Algorithms for Massive Data Science — **Lecture date: April 10, 2019**
Instructor: Ben Moseley — **Scribe: Thomas Lavastida**

1 Dynamic Programming in MPC

Dynamic programming is a standard algorithm design technique that has been quite powerful in the sequential setting. However, typical dynamic programming algorithms have long chains of sequential dependencies that make it difficult to adapt to parallel and distributed settings such as MPC. In this lecture we demonstrate a technique that can sometimes be used to convert a sequential DP to one that can be executed efficiently in MPC.

Here is a sample of what is known so far. There are $1+\epsilon$-approximation algorithms using $O(\frac{1}{\epsilon\delta})$ rounds of MPC with $O(n)$ total memory and $O(n^\delta)$ memory per machine for constants $\epsilon, \delta > 0$ for the following problems:

1. Optimal Binary Search Tree
2. Longest Increasing Subsequence
3. Weighted Interval Selection

These results are due to [3]. Recently, there have been other results for problems such as Longest Common Subsequence due to [2]. We will show a weaker result that will capture many of the crucial ideas. We will give an MPC algorithm for Weighted Interval Selection running in $O(\log n)$ rounds and using $\tilde{O}(n^\delta)$ memory per machine and $\tilde{O}(n)$ total memory.

2 Weighted Interval Selection

In the Weighted Interval Selection Problem there are n intervals, where the i'th interval is of the form $I_i = [s_i, e_i]$ and has an integer weight $w_i > 0$. A subset of intervals is independent if no two intervals in the subset intersect. The goal is to choose a maximum weight independent subset of intervals.

There is a classic dynamic program that goes as follows. First sort the intervals by their start points so that $s_1 \leq s_2 \leq \ldots \leq s_n$. We define $A(i)$ to be the value of the optimal solution on $I_i, I_{i+1}, \ldots, I_n$. Clearly, $A(n) = w_n$ and we compute $A(i)$ inductively for $i < n$ as follows:

$$A(i) = \max \begin{cases} A(i+1) \\ A(j) + w_i \quad j = \min\{j' \mid j' > i \text{ and } s_{j'} > e_i\} \end{cases} \tag{1}$$

Intuitively, we either drop I_i and take the solution computed inductively starting with I_{i+1}, or we take I_i gaining weight w_i and combine it with the solution computed inductively starting with I_j, where j is the earliest interval starting after I_i that doesn't intersect with it. Note that the above recurrence only computes the *optimal value*. There are standard techniques to recover the solution once we have computed this recurrence, so we only focus on computing the value.

In the distributed setting we will be interested in computing $(1 \pm \epsilon)$-approximations, which means we can assume that that the weights w_i are bounded by a polynomial in n. Let $\Delta = \max_i w_i$ be the maximum weight. Now consider creating a new instance by throwing away all intervals with

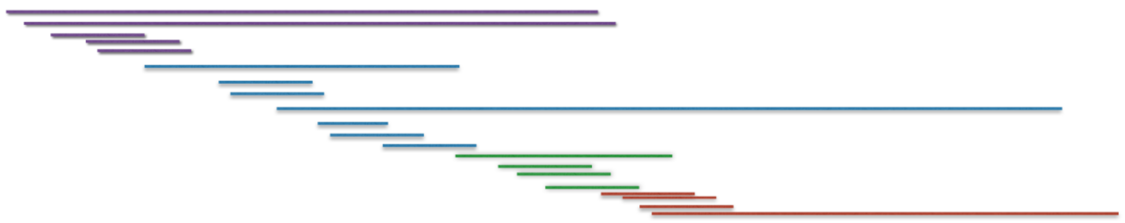

Figure 1: The different colors correspond to intervals on different machines. Concatenating locally optimal solutions does not guarantee feasibility.

$w_i < \Delta/n^3$, i.e. let the new instance be $S = \{i \mid w_i \geq \Delta/n^3\}$. Let OPT be the optimal value on the original instance and $OPT(S)$ be the optimal value on instance S. Since $OPT \geq \Delta$, then we have

$$OPT(S) \geq OPT - \sum_{i \notin S} w_i \geq OPT - \sum_{i \in S} \frac{\Delta}{n^3} \geq OPT(1 - 1/n^2)$$

So the optimal solution on the new instance S preserves a large fraction of the original optimal value. Moreover, by rescaling the weights, i.e. setting $w_i' = w_i(n^3/\Delta)$ implies that $1 \leq w_i' \leq n^3$ for all $i \in S$, giving polynomially bounded weights.

3 First Attempts at Distributed Algorithms

As a first attempt, let's consider simulating the standard DP in MPC in a naive way. Start by distributing the n intervals across m machines in sorted order by increasing start times (sorting can be easily done in MPC, see [1]). Now consider computing the recurrence $A(i)$ on each machine *locally*. Can we easily combine solutions between machines? Not really, as Figure 1 shows that maintaining feasibility when combining solutions is difficult.

