COMS 4995

ADVANCED ALGORITHMS

Modern Algorithm Design for Parallel Computing

Submitted By :
John Daciuk (jpd2184)
Rebecca Calinsky (rdc2164)
Pierre Tholoniat (pt2537)

May 12, 2020

Contents

Abstract

We give an overview of modern parallel algorithm design, with special attention to applications of Massive Parallel Computation. We highlight broad ideas by outlining how well known problems can be efficiently solved with MPC. In section 2, as a warm up, we present algorithms for sorting and maximal matching from [Gha18] and [Gha19]. In section 3 we give intuition for and analyze the Minimum Spanning Tree (MST) problem in Euclidean space from [And+14], and in section 4 we synthesize the progress in [IMS17] for the Weighted Interval Selection problem.

1 Introduction

1.1 Parallel Algorithm Design Today

Datasets are growing exponentially, with a rate far beyond that of the processing power or memory of any single machine, making parallel computing a necessity. Meanwhile, with the bandwidth of communication ever increasing and infrastructure maturing, parallel computing is now more practical than ever. As such, there has been a surge in interest in parallel computing over the past decade or so, and there is still much room for improved algorithm design to take advantage of recent hardware innovation.

1.2 PRAM

The Parallel Random Access Machines model, or PRAM, is considered the most widely used model for parallel computation today [Gha18]. In this model, p is the number of RAM processors used, where each has access to shared memory, and computations are broken down into individual RAM operations. This "fine-grained" view of parallelism assumes that operations occur at the same speed, and communication problems are ignored as well, often leading to algorithms that are not very practical for real-world scenarios. In order to mitigate this problem, a more "coarse-grained" model, such as the Massively Parallel Computation model discussed in Section 1.4, can be helpful.

1.3 MPC

In the Massively Parallel Computation framework, we assume communication to be the bottleneck. With the intention to leverage the power of a myriad of processors, it's a far fetched supposition that we could manage to give them all fast access to shared memory. Thus, in MPC we take each machine to have its own CPU and memory.

We say that the system as a whole is composed of M machines each with S words of memory[1]. We endeavor to handle massive inputs of size N, so we should not expect each machine to be able to have $\Theta(N)$ memory, rather we imagine S to be more like $\sqrt{N}$ or $N^{.9}$ depending on the task and resources. The input will hence be distributed across the machines, likely arbitrarily. Note that to even give the system the input, we'll need $M \geq N/S$. For example, if we have $S = N^{.9}$, we would need more than $N^{.1}$ machines, which could be on the order of hundreds of thousands if we're dealing with petabytes of input. MPC algorithms will typically assume that the system can hold something like twice the input, or some other small constant.

We think of MPC algorithms progressing in rounds. Naturally, each round consists of what each machine can do in parallel with all the others, which is to compute on its local slice of input, and then send and receive up to S words. Clearly a machine cannot even desire to receive more then S

[1]Where a word is perhaps 32 or 64 bits, depending on machine architecture.

words per round since such a message would exhaust its memory. On the other hand, although a machine may like to send out multiple copies of its memory in a single round, all machines cannot indulge in such a action in parallel because the system would not have the space to receive such a message. Consequently, the time it takes a machine to send and receive S words is the natural amount of parallel communication time to allocate to each round. Since this communication time across appreciable distance generally takes much longer than the compute time, we don't even explicitly account for compute time per round in runtime analysis, and instead concern our analysis entirely with the number of rounds needed. As such, the algorithms we design need only describe what information each machine should send and receive in each round; i.e. the time to compute this information is negligible.

1.4 BSP

The MPC and PRAM models are not the only frameworks used to formalize parallel algorithms. The Bulk-Synchronous-Parallel (BSP) model, introduced by Valiant in the 80s [Val90], can be seen as a wide model of which the MPC model is a special case. [IMS17], that we are going to cover later, emphasizes the simplicity of the MPC model, that has less parameters than the BSP model.

One of the main differences between BSP and MPC is synchronization. In the MPC model, it is taken for granted that processes can communicate in rounds and progress synchronously from one round to the next one. In other words, the MPC model requires synchronous communication.

In practice, modern networks do not offer strict guarantees on the time that a message might take to be delivered: communication might be asynchronous or partially synchronous. To account for that, the BSP model introduces a mechanism called *synchronization barrier*, such that each process reaching the barrier waits for the others. It allows us to divide the algorithm into a sequence of supersteps, where communication and computation happen asynchronously inside each superstep.

2 Warm Up MPC Algorithms

2.1 Propagating a Message

One common subroutine is to get data of size d from one master machine to all other machines. If $d > S/(M-1)$, this cannot be done in 1 round since the master machine cannot send more than S words per round. The naive approach would be to have the master machine send d to $\lfloor S/d \rfloor$ machines each round, taking roughly Md/S rounds in total. We can do much better than this with a broadcast tree. The master can send to $\lfloor S/d \rfloor$ machines the first round. In the second round, all of those $\lfloor S/d \rfloor + 1$ machines can send to $\lfloor S/d \rfloor$ machines each and so on. The number of machines that have the data after R rounds is lower bounded by the number of nodes in a tree with branching factor $\lfloor S/d \rfloor$. This tree is in turn lower bounded by its leaves, which are simply $\lfloor S/d \rfloor^R$ after R rounds. Since we want all the machines to receive the data, it is enough that $\lfloor S/d \rfloor^R \geq M$, or $R \geq \frac{\log M}{\log(S/d)}$.

2.2 Sorting

2.2.1 Algorithm Description

We now consider quick sort for MPC as presented in [Gha18]. We seek to correctly sort n items with high probability, such that each machine knows the rank of the items it started with. We take $S = n^\varepsilon$ for $\varepsilon > 0$ and $M \cdot S = 2n$, or $M = 2n^{1-\varepsilon}$ so that we have enough machines to hold twice the input. We show how to achieve $O(\frac{1}{\varepsilon^2})$ rounds.

