straightforward, since now we have two more conditions to deal with. That is, we need to make sure that

1. in each iteration, we take out at most one value from each processor, and
2. in each iteration, we write at most one value to each processor.

At the same time, we need each processor to know when to read from and write to the channels beforehand, just like in the fine-grained models. These three things make the problem difficult.

Proof. The second condition is easy to deal with. Using destination ordering, in each iteration, we move $p^{\frac{1}{2}+\epsilon}$ consecutive values, then it is obvious that there is no more than one value written to each processor in each iteration, and all processors will know when to read from and write to the channels. The second condition is resolved.

The hard part is the first condition. If we do as described above, it is possible that in one iteration, more than one value is required from a processor, then this iteration may

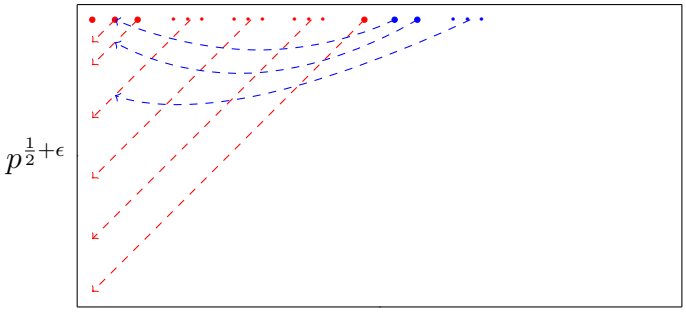


Figure 8.2: Shuffle the array

not be done in constant time, and the algorithm fails. Resolving this situation requires an observation that if we can first divide the mesh into $p^{\frac{1}{2}-\epsilon}$ blocks of size $p^{\frac{1}{2}+\epsilon}$, and inside each block we can somehow shuffle data round, then we can guarantee that there exists a layout of values such that in our redistribution process, in each iteration, there is at most one value taken out from each processor.

This can be achieved in the following way: sort all values by their destinations in each block according to the destination ordering, and save them in destination ordering. To better understand this, let us break this step into two. Given a block of size $p^{\frac{1}{2}+\epsilon}$, first sort all values by their destinations according to the destination ordering, and save them in the standard ordering. We know that now almost all values that need to be moved in each iteration are in a same processor. Now we look at the values in each block as a $p^{\frac{1}{2}+\epsilon} \times n/p$ matrix. We know that the values are now sorted in row major ordering. In row major ordering, we numbered all values from 0 to $n/p^{\frac{1}{2}-\epsilon} - 1$. Now we are going to put values with number mod $p^{\frac{1}{2}+\epsilon}$ equals to i in the ith processor. If multiple values are in the same processor, their ordering remains the same as before.

Another way to describe this is that we read the matrix in row major ordering, and write the values in column major ordering. More precisely, the first value goes to $(0, 0)$. The second value, which was previously at position $(0, 1)$, now goes to $(1, 0)$, and so on (see Figure 8.2). In this way, we can guarantee that in each iteration, no two required values are from a same processor. This can be proved by contradiction. Assume not, then there exists two values a, b from an iteration that are from a same processor. We know that there are at least $p^{\frac{1}{2}+\epsilon} - 1$ values in destination ordering that are between a, b, and all of them must be in this iteration. Then we get a contradiction, since this iteration now has at least $p^{\frac{1}{2}+\epsilon} + 1$ values.

Now we divide the problem into two cases: first case with $0 < \epsilon \leq \frac{1}{6}$, and second case with $\frac{1}{6} < \epsilon < \frac{1}{2}$. The main difference between two cases is how to shuffle the data.

Case 1 is when $0 < \epsilon \leq \frac{1}{6}$. This is the easy case. Divide the mesh into blocks of size $B = p^{\frac{1}{2}+\epsilon}$, and use standard mesh algorithm to sort it in order to achieve shuffling. The number of values inside each block is $n' = p^{\frac{1}{2}+\epsilon} \times n/p$, and the number of processors is $p' = p^{\frac{1}{2}+\epsilon}$. Thus the time it takes other than sorting in each processor is $\Theta(n'/\sqrt{p'}) = \Theta(p^{\frac{1}{2}+\epsilon} \times \frac{n}{p}/\sqrt{p^{\frac{1}{2}+\epsilon}}) = \Theta(\frac{n}{p}\sqrt{p^{\frac{1}{2}+\epsilon}})$. We want this time to be $\mathcal{O}(n/p^{\frac{1}{2}+\epsilon})$, and this requires $\frac{n}{p}\sqrt{p^{\frac{1}{2}+\epsilon}} \leq n/p^{\frac{1}{2}+\epsilon}$. This is equivalent to $p^{\frac{3}{2}+3\epsilon} \leq p^2$. This is true when $0 < \epsilon \leq \frac{1}{6}$.

Case 2 is when $\frac{1}{6} < \epsilon < \frac{1}{2}$. In this case we use our MCC$(n, p, p^{\frac{1}{2}+\epsilon})$ algorithm to sort values inside a block of size $p^{\frac{1}{2}+\epsilon}$. However, we have not introduced our algorithm yet, and also our algorithm depends on redistribution.

We can still solve the problem. Each block still has size $B = p^{\frac{1}{2}+\epsilon}$, and can use $p^{\frac{1}{2}+\epsilon}/(p/B) = p^{2\epsilon}$ channels. We know that $p^{2\epsilon} = p^{(\frac{1}{2}+\epsilon)(\frac{1}{2}+\epsilon')}$, with $0 < \epsilon' < \epsilon < \frac{1}{2}$ when $\frac{1}{6} < \epsilon < \frac{1}{2}$. Therefore, by induction our algorithm will work. After at most $\mathcal{O}(\frac{1}{1/2-\epsilon})$ iterations, we will be in the first case. Note that the number of values inside each block is $n' = n/p \times B = n/p^{\frac{1}{2}-\epsilon}$, the number of processors is $p' = B = p^{\frac{1}{2}+\epsilon}$, and the number of channels that each block can get is $p^{2\epsilon}$. Note that the number of values inside each processor is n'/p', and we actually do not care what p' is. We only care about the size of the channels comparing to p'. As long as $p^{2\epsilon} = p'^{\frac{1}{2}+\epsilon'}$, we can apply our algorithm. Thus, the run time is still $\Theta(n'/p^{2\epsilon}) = \Theta(n/p^{\frac{1}{2}+\epsilon})$ by induction. □

With the help of Lemma 8.2.3, we are now able to prove Theorem 8.2.2.

Proof of Theorem 8.2.2. The proof contains two cases: first case with $0 < \epsilon \leq \frac{1}{6}$, and second case with $\frac{1}{6} < \epsilon < \frac{1}{2}$. The main difference between two cases is how to do the base sort. For simplicity, let us assume that $p = \Omega(\log^2 n)$ to shadow the $\Theta(n/p \cdot \log n)$ term in the time complexity. One can see that the following algorithm works when such condition does not hold, but the time analysis will be much longer.