So we need a technique that will handle the interactions between intervals on different machines. Let's define a subproblem that will store information about conflicts. Let $B(i, j)$ be the optimal solution on intervals that start no earlier than s_i and *end* by s_j. We give a recurrence for $B(i, j)$. Clearly, $B(i, i+1) = w_i$ if $e_i \leq s_{i+1}$ and $B(i, i+1) = 0$ otherwise for each i. Inductively, for i, j such that $j > i + 1$ we have

$$B(i, j) = \max_{i < j' < j} \{B(i, j') + B(j', j)\} \tag{2}$$

Note that since we ordered the intervals by increasing start time, $B(i, j)$ can be computed correctly on a machines local set of intervals. So now suppose that each machine computes $B(i, j)$ on its local set of intervals. Afterwards, we communicate the local solutions to one machine and combine to find the optimal solution. We leave it as an exercise to the reader to show how local solutions can be combined into a global solution.

What is the issue with this idea? Well if we compute $B(i, j)$ on a local set of n/m intervals, then we need at least $\Omega((n/m)^2)$ space to store this solution. After communicating, we need $\Omega(m(n/m)^2) = \Omega(n^2/m)$, which is a superlinear amount of space per machine. This subproblem has nice properties, its main issue is that it is costly in space. Instead of generating yet another new idea we will try to "fix" the $B(i, j)$ subproblem to deal with the space issues.

4 An Approach that Works

Suppose that $B(i,j)$ equals some weight w. The main idea is that we will "swap" the weight w with the index j. Define $C(i,w) = \min\{j \mid B(i,j) \geq w\}$. That is, $C(i,w)$ is the least index j such that there is an independent subset with total weight at least w where all intervals start after s_i and end before s_j. We set $C(i,w) = \infty$ if the set $\{j \mid B(i,j) \geq w\} = \emptyset$. Note that since $B(i,j') \geq B(i,j)$ for $j' \geq j$, this implies that $C(i,w') \geq C(i,w)$ for $w' \geq w$. This "monotonicity" is a key property we want our subproblems to have in general.

We can compute $C(i,w)$ using the following recurrence. For the base case, we have $C(i,0) = i$ and $C(n,w) = \infty$ for all w. Inductively, for $w > 0$ we have:

$$C(i,w) = \min_{w_1,w_2 \geq 0} \{j_2 \mid w_1 + w_2 = w, j_1 = C(i,w_1), j_2 = C(j_1,w_2)\} \tag{3}$$

Intuitively, we guess a "split" of w into w_1 and w_2 and concatenate solutions corresponding to picking up weight at least w_1 and weight at least w_2. If we compute $C(i.w)$ for all w, then this captures the subproblem $B(i,j)$ exactly, albeit in a different form (as an exercise show how to convert back and forth from $B(,ij)$ and $C(i,w)$). However, we will show that we can compute approximate solutions using $C(i,w)$ in *significantly* less space, which will then lead to an efficient MPC algorithm.

The key idea to reducing the space cost is to "sketch" the weights. Specifically we want to compute the recurrence only for w's of the form $(1+\epsilon/\log n)^k$ for some non-negative integer k. Note that since $\Delta = \max_i w_i = O(n^3)$, we have that all w we need to solve the recurrence for are in the interval $[0, n\Delta] = [0, O(n^4)]$. This implies that for sketching we only need to consider exponents $k \in [O(\frac{1}{\epsilon}\log^2 n)]$.

Let $C'(i,w)$ be the approximation of $C(i,w)$ where we only store weights of the form $(1+\epsilon/\log n)^k$. Note that storing $C(i,w)$ for all i,w requires space $O(n\Delta) = O(n^4)$, while storing $C'(i,w)$ for the restricted set of weights requires space $O(\frac{1}{\epsilon}n\log^2 n)$. So approximating the weights greatly improves the space requirements.

Computing $C'(i,w)$ is similary to computing $C(i,w)$. Consider the following recurrence:

$$C'(i,w) = \min_{w_1,w_2 \geq 0} \left\{ j_2 \mid \frac{w}{1+\frac{\epsilon}{\log n}} \leq w_1 + w_2 \leq w(1+\frac{\epsilon}{\log n}), j_1 = C(i,w_1), j_2 = C(j_1,w_2) \right\} \tag{4}$$

Note that we lose a factor of $(1+\epsilon/\log n)$ each time we compute this recurrence. If we lose this $\Omega(\log^2 n)$ times, this will yield a very poor approximation. However if we only lose this factor $O(\log n)$ times then this will result in a $(1-\epsilon)$-approximation. Our MPC algorithm will account for this.

5 An Efficient MPC Algorithm

We now formally describe the MPC algorithm for weighted interval selection. Let m be the number of machines and n be the number of intervals. The idea is to "combine" subproblem solutions over several rounds. We use $w' \approx_\epsilon$ to denote that $w/(1+\epsilon) \leq w' \leq (1+\epsilon)w$ as a shorthand.