The goal is to break the problem down into smaller and smaller subproblems, each some factor smaller than the last, until a subproblem fits into a single machine. Here a subproblem takes the form of an unsorted list of *all* the elements between two known indices in the original list.

We start by having each machine tag its elements with probability $\frac{n^{\varepsilon/2}}{2n}$. The expected number of the n total elements to be tagged is $n \cdot \frac{n^{\varepsilon/2}}{2n} = \frac{n^{\varepsilon/2}}{2}$, and so by Markov is likely less than $n^{\varepsilon/2}$. Importantly, this is likely to fit on the first machine, which we'll call the master. In fact, assuming that we have enough machines to hold twice the input, the master should have $\frac{1}{2}n^\varepsilon$ free space, easily accommodating the $n^{\varepsilon/2}$ tagged items. In 1 round we can send all tagged items to the master; these will then be sorted by the master and act as pivots or boundary markers for subproblems.

Next, the master will send these sorted pivots to all other machines with the broadcasting technique discussed in section 2.1. Applying the general formula we derived there, we can bound the required rounds r for the broadcast

$$r \leq \frac{\log M}{\log(S/d)} = \frac{\log(2n^{1-\varepsilon})}{\log(n^{\varepsilon/2})} = O\left(\frac{1-\varepsilon}{\varepsilon/2}\right) = O\left(\frac{1}{\varepsilon}\right) \tag{1}$$

Since each machine now has the sorted pivots, each machine can determine between which two pivots each of its input items belongs to. Next, each machine will send counts to the master such that the master knows how many items fall between each consecutive pair of pivots. The master will then assign sets of machines to the items between each two consecutive pivots to roughly equally distribute the work. In another $O(\frac{1}{\varepsilon})$ rounds of broadcast communication, the master can give each machine the total assignment scheme.

Finally, each machine will randomly distribute its input items across the machines that were assigned to them. Because we have enough space to hold twice the input in the system, with good probability no machine will be assigned beyond its capacity. If a machine is overburdened, we'll have to try again. Otherwise, to more systematically determine a distribution scheme, we'd need more rounds of communication, which is precisely what we're trying to avoid. Throughout the redistribution of items, note that each item remains tagged with its initial machine.

As stated, we will repeat this process, recursively reducing the subproblems until the subproblems are down to size n^ε and can be sorted on one machine. As a last step, these single machines can then message back the original machines that held each item their orders.

2.2.2 Analysis

Let's again consider this first step in the recursion. Ultimately we'd like to know by what factor smaller are the subproblems than the original, where the subproblems are the items between each two consecutive pivots. The smaller the subproblems become in each step, the quicker they will be small enough to fit in a single machine.

Let's consider the number of elements between some two consecutive pivots, which we'll refer to as *width*. The *width* can't be too large, because by construction not a single one of these items happened to be marked. Naturally we might expect the *width* to be something like the reciprocal of the probability of items being marked, or in this case $2n^{1-\varepsilon/2}$, which is already suggestive that our subproblems are smaller by something like a $n^{\varepsilon/2}$ factor. However, since we may recurse many times, we'd like an extremely high probability guarantee. To be more precise, since none of these items within the width were marked, we have

$$Pr(width \geq x) \leq (1 - Pr(marked))^x = \Big(1 - \frac{n^{\varepsilon/2}}{2n}\Big)^x \leq exp\Big(\frac{-xn^{\varepsilon/2}}{2n}\Big) \tag{2}$$

where if $x = cn^{1-\varepsilon/2}\ln n$, we'd have $Pr(width \geq x) \leq n^{2/c}$. Thus, if $c = 30$, this probability becomes exceedingly small, and we can say with very high probability that the width, or number of items between two consecutive pivots, is less than $cn^{1-\varepsilon/2}\ln n$. This, in turn, is much less than $n^{1-\varepsilon/3}$ for the range of n we care about.

Consequently, in the first level of recursion, the size of our subproblems are likely no more than $n^{1-\varepsilon/3}$, and in the l^{th} level they are likely no more than $n^{1-l\varepsilon/3}$. After $l = O(\frac{1}{\varepsilon})$ levels of recursion, we can make the subproblems as small as we'd like.

In the final runtime analysis, recall that since we also need O $(\frac{1}{\varepsilon})$ rounds per level of recursion, our total runtime would be O $(\frac{1}{\varepsilon^2})$. Recalling that we assumed each machine to have n^{ε} space, we see that as we attempt to shrink ε and spread the input across more and more machines, our runtime pays a $1/\varepsilon$ factor twice. Once because of the time it takes the master machine to spread the message about the pivots, and another because we need that many more recursive reductions to fit the subproblems into individual machines.

2.3 Maximal Matching

We present an algorithm for maximal matching as described in [Gha19]. Suppose we have a graph $G = (V, E)$ with n vertices and m edges. Further, suppose the graph has far more edges than vertices, such that each machine in the MPC model can hold all the vertices but not all the edges. A *matching* is a set of edges such that no two edges are incident on the same vertex, and a *maximal matching* is a *matching* such that no addition edge can be added to the set. Note that a maximal matching is not a globally optimal matching, although it can be shown that the two differ at most by a factor of 2. The analysis and techniques that have been proposed for this problem are sensitive to the size of s, or the memory of each machine. Here we'll take the superlinear memory regime and assume that $s = n^{1+\varepsilon}$ for $\varepsilon \in (0, 1]$ and show how to achieve $O\frac{\log m}{\varepsilon \log n - \log 2}$ rounds.

To start edges are randomly distributed across machines and we desire that in the end each machine know which of its initial edges belong in the maximal matching set. In each round we add edges to our matching set M, starting with the empty set by doing what follows:

1. Mark each of the remaining m edges with probability $p = \frac{n^{1+\varepsilon}}{2m}$, and move marked edges to a master machine.
2. Have the master machine compute the maximal matching for the marked edges. Add that matching to M.
3. Communicate to all other machines the edges added to M so that all machines can drop matched vertices and other edges incident on those vertices.