Case 1 is when $0 < \epsilon \leq \frac{1}{6}$. The algorithm uses multiple splitters and takes 11 steps. Recall that the processor ordering is a space-filling curve.

1. Divide the mesh into p/B blocks of size $B = p^{\frac{1}{2}-\epsilon} \times p^{\frac{1}{2}-\epsilon}$. Sort the values within each block in parallel using only mesh. The number of values inside each block is $n' = B \times n/p$, and the number of processors is $p' = B$. Thus the run time is $\Theta(n'/\sqrt{p'}) = \Theta(n/p^{\frac{1}{2}+\epsilon})$.

2. Let $S = p^{2\epsilon}$ and in each block select every Sth value as a splitter. The total number of splitters is n/S. Note that the number of values inside each block is equal to the number of splitters, since $n/p \cdot B = n/S$ when $SB = p$.

3. For each position (i, j), we define its ID to be its position in destination ordering. Distribute all the splitters to each block as follows. Each processor with at least one splitter first determines which channel to use. It may need to use channels several times if it contains more than one splitter. The splitter processors send their values in parallel p/B times, sending to the blocks in order. The processor that contains position ID i in the block receives the value sent by the processor with the ith splitter. Note that this may result in one processor sending more than one value in each iteration. This can be resolved using the same method as in the redistribution operation, and the details are omitted here. Each block takes $\Theta(n/(S \cdot p^{\frac{1}{2}+\epsilon}))$ time. Thus this step takes $\Theta(n/(S \cdot p^{\frac{1}{2}+\epsilon}) \cdot p/B) = \Theta(n/p^{\frac{1}{2}+\epsilon})$ total time.

4. Inside each block, sort the splitters along with all data values using standard mesh sorting. Note that there are $2n/p$ values inside each processor, but the run time is still $\Theta(n/p^{\frac{1}{2}+\epsilon})$.

5. Within each block, divide data values into n/S groups, where each group contains the values between consecutive splitters. Count the number of values in each group, and for each value calculate the rank of itself within its group. This step can be done by a segmented scan in $\mathcal{O}(n/p^{\frac{1}{2}+\epsilon})$ time.

6. Denote the number of data values between the ith and $(i+1)$st splitter inside block b by $C(b, i)$. We can set splitter -1 to be $-\infty$, and splitter n/S to be ∞ to fix the boundary condition. Recall position ID that is defined in step 3. $C(b, i)$ is stored in the ith position of block b.

7. Processors that contain position ID i, $0 \leq i \leq n/S - 1$, creates a variable $N(i)$ that is initially 0. The blocks sequentially utilize the channels. Do the following substeps for p/B big iterations, one iteration for each block. In each big iteration, let b denote the current block. A big iteration contains $n/(S \times p^{\frac{1}{2}+\epsilon})$ small iterations. In the jth small iteration ($0 \leq j < n/(S \times p^{\frac{1}{2}+\epsilon})$), do the following.

   - Processors that contain position i, $jp^{\frac{1}{2}+\epsilon} \leq i \leq (j+1)p^{\frac{1}{2}+\epsilon} - 1$, writes $N(i)$ to channel $i - jp^{\frac{1}{2}+\epsilon}$, and this is read by the processor that has position i in block b;
   - the receiving processor stores this value, call it $N(b, i)$, and sends back $N(b, i) + C(b, i)$ using the same channel;
   - processors that contain position i reads the value sent and updates $N(i)$.

   This step takes $\Theta(p/B \times n/(S \times p^{\frac{1}{2}+\epsilon})) = \Theta(n/p^{\frac{1}{2}+\epsilon})$ time.

8. After going through all blocks, do a scan in positions $0 \leq i \leq n/S - 1$ to compute $M(i) = \sum_{j=0}^{i-1} N(j)$. Thus $M(i)$ is the total number of values less than the ith splitter. Go through all blocks again, sending them the M values. This step takes the same time $\Theta(n/p^{\frac{1}{2}+\epsilon})$ as the previous step.

9. Using the values from step 7 and 8, every position determines a destination ID for the value it contains. To determine the destination for a position in block b, suppose its value has rank r in the group of positions with value between the ith and $(i+1)$st splitter. Then its destination position ID is $M(i)+N(b,i)+r$. That is, it will go after all values less than the ith splitter, and after all values between the ith and $(i+1)$st splitter in preceding blocks, and after the values with smaller ranks in its group. This step is local calculation inside each block, and can be done in $\mathcal{O}(n/p^{\frac{1}{2}+\epsilon})$ time.

10. Redistribute all values. This step takes $\Theta(n/p^{\frac{1}{2}+\epsilon})$ time.

11. There are at most $n' = p/B \cdot S$ values between every two consecutive splitters, and these values in space-filling curve ordering must be covered by a block of size at most $p' = 4n'/(n/p)$. Sort all of these large blocks in parallel, taking $\Theta(n'/\sqrt{p'}) = \Theta(\sqrt{n}/p^{\frac{1}{2}-2\epsilon})$ time.

The algorithm finishes in $\Theta(n/p^{\frac{1}{2}+\epsilon} + \sqrt{n}/p^{\frac{1}{2}-\epsilon})$ time. When $0 < \epsilon \leq \frac{1}{6}$ and $p \leq n$, this is $\Theta(n/p^{\frac{1}{2}+\epsilon})$.

Case 2 is when $\frac{1}{6} < \epsilon < \frac{1}{2}$. The algorithm uses the same idea as before, but it requires recursive calls.

1. Divide the mesh into p/B blocks of size $B = p^{\frac{1}{2}+\epsilon}$. Sort the data values within each block in parallel using a recursive call. When doing the recursive call, uniformly split the $p^{\frac{1}{2}+\epsilon}$ channels to p/B blocks, and thus each block gets $p^{2\epsilon}$ channels. Each block uses its own channels to help its sort. This step takes $\Theta(T)$ time, where T is determined later.

2. Let $S = p^{\frac{1}{2}-\epsilon}$ and in each block select every Sth value as splitters. In total there are n/S of them. Note that $n/p \times B = n/S$. That is, the number of splitters equal to the number of data values inside each block.

3. Do step 3 in case 1. The number of blocks here is p/B, and thus it takes $\Theta(p/B \times n/(S \times p^{\frac{1}{2}+\epsilon})) = \Theta(n/p^{\frac{1}{2}+\epsilon})$ time.

4. Inside each block, sort the splitters along with data values using a recursive call. Note that the sort algorithm will have a constant factor of overhead. This step takes $\Theta(T)$ time.

5. Divide the data values into n/S groups, with each group contains only values between consecutive splitters. Count the number of data values in each group, and calculate the rank of values in their group. This step takes $\mathcal{O}(T)$ time.

6. Do step 6 through 8 in case 1. The only modification here is that S and B are different from their previous values. However, we know that $SB = p$ in both cases, and this will result in that two cases give the same run time. This step takes $\Theta(n/p^{\frac{1}{2}+\epsilon})$ time.