First we show that Algorithm 1 fits into the MPC model.

Algorithm 1 MPC Weighted Interval Selection

```
1: Evenly distribute the intervals across the machines sorted in order of increasing start time
2: Machine $k$ also stores a copy of machine $k+1$'s intervals
3: Each machine computes $C(i,w)$ exactly for each local interval
4: Let $C'_\ell(i,w)$ be the partial solution computed in iteration $\ell$
5: Initially $C'_0(i,w) = C(i,w)$
6: for $\ell = 0, 1, 2, \ldots, \log(m)$ do
7:     Each machine does the following in parallel:
8:     for Each $i$ interval on machine $k$ and sketched weight $w$ do
9:         for Each guess $w_1 + w_2 \approx_{\frac{\epsilon}{\log n}} w$ do
10:            $j_1 \leftarrow C'_\ell(i, w_1)$ stored locally
11:            Request $j_2 \leftarrow C'_\ell(j_1, w_2)$ from machine that stores $j_1$
12:        end for
13:        $C'_{\ell+1}(i,w) \leftarrow$ minimum $j_2$ value requested
14:    end for
15: end for
16: Return $\max\{w \mid C'_{\log m}(1,w) < \infty\}$ as the solution value
```

Lemma 1 *Algorithm 1 can be implemented in the MPC model with m machines using* $O(\frac{n}{m}\log^2(n))$ *memory per machine and each machine communicates* $O(\frac{n}{m}\log^4(n))$ *information per iteration. The algorithm terminates in* $O(\log m)$ *rounds.*

Proof: Note each machine stores $O(n/m)$ intervals and thus storing $C'(i,w)$ for each local interval and all weights w takes space at most $O(\frac{n}{m}\log^2(n))$ since there are $O(n/m)$ intervals and $O(\log^2(n))$ different weights.

Now fix an iteration ℓ. A machine needs to communicate 1 value for each combination of local interval and guess of wieghts w_1, w_2, of which there are $O(\frac{n}{m}\log^4(n))$ in total.

The algorithm runs in several iterations. Note that each iteration can be done in $O(1)$ rounds. The algorithm terminates in $O(\log m)$ iterations by definition, hence it terminates in $O(\log m)$ rounds. □

Lemma 2 *Algorithm 1 computes a* $(1 - O(\epsilon))$*-approximation to the maximum weight independent subset of intervals.*

Proof: The weight of the solution given by the subproblems $C'_\ell(i,w)$ corresponds to the largest weight w such that $C'_\ell(1,w) < \infty$. We show the following stronger claim which implies the lemma. Let w_ℓ be the largest such that $C'_\ell(1,w_\ell) < \infty$. Let w^*_ℓ be the largest weight such that interval $j^* = C(i, w^*_\ell)$ is stored on one of the first 2^ℓ machines. Then $w_\ell \geq (1 - \epsilon/\log n)^\ell w^*_\ell \geq (1 - O(\epsilon))w^*_\ell$ for $\ell \leq \log n$.

We prove the claim by induction on ℓ. The base case for $\ell = 0$ is clear since we set $C'_0(i,w) = C(i,w)$ using only local information. Now consider the inductive step. In iteration ℓ we construct w_ℓ by combining solutions on different machines and rounding down, i.e. $w_\ell = (w_1 + w_2)/(1 + \epsilon/\log n)$.

Similarly, w_ℓ^* can be constructed using partial solutions as $w_1^* + w_2^*$. Now by induction we have:

$$\begin{aligned} w_\ell = & \quad (w_1 + w_2)/(1 + \epsilon/\log n) \\ \geq & \quad ((1 - \epsilon/\log n)^{\ell-1} w_1^* + (1 - \epsilon/\log n)^{\ell-1} w_2^*)/(1 + \epsilon/\log n) \\ \geq & \quad (1 - \epsilon/\log n)^\ell (w_1^* + w_2^*) \\ = & \quad (1 - \epsilon/\log n)^\ell w_\ell^* \end{aligned}$$

which completes the induction step. □

The above lemmas imply the following result.

Theorem 3 *There is an MPC algorithm computing a $(1 - \epsilon)$-approximate solution for weighted interval selection. The algorithm uses $\tilde{O}(n/m)$ memory per machine and runs in $O(\log m)$-rounds.*

To close, is there a guiding principle at work here? We started with a recurrence $A(i)$ which was increasing. We then expanded it into a recurrence $B(i, j)$ that captured more information about the problem about start/end positions. Then we performed the "swap" to define $C(i, w)$. Note that $C(i, w)$ should be increasing as well. We can then compute $C(i, w)$ locally and perform merging over several rounds. Using approximate weights will help to reduce the space and communication costs enough to fit into the MPC mode.

References

[1] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[2] MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1654–1672. SIAM, 2019.

[3] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 798–811. ACM, 2017.