The algorithm will terminate when machines have no edges left to mark. At this point there will be no more available matches to make and we'll have a maximal matching. The correctness lies in the fact that edges only disappear as options when they are no longer viable because either they have been used or vertices they connect to have been matched. To show that with high probability we will be done in rounds roughly proportional to $1/\varepsilon$, we show that with high probability the number of remaining edges drops by a factor of $n^\varepsilon/2$ each round.

Lemma 6.17 from [Gha19] *After each iteration starting with m edges, with probability $1 - exp(-\Theta(n))$, the number of the remaining edges is at most $\frac{2m}{n^\varepsilon}$.*

Proof. The key is to realize that any large subset S of the vertices in G is so likely to have an edge marked that it's likely that *all* large vertex subsets will have an edge marked. Therefore, after some rounds our remaining vertex subset probably won't be large because we know that our remaining subset does not have an edge in it that was marked; if it did at least one of the vertices the edge connects would have been removed. More precisely, call a *heavy* vertex set one with at least $\frac{2m}{n^\varepsilon}$ edges. The probability that no edge was marked in a round is

$$(1 - pr(\text{marked}))^{\frac{2m}{n^\varepsilon}} = (1 - \frac{n^{1+\varepsilon}}{2m})^{\frac{2m}{n^\varepsilon}} \leq e^{-n} \tag{3}$$

Since there are no more than 2^n possible choices for a *heavy* S, with probability at most $e^{-n} \cdot 2^n$, at least 1 *heavy* subset of vertices will have had *no* marked edges. But this directly implies that with probability $1 - e^{-n} \cdot 2^n \geq 1 - exp(-\Theta(n))$ no *heavy* subset would have had no marked edges, or all *heavy* subsets will have marked edges. Therefore with this probability our remaining subset of vertices is not a heavy subset, again, because if it did have a marked edge, we could have further removed those additional two vertices the edge is incident on. Because the subset of vertices remaining at the end of a round is not *heavy*, it has at most $\frac{2m}{n^\varepsilon}$ edges. With this, we've shown that in each round the remaining edges are reduced by at least of a factor of $\frac{n^\varepsilon}{2}$. □

Since we reduce the edges by a factor of at least $\frac{n^\varepsilon}{2}$ each round, we need the following to get down to 0 available edges

$$m \cdot \left(\frac{2}{n^\varepsilon}\right)^r < 1 \tag{4}$$

$$r \log \frac{2}{n^\varepsilon} < \log \frac{1}{m} \tag{5}$$

$$r \log \frac{n^\varepsilon}{2} > \log m \tag{6}$$

$$r = \frac{\log m}{\varepsilon \log n - \log 2} \qquad \text{sufficient r} \tag{7}$$

where r is the rounds required.

Remarks

Note that we see this sort of probabilistic reasoning applied quite frequently in the MPC paradigm. Since machine communication is the bottleneck, it will often be advantageous to spread out work among machines by having machines take on random chunks of work, hoping that the job somewhat uniformly distributes.

One key to all of this working is that we do suppose machines to have large enough memory to carry the entire vertex set, just not the edge set. This is somewhat reasonable as edge sets can be exponentially larger than vertex sets. In this way machines were able to keep some global grip on the problem in a way that won't be the case when we study MST.

3 MPC for Geometric Graph Algorithms

This section will focus on presenting and analyzing the ideas put forward in [And+14]. In section 3.1 we begin by discussing the "solve and sketch" framework, which is a general scheme for MPC that is not limited to Minimum Spanning Tree (MST). Then in section 3.2 we address the application to a $(\log_s n)^{O(1)}$ round approximate MST algorithm.

3.1 Solve and Sketch framework

The idea behind the solve and sketch framework is to treat the algorithm that solves local problems as a black-box and instead focus on how the larger problem will be partitioned and local solutions combined. As such, solve and sketch becomes potentially useful for many geometric graph problems other than just MST. Solving particular problems like MST or Earth Mover Distance (EMD) then becomes a matter of designing A_u, which is the algorithm applied to partitioned local to single machines.

The partition of the space is hierarchical with a tree like structure. Each partition of space is split into about $\sqrt{s}$ children, making the total tree depth $O(\log_s n)$. Computation to solve local problems and then combine proceeds bottom-up from the leaves, which contain single points. A sketch of local solutions are sent up to parents. Since parents must take in about $\sqrt{s}$ local solutions from children, those solutions must be of size at most $\sqrt{s}$ to all fit in one machine with memory s. Therefore sketches of solutions must be drastically smaller than the solutions themselves to make slices of computation feasible for single machines as we move up the tree during runtime.

3.1.1 Partition

Without loss of generality, as can be shown, we can assume that all points have integer coordinates in $[0, \Delta]$, where $\Delta = n^{O(1)}$. We denote the partition as P and the levels as $P_0, P_1 ... P_L$. To create P_{l-1} we split P_l into c equal size cubes. Because points have integer coordinates in $[0, \Delta]$, and the whole space in d dimensions has volume Δ^d, we can only recurse into c splits $\log_c \Delta^d = d \log_c \Delta = O(d \log_c n)$ times. Each child cell of C is labeled with an integer in $[c]$ so that during implementation we can keep order of the cells and make reasonable assignments to machines. Ultimately we want to assign cells to machines such that the children of those cells also tend to get assigned to the same machines, to the extent possible.

3.1.2 Unit Step

Formally, we call the algorithm computes on input in individual cells through all partitions P_l $\mathcal{A}_u$. At the lowest level of recursion, $\mathcal{A}_u$ takes the actual points as input. At other levels $\mathcal{A}_u$ takes as input the output from the c children of the cell it's applied to. We limit the output, runtime and total space of $\mathcal{A}_u$ to $p_u(n_u)$, $t_u(n_u)$ and $s_u(n_u)$ respectively.