7. Using the values from step 6, all values inside each block can calculate a destination processor ID. This step takes $\mathcal{O}(n/p^{\frac{1}{2}+\epsilon})$ time.

8. Redistribute all values. This step takes $\Theta(n/p^{\frac{1}{2}+\epsilon})$ time.

9. We know that there are at most $n' = p/B \cdot S$ values between every two consecutive splitters. We also know that $p/B \cdot S$ values in a space-filling curve ordering must be covered by a block of size at most $B' = 4(p/B \cdot S)/(n/p)$. Therefore, first use a scan to determine the block sizes, and then determine the number of channels each block is going to use. Then sort all blocks in parallel using a recursive call. This step takes $\Theta(n'/\sqrt{B'}) = \Theta(\sqrt{nS/B}) = \Theta(\sqrt{n}/p^{\epsilon})$ time.

The algorithm takes $\Theta(n/p^{\frac{1}{2}+\epsilon} + \sqrt{n}/p^{\epsilon} + T)$ time. First notice that $n/p^{\frac{1}{2}+\epsilon} \geq \sqrt{n}/p^{\epsilon}$, since it is equivalent to $n \geq p$. To determine T, consider step 1. Each block of size $B = p^{\frac{1}{2}+\epsilon}$ has $p^{2\epsilon}$ channels and $n' = n/p \times B$ data values. When $\frac{1}{6} < \epsilon < \frac{1}{2}$, we have $\frac{2}{3} < \frac{1}{2} + \epsilon < 1$, and $\frac{1}{3} < 2\epsilon < 1$. More precisely, no matter what ϵ is, we can always do a recursive call, since we have $2\epsilon = (\frac{1}{2} + \epsilon)(\frac{1}{2} + \epsilon')$, with $0 < \epsilon' < \epsilon < \frac{1}{2}$. Thus by induction, our algorithm works, and the run time is $T = \Theta(n'/p^{2\epsilon}) = \Theta(n/p^{\frac{1}{2}+\epsilon})$. This analysis applies to step 1, 4 and 5.

One more concern is that in steps 4 and 5, each recursive call results in a problem size that grows by a constant factor of 2, and thus too many recursive calls will make the problem size too large. However, when $\frac{1}{6} < \epsilon < \frac{1}{2}$ is a constant the recursive depth is at most $\mathcal{O}(\frac{1}{1/2-\epsilon})$ and thus the resulting overhead is at most another constant. Thus the algorithm finishes in $\Theta(n/p^{\frac{1}{2}+\epsilon})$ time. □

8.3 Minimum Spanning Tree

We first show the result for edge inputs on a standard mesh and on an MCC. These algorithms are essentially the same as their fine-grained counterparts, and the proofs are omitted.

Theorem 8.3.1. *On a mesh-connected computer of size p with $1 \leq p \leq n$, for an undirected graph $G = (V, E)$ given as unsorted edge data with n edges, where each processor has n/p of them, a minimum spanning tree (or forest) can be found in $\Theta(n/p^{\frac{1}{2}} + n/p \cdot \log n)$ time.*

Theorem 8.3.2. *On an ER MCC$(n, p, p^{\frac{1}{2}+\epsilon})$ with $1 \leq p \leq n$ and $0 \leq \epsilon < \frac{1}{2}$, for an undirected graph $G = (V, E)$ given as unsorted edge data with n edges, where each processor has n/p of them, a minimum spanning tree (or forest) can be found in $\Theta(n/p^{\frac{1}{2}+\epsilon} \cdot \log n)$ time.*

The MCC algorithm has an extra logarithmic factor. We do not know if the logarithmic factor can be completely removed or not. However, without removing the logarithmic factor, it can still help for solving the problem with matrix input. Theorem 8.3.4 gives the result, and it requires the standard mesh version in Theorem 8.3.3.

Theorem 8.3.3. *On a mesh-connected computer of size p with $1 \leq p \leq n$, given the adjacency matrix of an undirected graph of $n^{\frac{1}{2}}$ vertices, where each processor has a $\sqrt{n/p} \times \sqrt{n/p}$ block of the matrix, a minimum spanning tree (or forest) can be found in $\Theta(n/p + p^{\frac{1}{2}})$ time.*

Proof. Our algorithm is similar to the fine-grained one. First we solve the problem inside each processor using Prim's algorithm [66]. It takes $\Theta(n/p)$ time since the input is a matrix. Then we stepwise merge the partial results. There will be $\log p^{\frac{1}{2}}$ levels, and at each level, we have two times to consider: data movement time and subproblem solving time. At the ith level ($1 \leq i \leq \log p^{\frac{1}{2}}$), the number of processors involved in a subproblem is 4^i, and the number of vertices is at most $2 \cdot 2^i$. Thus the size of data that needs to be moved and merged is $n' = \mathcal{O}(2^i\sqrt{n/p})$. Therefore

- the time for data movement is $\mathcal{O}(2^i\sqrt{n/p}/2^i + \sqrt{p}) = \mathcal{O}(\sqrt{n/p} + \sqrt{p})$; and
- the time for solving the problem is $\mathcal{O}(\sqrt{n/p} + n^{\frac{1}{4}})$.

Here is a more detailed analysis. For both moving and solving, the size of the problem is $2^i\sqrt{n/p}$, and the number of processors is 4^i. If we evenly split the data into processors, and solve the problem in the sub-mesh, it takes $\mathcal{O}(\sqrt{n/p} + \sqrt{n/p}/2^i \cdot \log(\sqrt{n/p}/2^i)) = \mathcal{O}(\sqrt{n/p})$ time. The $\mathcal{O}(\sqrt{p})$ in the moving time comes from the fact that it is possible that $4^i > 2^i\sqrt{n/p}$. For example, this is true when $p > \sqrt{n}$, since $p > \sqrt{p} \times \sqrt{n/p}$ in the last step. In this case, we are not going to use all the processors, and the problem is solved in n' processors instead. The moving time becomes just the distance, which is $\Theta(\sqrt{p})$. We know that the run time for solving in this case is $\Theta(\sqrt{n'})$, and the run time increases geometrically

so that the last step dominates. Thus the total time is $\Theta(\sqrt{\sqrt{p} \times \sqrt{n/p}}) = \Theta(n^{\frac{1}{4}})$. Note that $n^{\frac{1}{4}} = \mathcal{O}(\sqrt{p})$ when $p > \sqrt{n}$.