3.1.3 Solve and Sketch Algorithm

At the heart of the solve and sketch algorithm presented above is how to decide which cells will be assigned to which machines. In each round, we consider all of the cells that are either at the base level of recursion or the unit step has not been applied to their parents. These are the cells that are ready for local computation where the black-box $\mathcal{A}_u$ is then applied. Recall that cells are all numbered from the partition, and furthermore this numbering is *good* in the sense that if $C_2 > C_1$, then the children of C_2 are numbered higher than the children of C_1. This allows the solve and sketch algorithm to sort the cells and make assignments to machines such that machines tend to

Algorithm 1: Implementation of algorithms in the Solve-And-Sketch framework [And+14]

```
input : A set $S$, labeled by a hierarchical partition $P = (P_0, \dots, P_L)$ of degree $c$ such
        that $P_0$ is a partition into singletons.
1  for $r = 1, \dots R$ do
2      Let $\mathcal{C}_r$ be the be the set of non-empty cells $C$ of $P$ such that
           (1) $C \in P_0$ or the unit step $\mathcal{A}_u$ has already been applied to $C$, and
           (2) the unit step has not been applied to the parent of $C$.
3      Sort $\mathcal{C}_r$ according to the induced order. Let the sorted order be $C_1, \dots, C_{n_r}$.
4      Let $p_u(C_i)$ be the output size of cell $C_i$. Compute $h_i = \sum_{j \leq i} p_u(C_j)$ for all $i \leq n_r$
        using a prefix sum algorithm.
5      Consider some cell $i$, and let $j = \lceil \frac{h_i}{s} \rceil$. Machine $j$ receives the output of the unit step
        that has been applied in round $r - 1$ to $C_i$.
6      foreach machine $j$ do
7          for $\ell = 1, \dots, L$ do
8              while there exists a cell $C \in P_\ell$ such that $C \subseteq \bigcup_{i:(j-1)s \leq h_i < js} C_i$ do
9                  Apply the unit step $\mathcal{A}_u$ to $C$, where the inputs are:
                       for $\ell > 1$: the outputs of the unit step for children of $C$
                       for $\ell = 1$: the cells that are children of $C$
10                 (If the output of $\mathcal{A}_u$ is larger than its input size, the algorithm instead just
                    outputs the input; appropriately marked to be "lazy evaluated".)
```

get assigned cells with their descendants. Each machine, then computes $\mathcal{A}_u$ on its lowest level cells yet to be computed, and then the parents of those cells up until the machine has no cells left to which $\mathcal{A}_u$ need be applied. Rounds progress until all the computation is done. Since the entire input fits into the memory of all the machines, we should expect to compute $\mathcal{A}_u$ on P_0 in the first round. Then, since $\mathcal{A}_u$ returns sublinear output, we should expect to be able to move up at least 1 level of recursion in each round.

The correctness of the Solve-and-Sketch algorithm can be formally stated as follows:

Theorem 3.1 (Solve-And-Sketch [And+14])**.** *Fix space parameter $s = (\log n)^{\Omega(d)}$ of the MPC model. Suppose there is a unit step algorithm using local time $t_u(n_u)$, space $s_u(n_u)$, and output size $p_u(n_u)$ on input of size n_u. Assume the functions t_u, s_u, p_u are non-decreasing, and also satisfy: $s_u(p_u(s)) \leq s^{1/3}$ and $p_u(s) \leq s^{1/3}$. Then we can set $c = s^{\Theta(1)}$ and $L = O(\log_s n)$ in the partitioning from above, and we can implement the resulting Solve-And-Sketch algorithm in the MPC model in $(\log_s n)^{O(1)}$ rounds. Local runtime is $s \cdot t_u(s) \cdot (\log n)^{O(1)}$ (per machine per round).*

3.2 Minimum Spanning Tree

In this section we present the intuition behind the following result:

Theorem 3.2 (Approximate MST [And+14])**.** *Let $\varepsilon > 0$, and $s \geq (\varepsilon^{-1} \log_s n)^{O(1)}$. Then there exists an MPC algorithm that, on input a set S in $\mathbb{R}^d$ runs in $(\log_s n)^{O(1)}$ rounds and outputs a spanning tree of cost (under the Euclidean distance metric ℓ_2^d for $d = O(1)$) at most $1 + \varepsilon$ factor larger than the optimal. Moreover, the running time per machine is near linear in the input size n_u, namely $O(n_u \varepsilon^{-d} \log^{O(1)} n_u)$.*

3.2.1 Main Idea

It's instructive to first consider a naive and greedy approach to the problem, ignoring for a moment that greedy algorithms don't actually work for MST. For this, we could split the space up into c child spaces, and then recursively split each child space until the smallest blocks of space contain only 1 point, at which level the MST is trivial. We could then seek to solve MST on the smallest unsolved sub-problems. As we move up the levels of recursion our job becomes to connect the MST forest produced in the lower levels.

We can get a sense that this idea would adapt well to the MPC model. Starting from the lowest level of recursive space splitting, each machine can get a patch of space with few enough points that can fit in memory. After the MST forest is computed, one tree per machine, we can then sketch those trees to be sent up to the next higher level of recursion. The sketch will be some way of representing key information and points in each tree with a memory footprint much smaller than the whole tree. The idea that we don't need to see an MST forest at full resolution to connect it approximately well is fairly intuitive and central to the whole scheme.

As sketches are themselves recursively sketched, at the highest level of recursion a single machine can connect the forest. The key is that since the output to a MST calculation for a block of space is sketched, a single machine can then take on the order of s of these outputs into memory for the next level of computation. This allows us to do the recursive space splitting with a branching factor of s. By changing the resolution of the computation by a constant factor during each round, we could get $O(\log_s n)$ rounds.

The problem with all of this is that local solutions to MST are not necessarily globally optimal. For example, at some level of the recursive space splitting, if a space has just two points near opposite boundaries, these points would have to be connected to each other at a cost roughly the distance of the block. Alternatively, there may be other points much closer to each on the other sides of the boundaries which would likely make for a cheaper spanning tree. This is part of the reason why Kruskal's MST algorithm works by taking the shortest edge that can connect two forests *globally.* The solution is to only use edges that are within ε times the width of the sub-blocks at any level of recursion. Thus, rather than providing fully connected MSTs for the sub-blocks, the local solutions themselves are really only responsible for providing a forest, wherein each tree consists of points all within ε of the rest of the tree. A sketch, or blurred picture, of this forest can then be sent up to the next level of the recursion, along with connectivity information, for the forest to then be roughly connected up. The algorithm we will describe from [And+14] will take many cues from Kruskal's MST algorithm, and the approximation guarantee may be proved by comparing the resulting tree to what Kruskal's would have produced on a slightly modified graph.

3.2.2 Hierarchical Partitioning

Topology. We note $\rho(u, v)$, the distance between two points u, v. For a set of points $S' \subseteq S$, the diameter $\Delta(S')$ is the maximum distance between two points of S'.

Deterministic hierarchical partition. First, we give more details about how to describe a hierarchical partition of points into cells, without any randomness so far. In Section 3.1.1 we gave a general intuition of the Euclidean partition where each level is simply created by splitting each cell of the previous level into c equally sized cubes, where the points have integer coordinates.

Let us now consider the slightly more general case where the points are contained in $S \subseteq [0, \Delta(S)]^d$, without necessarily having integer coordinates. For a given deterministic hierarchical partition P

with L levels $P_0, \ldots, P_L$ (where $P_L = \{S\}$ and $P_0 = \{\{p_1\}, \ldots, \{p_{|S|}\}\}$), the *diameter* of a level P_ℓ is defined by $\Delta(P_\ell) = \max_{C \in P_\ell} \Delta(C)$.

We define $\Delta_\ell := c^{(\ell-L)/d}\Delta(S)$, that verifies $\Delta(P_\ell) \leq \Delta_\ell$. We can check that $\Delta_L = \Delta(S)$ and $\Delta_0 = \frac{1}{c^{L/d}}\Delta(S)$. To summarize, for the Euclidean grid partition we have:

$$\begin{aligned} &\forall \ell \in \{0, \ldots, L\}, \forall C \in P_\ell, \forall (u, v) \in C, \\ &\rho(u, v) \leq \Delta(C) \leq \Delta(P_\ell) \leq \Delta_\ell \end{aligned} \tag{8}$$

Randomized hierarchical partition. We saw that there is an obvious problem when we use a naive grid to split the MST problem into local cells: two points in adjacent cells will not get connected even if they are very close to each other but on opposite sides of the border.

The solution is to randomize the grid. We keep the same deterministic construction presented earlier, but we add a randomized offset to the grid. In practice, we take a slightly bigger grid of diameter 2Δ, and we place its bottom corner randomly in $[-\Delta, 0]^d$. If we shake the grid before throwing it onto the set of points, we must be quite unlucky to have a cell border that goes right between two close points.

More formally, we now consider the uniform distribution of hierarchical partitions instead of sticking to a single deterministic grid. Following this distribution, the probability of cutting an edge is bounded by:

$$\forall x, y \in S, \Pr[C_\ell(x) \neq C_\ell(y)] = O\left(d \cdot \frac{\rho(x, y)}{\Delta_\ell}\right) \tag{9}$$

Distance-preserving hierarchical partition. In the general setting, we do not have to necessarily define the partition as a grid, especially if we are not working in Euclidean spaces. It is sufficient to have a distribution that essentially verifies Equation 8 and 9, with potentially different constants.

3.2.3 Unit Step Algorithm

Thanks to the above partition and Theorem 3.1, it is sufficient to give an algorithm for a unit step (that verifies the complexity requirements) to obtain a global MPC algorithm that computes an approximate MST. The unit step algorithm is presented in Algorithm 2.

Intuition. This unit step algorithm is close to Kruskal's algorithm, as presented in Section 3.2.1: at each level the unit step algorithm browses the connected components in order and connects the two closest ones. However, we stop this process as soon as edges are longer than $\varepsilon\Delta_\ell$, in order to avoid accepting long edges that might be locally optimal but globally wrong.

Each unit step for a cell C at level ℓ is actually producing two kinds of outputs:

- A set of new edges selected during the step at level ℓ. These edges are part of the original graph, and they connect sub-trees already produced by the children of C. At the last level L, for $P_L = \{S\}$, the output of the unit step will be the approximate MST. This means that as soon as a unit step is accepting an edge, this edge will be present in the global solution.
- A sketch that we will be combined with the sketches output by C's siblings to serve as input for C's parent. We describe the sketch below.

Algorithm 2: Unit Step at Level ℓ [And+14]

```
  input : Cell $C \in P_\ell$; a collection $V(C)$ of points in $C$, and a partition $Q = \{Q_1, \dots Q_k\}$ of
          $V(C)$ into previously computed connected components.
1 repeat
2     Let $\tau = \min_{\substack{i,j \\ i \neq j}} \min_{u \in Q_i, v \in Q_j} \rho(u, v)$.
3     Find $u \in Q_i$ and $v \in Q_j$ for some $i, j : i \neq j$ such that $\rho(u, v) \leq (1 + \varepsilon)\tau$.
4     Let $\theta = \rho(u, v)$.
5     if $\theta \leq \varepsilon \Delta_\ell$ then
6         Output tree edge $(u, v)$.
7         Merge $Q_i$ and $Q_j$ and update $Q$.
8 until $\theta > \varepsilon \Delta_\ell$;
  output: $V' \subseteq V$, an $\varepsilon^2 \Delta_\ell$-covering for $C$, the partition $Q(V')$ induced by $Q$ on $V'$.
```

Sketch. We explained in Section 3.2.1 that the sketch's role is to blur the input for the next level, in order to keep a low space complexity. However, we want each unit step to output real edges of the graph in order to obtain an MST at level L, so we cannot choose arbitrary points (such as the mean of some points).