Thus the data movement will cost $\Theta(\sqrt{n/p}\log p + \sqrt{p})$ time in total, and solving the partial problem takes $\Theta(\sqrt{n/p}\log p + n^{\frac{1}{4}})$ at most. Thus the total run time is $\Theta(n/p + \sqrt{n/p}\log p + \sqrt{p} + n^{\frac{1}{4}})$. Notice that $\sqrt{n/p}\log p$ is always smaller than n/p when $p \leq n^{\frac{3}{4}}$, though this is not a tight bound. When $p > n^{\frac{3}{4}}$, $\sqrt{n/p}\log p < \sqrt{p}$. Thus this term never dominates, and can be ignored. Same thing happens for $n^{\frac{1}{4}}$. Thus the simplified run time is just $\Theta(n/p + p^{\frac{1}{2}})$. □

Note that the algorithm is optimal, since one cannot do better than $\Theta(n/p + p^{\frac{1}{2}})$. One term is the serial bound for reading all data, and the other is the diameter bound on mesh.

We now solve the problem on an ER MCC$(n, p, p^{\frac{1}{2}})$.

Theorem 8.3.4. *On an ER MCC$(n, p, p^{\frac{1}{2}})$ with $1 \leq p \leq n$, given the adjacency matrix of an undirected graph of $n^{\frac{1}{2}}$ vertices, where each processor has a $\sqrt{n/p} \times \sqrt{n/p}$ block of the matrix, a minimum spanning tree (or forest) can be found in $\Theta(n/p + n^{\frac{1}{4}})$ time.*

Proof. Our algorithm is similar to the fine-grained one. First divide the mesh into p/B blocks, each of size $\sqrt{B} \times \sqrt{B}$. Find a minimum spanning forest in each block using a standard mesh algorithm. This step takes $\Theta(n/p + \sqrt{B})$ time.

For each block, only the edge information is needed for future steps. The information is merged stepwise, and we only use channels to do data movements. At the ith level ($1 \leq i \leq \log(p/B)^{\frac{1}{2}}$), the size of data that needs to be moved and merged is $n' = 2^i\sqrt{B \cdot n/p}$. Therefore

- the time for data movement is $\Theta(\sqrt{B} \times \sqrt{n/p} \times (p/B)/p^{\frac{1}{2}} + \sqrt{n/p})$, since the time for the first step will always dominate, and the $\sqrt{n/p}$ term deals with the case that each processor may need to read from or write to the channels more than once;
- the time for solving is still $\Theta(\sqrt{n/p} + n^{\frac{1}{4}})$, since we are still using the standard mesh algorithm.

Note that if we use an MCC algorithm to solve, the time it takes is $\Theta(n'/(4^i \cdot B/p^{\frac{1}{2}}) \cdot \log n) = \Theta(\sqrt{n/B}/2^i \cdot \log n)$. This term never dominates. The number of channels is not big enough to support a faster algorithm.

The total run time is $\Theta(\frac{n}{p} + B^{\frac{1}{2}} + \sqrt{\frac{n}{B}} + \sqrt{\frac{n}{p}}\log p + n^{\frac{1}{4}})$. Let $B = n^{\frac{1}{2}}$, we get time $\Theta(n/p + \sqrt{\frac{n}{p}}\log p + n^{\frac{1}{4}})$. Similar as before, $\sqrt{\frac{n}{p}}\log p$ never dominates, and finally we get run time $\Theta(n/p + n^{\frac{1}{4}})$. Note that $B = n^{\frac{1}{2}}$ requires $p \geq n^{\frac{1}{2}}$. When $p < n^{\frac{1}{2}}$, n/p is at least $n^{\frac{1}{2}}$, thus is bigger than $B^{\frac{1}{2}}$, $\sqrt{n/B}$ and $n^{\frac{1}{4}}$. Thus the run time is as claimed. □

This algorithm is optimal, since n/p is the serial bound for reading all data, and $n^{\frac{1}{4}}$ is the run time lower bound for $p = n$. One can use the same idea as in Section 3.4 to check this bound for other p's.

Things become more interesting when we allow concurrent read. There are n/p items in each processor, and we need to be able to access and modify each item quickly enough; otherwise the power of concurrent read may be useless. Fortunately, this can be done perfectly by a hash table in serial.

We now solve the problem for finding an arbitrary spanning tree using a coarse-grained MCC with concurrent read. It requires a k-label selection lemma, similar to Lemma 6.3.1.

Lemma 8.3.5. *On a CR MCC$(n, p, 1)$ with $1 \leq p \leq n$, given $1 \leq k \leq n$, the k-label selection problem can be solved in $\Theta(n^{\frac{2}{3}}/p^{\frac{1}{3}} + n/p \cdot \log n + k)$ time when $p \geq n^{\frac{1}{2}}$, and $\Theta(n/p + p + k)$ time otherwise.*

Proof. The key difference between the two cases is that whether we need a base sort. If the number of processors is bigger than roughly $n^{\frac{1}{2}}$, sort them is better. Otherwise, sort is just a waste of time. Note that when p is greater than roughly $n^{\frac{1}{2}} \log^{\frac{3}{2}} n$, the time for later steps will dominate and shadow the term for sort in a single processor, and the time becomes $\Theta(n^{\frac{2}{3}}/p^{\frac{1}{3}} + k)$. We can save some fractions of the logarithmic factor when p is very close to $n^{\frac{1}{2}}$, but these modifications are tiny and subtle, and are ignored here.

In the first case, divide the mesh into p/B blocks of size $B^{\frac{1}{2}} \times B^{\frac{1}{2}}$, with $B = p^{\frac{4}{3}}/n^{\frac{2}{3}}$. This can happen only if $p \geq n^{\frac{1}{2}}$, which is true in this case. In each block, sort the labels and eliminate duplicates, then compress the remaining labels, and save them in a hash table in each processor. Using the same method as in Lemma 6.3.1, broadcast k values, or all values if there is not this many. Note that the only change is that in the two moves after each broadcast, processor p_i that has a null value asks a new value from its later processor p_{i+1}, and processor p_{i+1} can send an arbitrary valid one from the hash table. After receiving the value, processor p_i saves the value to the hash table, and this move is finished. The run time for this case is $\Theta((n/p \cdot B)/\sqrt{B} + n/p \cdot \log n + p/B + k) = \Theta(n^{\frac{2}{3}}/p^{\frac{1}{3}} + n/p \cdot \log n + k)$.

The second case is easy. Each processor use a hash table to save values, and broadcast one at a time. In the worst case, we need to loop through all processors, and the time is $\Theta(n/p + p + k)$. □

Theorem 8.3.6. *On a CR MCC$(n, p, p^{\frac{1}{2}})$ with $1 \leq p \leq n$, given the adjacency matrix of an undirected graph G of $n^{\frac{1}{2}}$ vertices stored in block format, a spanning tree, or a spanning*

forest, can be found in time

$$\begin{cases} \Theta(n/p) & \textit{when } 1 \le p < n^{\frac{1}{2}}, \\ \tilde{\Theta}(n^{\frac{3}{4}}/p^{\frac{1}{2}}) & \textit{when } n^{\frac{1}{2}} \le p < n^{\frac{3}{4}}, \\ \tilde{\Theta}(n/p^{\frac{5}{6}}) & \textit{when } n^{\frac{3}{4}} \le p \le n. \end{cases}$$

Proof. The algorithm contains three cases.