Hence, the sketch V' output by the unit step algorithm is a subset of its input points V. We will choose V' to be an $\varepsilon^2 \Delta_\ell$ covering for C, which means that $\forall x \in C, \exists y \in V' : \rho(x, y) \leq \varepsilon^2 \Delta_\ell$. This sketching is not too aggressive: we simply cover the input points with balls of radius $\varepsilon^2 \Delta_\ell$, and only forward the centers of these balls to the next level.

We are not loosing any information because most of the points covered by a given ball have already been connected when we parsed the edges up to $\varepsilon \Delta_\ell > \varepsilon^2 \Delta_\ell$, except for edges that were crossing a cell boundary (but this happens with low probability).

At each level, we zoom out the sketching size from $\varepsilon^2 \Delta_\ell$ to $\varepsilon^2 \Delta_{\ell+1}$, i.e. we scale the balls by a factor $c^{1/d}$ in radius, which is a factor c in volume: it corresponds to the branching factor.

Also, to speed up the process we keep track of the connected components that have been built so far and include them along the points of the sketch.

Approximate nearest-neighbor optimization. The unit step algorithm adds an optimization to improve the running time compared to Kruskal's algorithm: instead of connecting the two closest pair of connected components, we connect the approximately closest pair with an approximation factor $(1 + \varepsilon)$. Indeed, we only want an approximate result and approximate nearest neighbor search provides better complexity.

4 MPC for Dynamic Programming Algorithms

4.1 Key Properties

There are two key properties that a problem should exhibit in order to be able to leverage efficient distributed algorithms. These properties are *monotonicity* and *decomposability*, as described in [IMS17]. Monotonicity alone can lead to algorithms running in $O(\log n)$ rounds for some problems, while decomposability may enable a further reduction to just $O(1)$ rounds.

Definition 4.1 (Monotonicity, Decomposability)**.**

The following definitions are interpreted from [IMS17].

1. ***Monotonicity.*** *If a maximization problem is broken up into several sub-problems, then the solution to any of the sub-problems* <u>*cannot*</u> *be greater than that of the original problem (and vice-versa for a minimization problem).*

   The intuition here is that there must be a trade-off in being able to divide the original problem into smaller units – valid sub-problems (which are smaller and therefore easier to solve than the original problem) should not also be able to attain better optimal solutions. Accordingly, sub-problem solutions will lead to valid components of the larger solution.

2. ***Decomposability.*** *If the input can be split into a two-part hierarchy, where the top tier elements are called groups, and each group is comprised of a distinct set of bottom-tier partial inputs called blocks, then:*

   - *consolidating sub-solutions from individual groups can lead to a nearly optimal solution*
   - *each group can achieve a nearly optimal sub-solution using only $O(1)$ of its blocks*

   The (loose) intuition here is that "a chain is only as strong as its weakest link", or in other words: a problem that can be broken down into (parallel) units can only be as difficult as its most difficult unit. If each group computes a nearly optimal sub-solution in $O(1)$ rounds as it only requires the use of inputs from $O(1)$ of its blocks, then achieving the overall (approximate) solution simply requires a final consolidation.

In the rest of this section, we focus on the Weighted Interval Selection problem, exemplifying how the properties defined above can enable the use of efficient distributed algorithms.

4.2 Weighted Interval Selection

Definition 4.2 (Weighted Interval Selection (WIS) problem)**.** *Consider an interval $I = (a, b)$ on the real number line. Given a collection of n such intervals $\{I_i = (a_i, b_i)\}$ and corresponding (strictly) positive weights $\{w_i\}$, the goal is to choose a subset of non-overlapping intervals that maximizes the sum of the weights in that subset.*

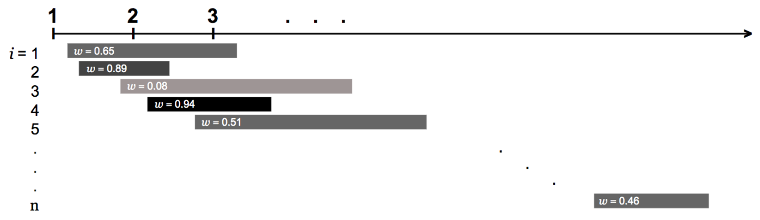

Figure 1: Example of a collection of weighted intervals.

When all of the data can fit onto a single machine, the above problem can be solved in polynomial time. The following outlines a simple dynamic programming approach to find an exact solution.

Given a selection of weighted intervals S, let $\mathrm{OPT}(S)$ denote the greatest possible weight sum of non-overlapping intervals that can be achieved using some optimal subset in S.

Assuming that the intervals I_i are sorted by their start-points a_i, define $A(i) = \text{OPT}(\{I_i, I_{i+1}, ..., I_n\})$. Accordingly, the optimal solution to WIS is equivalent to $w^* = A(1)$. Note that $A(i)$ can be broken down using the following recursive formula:

$$A(i) = \max\{A(i+1),\ w_i + A(j)\} \tag{10}$$

$$\text{where } \ j = \min_{a_k > b_i}\{k\} \ \text{ and } \ A(n{+}1) = 0$$

In other words, the optimal weight sum (starting at interval I_i) is simply the better of two choices: (a) excluding interval I_i and just using the optimal weight sum starting at I_{i+1}, or (b) including interval I_i, and using the optimal weight sum over the set of intervals starting at I_j (where j is the index of the left-most non-overlapping interval with I_i).

This leads us to a simple algorithm:

Algorithm 3: Single-Machine Weighted Interval Selection

```
  input  : Set of $n$ intervals $\{I_i = (a_i, b_i)\}$
1 sort: $\{I_i = (a_i, b_i)\}$     (in order of increasing $a_i$)
2 set: $A(n{+}1) = 0,\ a_{n+1} = \infty$
3 for $i = n, n{-}1, ..., 1$ do
4     compute: $j = \min_{a_k > b_i}\{k\}$     (using binary search)
5     set: $A(i) = \max\{A(i{+}1), w_i + A(j)\}$
  output: $w^* = A(1)$
```

The initial sorting is achieved in $O(n \log n)$. There are $O(n)$ for-loop iterations, each requiring $O(\log n)$ to compute j, for a total run-time of $O(n \log n)$.

4.2.1 MPC Weighted Interval Selection

When the number of intervals n is very large (where they no longer fit onto a single machine), a different approach must be taken. For some fixed constant $0 < \delta < 1$, if individual machines only have memory $s = \tilde{O}(n^\delta)$, the data has to be distributed over m machines, and MPC is employed. If n is much larger than a single machine's memory, then δ is small; hence, $m = n/s \approx O(n)$.

Leveraging the key properties defined in section 4.1 above, the following theorems outline efficient distributed algorithms to the WIS problem. Again, due to the memory constraints, *sketching* is used (similar to Sec. 3.2.3), thereby returning only an approximate solution, albeit, guaranteed to be within a factor of $1{-}\varepsilon$ of the optimum, for some fixed constant $\varepsilon > 0$.

In order to simplify the proofs (and without loss of generality), we assume that the given intervals are sorted and modified using the following procedure, which can be achieved in $O(1)$ rounds:

1. Sort intervals I_i in order of increasing start-points a_i. Note that this can be achieved in $O(1)$ rounds, by appropriately adapting the sample-sort algorithm as stated in [IMS17].
2. Ensure that no start-points and/or end-points have the same value by infinitesimally perturbing any set of points that share the same value.
3. Rescale the interval weights such that the smallest weight is ε/n and the largest is 1. This is a simple linear transformation and can be easily inverted to correct the returned solution.

4. Maintaining their ordering, "move" the first $s = n/m$ interval start-points into the range $(1, 2)$, ensuring that any end-points caught in the mix are "dragged along" to maintain their ordering. Move for the next s start-points into $(2, 3)$, then the next s into $(3, 4)$, etc. Note that these modifications do not effect the optimum or any intermediate computations thereof since the relative ordering of the intervals is carefully kept consistent. Let M_1 be the collection of all intervals with start-points in the range $(1, 2)$, M_2 those with start-points in $(2, 3)$, etc. Accordingly, each individual machine q is assigned the s intervals in M_q respectively.

4.2.2 Leveraging Monotonicity for $O(\log n)$-round WIS

Theorem 4.3 (WIS in $O(\log n)$ rounds [IMS17]). *There exists a $(1-\varepsilon)$-approximate algorithm for the Weighted Interval Selection problem running in $O(\log n)$ rounds, when each machine has memory $\tilde{O}(n^\delta)$, for any $0 < \varepsilon, \delta$.*

Proof.

Let $B(i, j) = \mathrm{OPT}(\{I_k = (a_k, b_k) \mid a_i \leq a_k < b_k < a_j\})$, the optimal weight sum of the subset of intervals that start at or after a_i and that end before a_j. Note that for any $i \leq i'$, $j' \leq j$, then $\mathrm{OPT}(i', j') \leq \mathrm{OPT}(i, j)$. This is clear since the new smaller collection corresponding to $B(i', j')$ has fewer intervals to choose from; hence, cannot have a better optimal solution. Accordingly, the sub-problem defined by B exhibits *monotonicity.*

However, dealing with pairs of indices (i, j) requires $O(s^2)$ storage, which is unavailable. Instead, for reasons that will become apparent shortly, given a weight sum w, define $C(i, w)$ as a sort of dual of $B(i, j)$:

$$C(i, w) = \min_{B(i,j) \geq w} \{j\} \quad \text{(or } \infty \text{ if no such } j \text{ exists, ie. } C(i, w) \text{ is \textit{infeasible})} \tag{11}$$

Similar to the recursion in Eq. 10, $C(i, w)$ can be computed using the recursion:

$$C(i, w) = \min_{\substack{u,v \geq 0 \\ u+v=w}} \{C(j, v) \mid j = C(i, u)\} \tag{12}$$

which can be interpreted as finding a valid break (ie. one that keeps the same total weight sum) in the set of intervals corresponding to $C(i, w)$, where the original interval $[i, k)$ is broken into two intervals $[i, j)$ and $[j, k)$, (one of which may be empty).

Note also, that using this representation, the optimal solution to WIS is equivalent to:

$$w^* = \max_{C(1,w) < \infty} \{w\} \tag{13}$$

Since the problem consists of n interval weights with each one's value between ε/n and 1, weight sums are in the range $[\varepsilon/n, n]$, but there may be up to $2^n - 1$ unique values; much more than can be stored onto any single machine – and far worse than the $O(s^2)$ that we tried to avoid! This is where *sketching* comes into play. Instead of considering all possibilities, the weight sums are sketched down to the set W, parameterized by a cleverly chosen constant $\eta = \varepsilon/\log m$:

$$W = \{0,\ \varepsilon/n,\ (1+\eta)\varepsilon/n,\ (1+\eta)^2\varepsilon/n,\ ...,\ n\} \tag{14}$$

$$|W| \approx \frac{\log(n^2/\varepsilon)}{\log(1+\eta)} \approx \frac{\log m}{\varepsilon} \log\left(\frac{n^2}{\varepsilon}\right) = O\Big(\frac{1}{\varepsilon} \log^2 n\Big) = \tilde{O}(1) \tag{15}$$

Further, due to monotonicity, if $C(i, w)$ returns a feasible solution, then $C(i, w')$ also returns a feasible solution for any $w' \leq w$, and if $w' \approx w$, we lose only a small weight, at most $w - w'$. As a result, we are able to approximate weights by slight under-estimation without losing feasibility or incurring too high of an approximation error. This subtle but crucial property of monotonicity is what allows us to leverage sketching for this problem.

Armed with the sketched set of weight sums and making use of dynamic programming, each machine q is able to store its own table of estimates $C'_q(i, w)$ for $i \in M_q$, requiring $s \cdot |W| = \tilde{O}(n^\delta)$ storage space per machine; which is now within the MPC memory constraints.