The first case is when $p < n^{\frac{1}{2}}$. In this case, all edges from a vertex is saved in the same processor. Thus, the label selection is easy, and takes $\mathcal{O}(n/p)$ time for all vertices. We choose $n^{\frac{1}{8}}$ edges from each vertex. Find a spanning tree using the whole mesh. The number of edges is $n^{\frac{5}{8}}$, and the mesh size is p. Using Theorem 8.3.1, a spanning tree can be found in $\Theta(n^{\frac{5}{8}}/p^{\frac{1}{2}} + n^{\frac{5}{8}}/p \cdot \log n)$ time. Use the mesh to propagate labels back, and this is no more than a sort for $n^{\frac{1}{2}}$ values. Do the previous steps 4 times, and a spanning tree is found. The total run time is $\Theta(n/p)$.

The second case is when $p \ge n^{\frac{3}{4}}$. In this case, each vertex's block is saved in $p/n^{\frac{1}{2}}$ processors. Giving each $n^{\frac{1}{2}}/p^{\frac{1}{2}}$ vertex blocks a channel, do an $n^{\frac{1}{8}}$-label selection. This part takes $\Theta((n^{\frac{1}{2}})^{\frac{2}{3}}/(p/n^{\frac{1}{2}})^{\frac{1}{3}} + n/p \cdot \log n + n^{\frac{1}{8}}) = \Theta(n^{\frac{1}{2}}/p^{\frac{1}{3}} + n/p \cdot \log n)$ time for each vertex. Since every $n^{\frac{1}{2}}/p^{\frac{1}{2}}$ vertices share a channel, the total time is $\Theta(n/p^{\frac{5}{6}} + n^{\frac{3}{2}}/p^{\frac{3}{2}} \cdot \log n)$. Finding a spanning tree on mesh takes $\Theta(n^{\frac{5}{8}}/p^{\frac{1}{2}} + n^{\frac{5}{8}}/p \cdot \log n) = \mathcal{O}(n/p^{\frac{5}{6}})$ time. The total run time is $\Theta(n/p^{\frac{5}{6}} + n^{\frac{3}{2}}/p^{\frac{3}{2}} \cdot \log n)$. Note that when p is bigger than roughly $n^{\frac{3}{4}} \log^{\frac{3}{2}} n$, the $n^{\frac{3}{2}}/p^{\frac{3}{2}} \cdot \log n$ term will be shadowed, and the time becomes $\Theta(n/p^{\frac{5}{6}})$.

The third case is when $n^{\frac{1}{2}} \le p < n^{\frac{3}{4}}$. In this case, we use a mesh sort in each vertex block to select labels. Note that we can use Lemma 8.3.5 to select labels as well. However, since each $n^{\frac{1}{2}}/p^{\frac{1}{2}}$ vertices need to share a channel, this overhead is too big, and the concurrent read cannot help anymore. The standard mesh sort takes $\Theta(n^{\frac{1}{2}}/(p/n^{\frac{1}{2}})^{\frac{1}{2}} + n/p \cdot \log n) = \Theta(n^{\frac{3}{4}}/p^{\frac{1}{2}} + n/p \cdot \log n)$ time. After this, we still use a standard mesh algorithm to find a spanning tree, which takes $\Theta(n^{\frac{5}{8}}/p^{\frac{1}{2}} + n^{\frac{5}{8}}/p \cdot \log n)$ time. Thus the total run time for this case is $\Theta(n^{\frac{3}{4}}/p^{\frac{1}{2}} + n/p \cdot \log n)$. Similar to the previous case, when p is bigger than roughly $n^{\frac{1}{2}} \log^2 n$, the $n/p \cdot \log n$ term will be shadowed, and the time becomes $\Theta(n^{\frac{3}{4}}/p^{\frac{1}{2}})$.

Note that in all cases, we use a mesh algorithm to find a spanning tree. This is so because the number of channels here is just $p^{\frac{1}{2}}$. It needs to be at least $p^{\frac{1}{2}+\epsilon}$ in order to help more. What's more, although the run times are represented by $\tilde{\Theta}$, they do not contain any logarithmic factor when p is not very close to the boundary. In these cases, same as in Lemma 8.3.5, we can save some fractions of the logarithmic factor, but these modifications are ignored here. □

8.4 Convex Hull

Computational geometry problems are different from graph problems with matrix input in that most problems have a sort lower bound. In this case we have to use an MCC$(n, p, p^{\frac{1}{2}+\epsilon})$, instead of just an MCC$(n, p, p^{\frac{1}{2}})$, in order to get any improvement. Here we use the problem of finding a convex hull as an example. Again, the mesh algorithm does not yet exist. Though the MCC algorithm is very different from the mesh algorithm, and does not use it as a base case, we still solve the mesh problem in Theorem 8.4.1 because we believe that the algorithm itself is interesting enough. The MCC algorithm is in Theorem 8.4.2. Note that solving the serial problem has a $\Theta(n \log n)$ run time, thus the serial time will dominate when p is very small.

Theorem 8.4.1. *On a mesh-connected computer of size p with $1 \leq p \leq n$, given n or fewer planar points, n/p in each processor, the extreme points of their convex hull can be found in $\Theta(n/p \cdot \log n + p^{\frac{1}{2}})$ time.*

Proof. First find the convex hull, and sort the extreme points in each processor by their x-coordinates. This takes $\Theta(n/p \cdot \log n/p)$ time. We use the same algorithm as in the fine-grained model [19]. We stepwise merge the upper and lower hulls. In the fine-grained model, each time after the binary search, we move all points to a smaller mesh to decrease run time. Here we need to do slightly more. One step of the binary search takes $2^i + n/p$, if the current block is of size 4^i, and there are n/p points in each processor. We need to make both terms decrease geometrically as the algorithm moves on. This can be done as follows. In odd steps, move all values into a submesh, without changing the number of values in each processor; whereas in even steps, spread out all values so that the number of values in each processor decreases by a factor of $1/2$. In this way, both terms decrease geometrically as the binary search goes on, and the total run time for one binary search is just $\Theta(2^i + n/p)$, if the current block size is 4^i. This gives that the final run time for all binary searches is $\Theta(p^{\frac{1}{2}} + n/p \cdot \log p)$. Therefore the total run time is as claimed. □

This algorithm is optimal, unless one can solve the serial convex hull problem faster than $\Theta(n \log n)$. This can be done in some expected cases, or there are other constrains on the points given. These cases will not be discussed here.