We can now outline the algorithm:

Algorithm 4: MPC Weighted Interval Selection

```
   input : Set of $n$ intervals $\{I_i = (a_i, b_i)\}$
1  modify inputs   (as outlined in the procedure in 4.2.1)
   // initialize locally feasible $C'(i,w)$
2  parfor $q \in [m]$ do
3      for $i \in M_q$ and $w \in W$ do
4          init: $C'_q(i,w)$   (ie. the correct value of $C(i,w)$, or $\infty$ if locally infeasible)
   // update $C'(i,w)$ for progressively larger $w$ with each round
5  for round $r = 1, ..., \log m$ do
6      parfor $q \in [m]$ do
7          for $i \in M_q$ and $w \in W$ do
8              reset: $S_{qi} = \{\}$
9              for $u, v \in W,\ w/(1+\eta) \leq u + v \leq w$ do
10                 get: $j = C'_q(i,u)$   (from local machine $q$)
11                 if $j < \infty$ then
12                     request: $k = C'_p(j,v)$   (from machine $p = \lceil j/s \rceil$)
13                     update: $S_{qi} \leftarrow S_{qi} \cup \{k\}$
14             update: $C'_q(i,w) \leftarrow \min\{S_{qi}\}$
   output: $w' = \max_{w \in W,\ C'_1(1,w) < \infty} \{w\}$
```

Upon initialization, each machine sets the correct value for $C'(i, w)$ for weight sums $w \in W$ that are locally feasible. With each round, this information is "doubled" across machines, until finally, after $\log m$ rounds, all m machines have access to all the information; especially $C'_1(1, w)$.

In every iteration of the outer update loop, each machine requests non-local values of $C'(j, v)$ for each of its s intervals and $|W|$ weight sums, for a total of $\tilde{O}(n^\delta)$ requests. This fits is within the allotted MPC bandwidth limit; hence, can be achieved in $O(1)$ rounds.

Due to the "sketchiness" of the weight sums, if w^* is the true optimum and w' is the solution returned by the algorithm, then: $w^*/(1+\eta) \leq w' \leq w^*$. Furthermore, a $(1+\eta)$-approximate equality (ie. $u + v \approx w$) has to be permitted in order to facilitate breaking down $C'(i, w)$ into sub-problems $C'(i, u)$ and $C'(j, v)$. Consequently, this approximation *deteriorates* intermediate weight sums by a factor of at most $1+\eta$ with each round. Hence, after $\log m$ rounds:

$$w^* \geq w' \geq \frac{w^*}{(1+\eta)^{\log m}} = \frac{w^*}{(1+\varepsilon/\log m)^{\log m}} \approx \frac{w^*}{1+\varepsilon} > (1-\varepsilon)\, w^* \tag{16}$$

By construction, the given algorithm terminates in $\log m = O(\log n)$ rounds, and uses $s = \tilde{O}(n^\delta)$ storage space and communication bandwidth, and the resulting solution is within a $1-\varepsilon$ factor of the optimum, thereby completing the proof.

□

4.2.3 Leveraging Decomposability for $O(1)$-round WIS

Note that in the theorem and proof above, only monotonicity is invoked, and the claim is that WIS can be approximately solved in $O(\log n)$ rounds. As it turns out, the WIS problem also exhibits decomposability, facilitating an approximate solution in $O(1)$ rounds.

In splitting input intervals over the m machines, each machine can be seen as a separate block q. An interval I_i that starts at block $q = \lfloor a_i \rfloor$, may either be *local* (ie. ending in the same block), or it may be crossing (ie. ending in a later block $\lfloor b_i \rfloor > q$). Consider the set of optimally-selected crossing intervals (ie. those that comprise of the optimal solution). For some fixed constant L, it is possible to pool all the blocks into groups such that each group has $L = O(1)$ (optimally-selected) crossing intervals (the last group is allowed to have less). Accordingly, each group's optimal solution can be computed in $O(1)$ rounds.

Of course, the issue is how to a-priori determine this grouping of blocks without already knowing the solution. The following theorem and associated procedure ([IMS17]) outlines a method to intelligently initialize a set of groups and then tweak it in $O(1)$ rounds so as to achieve a nearly optimal group assignment (and thereby output an approximately-optimal final solution).

Theorem 4.4 (WIS in $O(1)$ rounds [IMS17])**.** *There exists a $(1-\varepsilon)$-approximation for the Weighted Interval Selection problem running in $O\left(\frac{1}{\delta}\left(\log\frac{1}{\varepsilon} + \log\frac{1}{\delta}\right)\right)$ rounds, when each machine has memory $\tilde{O}(n^\delta)$, for any $0 < \varepsilon, \delta$.*

While this is an interesting result, the proof of this theorem is involved and does not add much to the intuition behind the concepts defined in the paper. We direct the reader to [IMS17] for details.

References

[Gha18] Mohsen Ghaffari. *Chapter 6: Parallel Algorithms.* 2018. URL: `https://people.inf.ethz.ch/gmohsen/CHParallel18.pdf` (visited on 04/18/2020).

[Gha19] Mohsen Ghaffari. *Massively Parallel Algorithms.* 2019. URL: `http://people.csail.mit.edu/ghaffari/MPA19/Notes/MPA.pdf` (visited on 03/30/2020).

[And+14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. "Parallel algorithms for geometric graph problems". In: *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014.* Ed. by David B. Shmoys. ACM, 2014, pp. 574–583. DOI: `10.1145/2591796.2591805`. URL: `https://doi.org/10.1145/2591796.2591805`.

[IMS17] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. “Efficient massively parallel methods for dynamic programming”. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing - STOC 2017*. the 49th Annual ACM SIGACT Symposium. Montreal, Canada: ACM Press, 2017, pp. 798–811. ISBN: 978-1-4503-4528-6. DOI: 10.1145/3055399.3055460. URL: http://dl.acm.org/citation.cfm?doid=3055399.3055460 (visited on 03/25/2020).

[Val90] Leslie G. Valiant. “A Bridging Model for Parallel Computation”. In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: https://doi.org/10.1145/79173.79181.