Theorem 8.4.2. *On an ER MCC$(n, p, p^{\frac{1}{2}+\epsilon})$ with $1 \leq p \leq n$ and $0 < \epsilon < \frac{1}{2}$, given n or fewer planar points, n/p in each processor, the extreme points of their convex hull can be found in $\Theta(n/p + p^{(\frac{1}{2}-\epsilon)/3} \cdot \log^2 n)$ time if the points are pre-sorted, and $\Theta(n/p^{\frac{1}{2}+\epsilon} + n/p \cdot \log n)$ time otherwise.*

Proof. First note that if the input is pre-sorted by their x-coordinates, finding a convex hull takes $\Theta(n)$ time, instead of $\Theta(n \log n)$. The algorithm uses the same idea as in the fine-grained algorithm. For example, for the upper hull, repeatedly join consecutive pairs of segments, finding the upper support line joining them. This can be accomplished via binary searches, where each step involves broadcasting a line segment of the partial hull on the left, and a reduction is used on the remaining points on the right segment determine if any of them lie above the extension of this segment. Once the support lines have been determined, some formerly active points from each side may now become inactive. Broadcasting the index and coordinates of the rightmost point remaining on the left side, and leftmost on the right side, allows every remaining point to adjust its index and neighbor information.

This process takes $\Theta(p^{(\frac{1}{2}-\epsilon)/3})$ for broadcasting a value to p processors. However, here we need to do more. Since each processor has n/p values, simply comparing all these values with the broadcast value takes $\Theta(n/p)$ time. The time can be reduced by a binary search in each processor: each processor keeps an array, and the left and right boundaries of it. In this way, each time the processor receives a broadcast value, it can determine and update the boundary in $\mathcal{O}(\log n)$ time.

Since we need to do binary search $\lfloor \log p \rfloor$ times, the total time is $\Theta(n/p + p^{(\frac{1}{2}-\epsilon)/3} \cdot \log^2 n + \log^3 n)$ if the input is pre-sorted, and the $\log^3 n$ term can be eliminated since it never dominates. Otherwise, we need to add the time for sort, and the time becomes $\Theta(n/p^{\frac{1}{2}+\epsilon} + n/p \cdot \log n)$. □

CHAPTER 9

Conclusion

Our model was first developed in 2015 [93], and lots of gaps need to be filled. The model is highly realistic and scalable, and similar real computers have been built. One goal for doing research on this model is that it can well separate local and global communications that are required in a problem, and help us understand whether a problem is essentially more "local" or "global". The goal is trying to avoid global communications as much as possible, so that the algorithms can be better parallelized.

We have shown that extending mesh-connected computers with global communication channels gives a significant improvement for a range of problems, including sort, FFT, graph problems, geometry problems, etc. The time is reduced from $\Theta(n^{\frac{1}{2}})$ to $\tilde{\Theta}(n^{\frac{1}{2}-\epsilon})$ on an ER MCC$(n, n^{\frac{1}{2}+\epsilon})$ when sort serves as a lower bound, and to $\tilde{\Theta}(n^{\frac{1}{3}(1-\epsilon)})$ or $\tilde{\Theta}(n^{\frac{1}{2}(1-\epsilon)})$ otherwise on an ER MCC(n, n^{ϵ}), depending on how much data needs to be moved globally. Some algorithms have an extra logarithmic factor, and it is an open question if it can be removed. Most algorithms are optimal, with time that matches the lower bound given by the bisection bandwidth of the system.

Two-dimensional mesh computers have been built for many years [7, 8, 9], and projections are that many more will be built in the future. There are now meshes with 1000 cores [13, 14] and most plans for processors on exascale computers assume the chips will have meshes with myriad cores [10, 11, 12]. A DARPA-funded project is developing a chip with approximately 4000 simple cores to be used for image processing, where here too they are connected as a mesh [94].

These projected designs include additional communication to overcome the $\Omega(n^{\frac{1}{2}})$ diameter and bandwidth constraints of standard meshes [10, 11, 12], but algorithms haven't been developed to fully utilize this communication. Some algorithms have been developed for meshes augmented by buses of various forms [24, 25, 26, 27, 28, 29, 30, 31], but buses are concurrent read channels, a construct that is difficult to scale. Our model only requires exclusive read and exclusive write in most algorithms. Further, none of these systems

can improve basic data movement-intensive operations such as sorting or matrix transpose since they do not increase the bisection bandwidth.

We don't specify an implementation, but show that the functionality is useful. There are several ways that one might achieve this functionality, and probably new ways will be discovered. Whatever the implementation, previous results [93, 95] and the results here show that adding flexible global communication is a possibility that should be explored.

9.1 Future Work

There are numerous problems that can be solved faster on the MCC than on a standard mesh without additional communication channels. For many problems sorting is the bottleneck, so Theorem 4.1.2 shows that MCCs with global communication $\Omega(n^{\frac{1}{2}+\epsilon})$ should offer a speedup. For example, this should be true for finding an Eulerian tour and determining the position of an edge in the ordering since in the worst case, a long chain, the bottleneck is sorting.

One can always try harder problems on a new model. One example is to solve NP-complete problems in random cases, e.g., constructing a Hamiltonian cycle on a random graph. Much work has been done on this problem [96, 97, 98, 99]. Gurevich and Shelah [97] show that given a random graph, a Hamiltonian cycle can be constructed in $\mathcal{O}(n)$ expected time in serial, and MacKenzie and Stout [99] show that it can be constructed in $\mathcal{O}(\log^* n)$ expected time on a CRCW PRAM in parallel. Solving this on our model is very challenging, and this may give us intuition on how to solve other NP-complete problems in parallel given random input.

The major concern for digging into a hard problem is that it is difficult to see why we want to solve such problem using our model. Solving these kind of problems may serve as a base for other even harder problems. However, they may not give much information about how to solve other problems. Therefore, it is difficult to determine which hard problems are worth solving on our model.

There are many other problems that do not require moving many things globally. Just like in the NP-complete problems case, randomized algorithms that solve many other easier problems in the expected case are very likely to have such property since random sampling may significantly reduce the amount of data that needs to be moved [100, 101], where by "easier" we mean in the sense of computational complexity. One should be able to solve these problems efficiently on our model.

Other than the problems, there are many other aspects of the model that needs to be explored. One of the most important aspects are energy and peak power consumption by

processors. They are becoming increasingly important in parallel computing. The DOE technical report [102] already claimed that "the primary design constraint for future High Performance Computing (HPC) systems will be power consumption". Many researches have already looked into such constrains, and developed algorithms that have a time-power trade-off and relatively balanced peak power consumption by processors [44, 42].

Though we did not explicitly analyze power consumption in our algorithms, one can expect that the peak power consumption by a single processor should be quite small if one choose the sub-mesh in the recursive calls very carefully in most of our algorithms. Despite this fact, energy and power analysis and time-power trade-off algorithms are still very interesting, and these possibilities should be explored in future.

BIBLIOGRAPHY

[1] "Bridges user guide," https://portal.xsede.org/psc-bridges, accessed: 2018-10-30.

[2] M. Ajtai, J. Komlós, and E. Szemerédi, "An $\mathcal{O}(n \log n)$ sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM, 1983, pp. 1–9.

[3] F. Y. Chin, J. Lam, and I.-N. Chen, "Efficient parallel algorithms for some graph problems," *Communications of the ACM*, vol. 25, no. 9, pp. 659–665, 1982.

[4] D. R. Karger, N. Nisan, and M. Parnas, "Fast connected components algorithms for the erew pram," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992, pp. 373–381.

[5] K. W. Chong, Y. Han, Y. Igarashi, and T. W. Lam, "Improving the efficiency of parallel minimum spanning tree algorithms," *Discrete Applied Mathematics*, vol. 126, no. 1, pp. 33–54, 2003.

[6] J. Seiferas, "Sorting networks of logarithmic depth, further simplified," *Algorithmica*, vol. 53, no. 3, pp. 374–384, 2009.

[7] K. Batcher, "Retrospective: architecture of a massively parallel processor," in *25 years of the International Symposia on Computer Architecture (selected papers)*. ACM, 1998, pp. 15–16.

[8] J. R. Nickolls, "The design of the Maspar MP-1: A cost effective massively parallel computer," in *Proc. Compcon*, 1990, pp. 25–28.

[9] S. F. Reddaway, "DAP—a distributed array processor," in *ACM SIGARCH Computer Architecture News*, vol. 2, no. 4. ACM, 1973, pp. 61–65.

[10] R. Stevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols *et al.*, "Scientific grand challenges: Architectures and technology for extreme scale computing," 2009.

[11] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "ATAC: a 1000-core cache-coherent processor with on-chip optical network," in *Proceedings 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 477–488.

[12] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2010, pp. 1–25.

[13] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip," *arXiv:1610.01832*, 2016.

[14] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array," in *VLSI Circuits, 2016 IEEE Symposium on*. IEEE, 2016, pp. 1–2.

[15] R. Computing, "The NEO chip," Tech. Rep., 2015.

[16] M. J. Atallah and S. E. Hambrusch, "Solving tree problems on a mesh-connected processor array," in *Foundations of Computer Science, 26th Annual Symposium on*. IEEE, 1985, pp. 222–231.

[17] M. J. Atallah and S. R. Kosaraju, "Graph problems on a mesh-connected processor array," *Journal of the ACM (JACM)*, vol. 31, no. 3, pp. 649–667, 1984.

[18] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays· Trees· Hypercubes*. Elsevier, 1992.

[19] R. Miller and Q. F. Stout, *Parallel algorithms for regular architectures: meshes and pyramids*. Mit Press, 1996.

[20] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Communications of the ACM*, vol. 20, no. 4, pp. 263–271, 1977.

[21] K. W. Chong, Y. Han, and T. W. Lam, "Concurrent threads and optimal parallel minimum spanning trees algorithm," *Journal of the ACM (JACM)*, vol. 48, no. 2, pp. 297–323, 2001.

[22] R. Miller and Q. F. Stout, "Efficient parallel convex hull algorithms," *IEEE transactions on Computers*, vol. 37, no. 12, pp. 1605–1618, 1988.

[23] N. M. Amato, M. T. Goodrich, and E. A. Ramos, "Parallel algorithms for higher-dimensional convex hulls," in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. IEEE, 1994, pp. 683–694.

[24] D. Bhagavathi, P. J. Looges, S. Olariu, J. L. Schwing, and J. Zhang, "A fast selection algorithm for meshes with multiple broadcasting," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 7, pp. 772–778, 1994.

[25] V. K. Prasanna-Kumar and C. S. Raghavendra, "Array processor with multiple broadcasting," *Journal of Parallel and Distributed Computing*, vol. 4, no. 2, pp. 173–190, 1987.

[26] V. K. Prasanna-Kumar and D. Reisis, "Image computations on meshes with multiple broadcast," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 11, pp. 1194–1202, 1989.

[27] Q. F. Stout, "Meshes with multiple buses," in *Foundations of Computer Science, 27th Annual Symposium on*. IEEE, 1986, pp. 264–273.

[28] M. J. Serrano and B. Parhami, "Optimal architectures and algorithms for mesh-connected parallel computers with separable row/column buses," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 10, pp. 1073–1080, 1993.

[29] S. H. Bokhari, "Finding maximum on an array processor with a global bus," *Computers, IEEE Transactions on*, vol. 100, no. 2, pp. 133–139, 1984.

[30] Q. F. Stout, "Broadcasting in mesh-connected computers," in *Proceedings 1982 Conference on Information Sciences and Systems*, pp. 85–90.

[31] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, "Parallel computations on reconfigurable meshes," *Computers, IEEE Transactions on*, vol. 42, no. 6, pp. 678–692, 1993.

[32] J. Jang and V. K. Prasanna-Kumar, "An optimal sorting algorithm on reconfigurable mesh," *Journal of Parallel and Distributed Computing*, vol. 25, no. 1, pp. 31–41, 1995.

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

[34] J. Jenq and S. Sahni, "Reconfigurable mesh algorithms for image shrinking, expanding, clustering, and template matching," in *Parallel Processing Symposium, 1991. Proceedings., Fifth International*. IEEE, 1991, pp. 208–215.

[35] K. Preston, *Multicomputers and Image Processing: Algorithms and Programs*. Elsevier, 1982.

[36] R. Miller and Q. F. Stout, "Pyramid computer algorithms for determining geometric properties of images," in *Proceedings of the first annual symposium on Computational geometry*. ACM, 1985, pp. 263–271.

[37] Q. F. Stout, "Drawing straight lines with a pyramid cellular automaton," *Information Processing Letters*, vol. 15, no. 5, pp. 233–237, 1982.

[38] A. Rosenfeld, *Multiresolution image processing and analysis*. Springer Science & Business Media, 2013, vol. 12.

[39] P. J. Burt and G. S. van der Wal, "Iconic image analysis with the pyramid vision machine (pvm)," in *IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, 1987, pp. 137–144.

[40] G. Fritsch, W. Kleinoeder, C. Linster, and J. Volkert, "Emsy85-the erlangen multiprocessor system for a broad spectrum of applications," 1983.

[41] R. Miller and Q. F. Stout, "Data movement techniques for the pyramid computer," *SIAM Journal on Computing*, vol. 16, no. 1, pp. 38–60, 1987.

[42] P. Poon and Q. F. Stout, "Time-power tradeoffs for sorting on a mesh-connected computer with optical connections," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 611–619.

[43] ——, "An optimal time-power tradeoff for sorting on a mesh-connected computer with on-chip optics," *International Journal of Networking and Computing*, vol. 4, no. 1, pp. 70–87, 2014.

[44] Q. F. Stout, "Algorithms minimizing peak energy on meshconnected systems," in *Proc. 18th ACM Symp. Parallelism in Algorithms and Architectures (SPAA), Cambridge, MA, USA*. Citeseer, 2006, pp. 331–334.

[45] D. Nassimi and S. Sahni, "Data broadcasting in simd computers," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 101–107, 1981.

[46] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.

[47] C. R. Cook and D. J. Kim, "Best sorting algorithm for nearly sorted lists," *Communications of the ACM*, vol. 23, no. 11, pp. 620–624, 1980.

[48] M. J. Atallah and S. E. Hambrusch, "Solving tree problems on a mesh-connected processor array," *Information and Control*, vol. 69, no. 1-3, pp. 168–187, 1986.

[49] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[50] A. Bar-Noy and D. Peleg, "Square meshes are not always optimal," in *Proceedings First annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 138–147.

[51] J. Fourier, *Theorie analytique de la chaleur, par M. Fourier*. Chez Firmin Didot, père et fils, 1822.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[53] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[54] B. A. Cipra, "The best of the 20th century: Editors name top 10 algorithms," *SIAM news*, vol. 33, no. 4, pp. 1–2, 2000.

[55] L. R. Rabiner, R. W. Schafer, and C. M. Rader, "The chirp z-transform algorithm and its application," *Bell System Technical Journal*, vol. 48, no. 5, pp. 1249–1292, 1969.

[56] M. J. Fischer and M. S. Paterson, "String-matching and other products," Massachusetts Institute of Technology Cambridge Project MAC, Tech. Rep., 1974.

[57] J. Aoe, *Computer algorithms: string pattern matching strategies*. John Wiley & Sons, 1994, vol. 55.

[58] S. Kim and Y. Kim, "A fast multiple string-pattern matching algorithm," in *Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999, pp. 44–49.

[59] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[60] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[61] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[62] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas, "The smallest automation recognizing the subwords of a text," *Theoretical computer science*, vol. 40, pp. 31–55, 1985.

[63] U. Vishkin, "Optimal parallel pattern matching in strings," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1985, pp. 497–508.

[64] G. M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of algorithms*, vol. 10, no. 2, pp. 157–169, 1989.

[65] P. Clifford and R. Clifford, "Simple deterministic wildcard matching," *Information Processing Letters*, vol. 101, no. 2, pp. 53–54, 2007.

[66] R. C. Prim, "Shortest connection networks and some generalizations," *Bell system technical journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[67] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[68] Q. F. Stout, "Optimal component labeling algorithms for mesh-connected computers and vlsi," *arXiv preprint arXiv:1502.01435*, 2015.

[69] S. Pettie and V. Ramachandran, "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1879–1895, 2002.

[70] R. E. Tarjan, "A note on finding the bridges of a graph," *Inf. Process. Lett.*, vol. 2, pp. 160–161, 1974.

[71] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[72] Q. F. Stout, "Tree-based graph algorithms for some parallel computers." in *ICPP*, 1985, pp. 727–730.

[73] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[74] ——, "Finding biconnected componemts and computing tree functions in logarithmic parallel time," 1984.

[75] D. B. Johnson and P. Metaxas, "A parallel algorithm for computing minimum spanning trees," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992, pp. 363–372.

[76] S. Bu-Qing and L. Ding-Yuan, *Computational geometry: curve and surface modeling*. Elsevier, 2014.

[77] S. L. Devadoss and J. O'Rourke, *Discrete and computational geometry*. Princeton University Press, 2011.

[78] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan, "Cg_hadoop: computational geometry in mapreduce," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2013, pp. 294–303.

[79] K. Bringmann and T. Friedrich, "Approximating the volume of unions and intersections of high-dimensional geometric objects," *Computational Geometry*, vol. 43, no. 6-7, pp. 601–610, 2010.

[80] D. T. Lee and F. P. Preparata, "Computational geometry? a survey," *IEEE Transactions on Computers*, no. 12, pp. 1072–1101, 1984.

[81] F. P. Preparata and M. I. Shamos, "Computational geometry: an introduction," in *Computational Geometry*. Springer, 1985, pp. 1–35.

[82] R. Cole and M. T. Goodrich, "Optimal parallel algorithms for polygon and point-set problems," in *Proceedings of the fourth annual symposium on Computational geometry*. ACM, 1988, pp. 201–210.

[83] M. J. Atallah, H. Wagener, and D. Z. Chen, "An optimal parallel algorithm for the visibility of a simple polygon from a point," *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 515–532, 1991.

[84] M. T. Goodrich, "Parallel algorithms in geometry," 2004.

[85] M. I. Shamos and D. Hoey, "Closest-point problems," in *Foundations of Computer Science, 1975., 16th Annual Symposium on*. IEEE, 1975, pp. 151–162.

[86] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal expected-time algorithms for closest point problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 6, no. 4, pp. 563–580, 1980.

[87] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 162–170.

[88] M. O. Rabin, *Probabilistic algorithms, in Algorithms and Complexity: New Dwectmns and Recent Results*. J.F. Traub (Ed.), Academic Press, 1976.

[89] X. Shen, W. Liang, and Q. Hu, "On embedding between 2d meshes of the same size," *IEEE transactions on computers*, no. 8, pp. 880–889, 1997.

[90] C. Rüb, "Line-segment intersection reporting in parallel," *Algorithmica*, vol. 8, no. 1-6, pp. 119–144, 1992.

[91] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *SIAM Journal on Computing*, vol. 20, no. 4, pp. 737–755, 1991.

[92] M. T. Goodrich, S. B. Shauck, and S. Guha, "Parallel methods for visibility and shortest-path problems in simple polygons," *Algorithmica*, vol. 8, no. 1-6, pp. 461–486, 1992.

[93] Y. An and Q. F. Stout, "Optimal algorithms for graphs and images on a shared memory mesh," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 883–891.

[94] J. Bates, "Singular Computing," private communication.

[95] Y. An and Q. F. Stout, "Optimal algorithms for a mesh-connected computer with limited additional global bandwidth," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 937–946.

[96] B. Bollobas, T. I. Fenner, and A. M. Frieze, "An algorithm for finding hamilton paths and cycles in random graphs," *Combinatorica*, vol. 7, no. 4, pp. 327–341, 1987.

[97] Y. Gurevich and S. Shelah, "Expected computation time for hamiltonian path problem," *SIAM Journal on Computing*, vol. 16, no. 3, pp. 486–502, 1987.

[98] A. M. Frieze, "Parallel algorithms for finding hamilton cycles in random graphs," *Inf. Process. Lett.*, vol. 25, no. 2, pp. 111–117, 1987.

[99] P. D. MacKenzie and Q. F. Stout, "Optimal parallel construction of hamiltonian cycles and spanning trees in random graphs," in *Proceedings of the fifth annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1993, pp. 224–229.

[100] K. L. Clarkson and P. W. Shor, "Applications of random sampling in computational geometry, ii," *Discrete & Computational Geometry*, vol. 4, no. 5, pp. 387–421, 1989.

[101] D. R. Karger, P. N. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM (JACM)*, vol. 42, no. 2, pp. 321–328, 1995.

[102] R. Stevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols *et al.*, "Architectures and technology for extreme scale computing," in *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.