Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 9: Oblivious Parallel RAMs (OPRAMs)

Scribe: Jeff Xu

May 1, 2024

1 Overview

We cover parallel computing today which has a growing intersection with cryptography lately for application of multi-party computation and zero-knowledge proof in the interplay with big data. Today we are going to focus on oblivious RAM which also has application in secure multiparty computation [CS16, CSLN20].

2 Oblivious Parallel RAM

Recall that in the earlier lectures we assumed a sequential model of computation and showed how to compile a RAM into an ORAM. However, we will start with a *Parallel* RAM program today. Even though cryptographers may also ask whether we can parallelize crypto by starting with a possibly insecure RAM, we actually focus on starting with an already Parallel RAM program here. However, the focus here would be how one may preserve parallelism in the oblivious component.

2.1 PRAM Model

In this model, imagine we have m CPUs/Processors with a shared array memory of size n.

Remark 1. Assume we have a sufficient source of parallelism such that $m = \omega(n)$, as otherwise, a smaller degree of parallelism may be translated into a sequential program with polylog blowup. The polylog overhead is usually not important, however, by some previous paper [link], the polylog factor can be tightened to match the sequential case.

Furthermore, we consider the following notations,

1. NI_i: the next instruction circuit for each CPU i;
2. $St_{i,t}$: the state of CPU i at time-stamp t;
3. $addr_{i,t}$ denote the address to visit of CPU i at time t.

And we further adopt the setting of CRCW-PRAM (concurrent-read-concurrent-write PRAM). The model is captured by the following,

1. At each time stamp, the CPU i reads its corresponding $NI_i(St_{i,t}, mem[addr_{i,t}])$;
2. Outputs the computed subsequent state $St_{i,t+1}$, $addr_{i,t+1}$ the address to visit next time, $data_t$ some data to write;

3. The PRAM writes the given data $data_t$ to the location just read, i.e., it sets $mem[addr_{i,t}] := data_t$.

To handle the confusion of concurrent-write, we first observe that the above model would function well if different CPUs all write to different addresses. However, clash may happen, and this can be handled via various refinements.

Definition 1 (Refinement via Arbitrary PRAM)**.** Resolve the write-conflict arbitrarily.

For algorithm designers, the conflict resolution cannot be predicted and the correctness of algorithm should be guaranteed regardless of how the conflicts are resolved.

Definition 2 (Refinement via Priority CRCW-PRAM)**.** Resolve write-conflict based on processor's ids (which indicate its priority), and ignore the non-prioritized write-requests.

It may be observed that arbitrary PRAM is a weaker model as any algorithm works in the arbitrary setting also works in the priority model.

Definition 3 (Oblivious PRAM)**.** An algorithm is Oblivious PRAM if for any input I_0, I_1 of some length,

$$\text{AccessPattern}(I_0) \approx_{Comp./Stats.} \text{AccessPattern}(I_1)$$

where the AccessPattern takes into account of all CPUs at all time-stamps, and each operation.

Remark 2. The definition is essentially the same for ORAM except we take into account of all devices.

3 (CRCW) PRAM to OPRAM: on a high-level

In this section, we see how to transfer a PRAM into an OPRAM. In fact, we will ignore the refinement rule above and the compilation applies for both of the above refinements.

We start by focusing on the non-recursive component of the algorithm and assume a position map at hand. On a high-level, we follow the path over RAM algorithm and essentially translate it for OPRAM. We first imagine that we are ignoring the initial steps as conflict are more likely to happen there when we have concurrent devices. However, later on in the process, the requests would get disjoint enough and we now view this as a collection of subtrees. At the moment, fix $\frac{m}{Z}$ the number of subtrees for some $Z = \omega(\log n)$ (for negligible failure probability) and m the number of requests.

By construction of PathORAM, each request follows a random path and we may now think of each request as being dropped into $\frac{m}{Z}$ subtrees. Observe that each tree has expected load Z, and enjoys nice concentration by Chernoff Bound. Wigh high probability, each subtree has load at most $2Z$. In other words, this gives a random partition of a single big ORAM into $\frac{m}{Z}$ smaller ORAMs. One may further parallelize within each ORAM but that may not be necessary.

However, we are closing our eyes on several technicalities here. First, we assume the requests are sufficiently disjoint. In the extreme case where multiple CPUs are all requesting the same memory address, the disjointness assumption is no longer valid and load-balance is no longer true. That said, one would hope to show that all these m requests are "independent", and they follow a path independent of other (which is not true if the addresses are the same). Towards this end, we introduce a conflict-resolution stage before we even start OPRAM and first preprocess the requests.

Pre-processing We first reveal the incoming m requests, and suppress the duplicate ones. This may give result in a shorter stream, and we then pack it with fake distinct queries.

Post-processing Obliviously route the requested address back to the requesting CPU.

4 Formalizing via Building Blocks

4.1 Oblivious Sort

It has both small work (total computation if performed on a sequential order) and small parallel runtime. Concretely, it can be done via AKS with work $O(n \log n)$ and parallel runtime $O(\log n)$.

4.2 Segmented Scan / Segmented Prefix-Sum

We first consider the classical version, where we are given an array A of n numbers, and we are to compute $\sum_{t=1}^{i} A[i]$ for each i. In the classical setting, it can be done using $O(\log n)$ depth by building a binary tree and join them in pairs. This can be done in work $O(n)$ and $O(\log n)$ depths.

The algorithm works in the recursive manner as the following,

Algorithm 1: Scan[A_i]

```
1. if index $i$ is even, let $C_i = A[i-1] \oplus A[i]$;
2. $b[i] = scan([c[i]])$ ($i$ even)
3. if index $i$ is odd: $b[i] = b[i-1] \oplus A[i]$;
4. return $[b[i]]$
```

Remark 3. In the classic setting of prefix sum, we instantiate $\oplus$ as the usual addition operator. However, it may be easily replaced for other functionalities.

Definition 4 (Segmented Prefix Sum)**.** Given an array A of n numbers, and an array C of n 0/1 bits with each 1 bit indicating start of a new segment, for each index, compute the prefix sum of its corresponding segment up until the index.

Definition 5 (Extended operator for segmented prefix-sum)**.** We instantiate $\oplus$ as the following operator. For any two numbers x, y, let c_x, c_y be their corresponding bit in C indicating whether they are the starts of a new segments. We define $\oplus$ on (x, b_x) and (y, b_y) as the following operation,

$\oplus$	$(y, 0)$	$(y, 1)$
$(x, 0)$	$(x+y, 0)$	$(y, 1)$
$(x, 1)$	$(x+y, 1)$	$(y, 1)$

Claim 6. The above algorithm computes the segmented prefix sum with work $O(n)$ and depth $O(\log n)$.

4.3 OPRAM Algorithm

1. Conflict Resolution
   Input: array of m requests
   Output: duplicate suppressed array of m arrays padded with filler requests
   This can be done via Oblivious Sorting as sorting by their addresses will bucket duplicate requests into a single group. Now it suffices for us to iterate through the elements, and for each group, remove all but one element).

2. Read and Remove: by concentration bound, we know each subtree receives at most $2Z$ requests with $1 - o_n(1)$ probability. For each subtree, read and remove for each non-filler request; otherwise, read a random path and return a filler element.

3. Take Pool P and union with all the fetched elements. Obliviously route from the pool to CPUs (via OSORT and Segmented Scan)

4. Write-back: each CPU writes back a block to the pool and use OSORT to duplicate suppress (when multiple CPUs have the same write-requests) and maintain the fresh copy for each address.

5. Eviction: pick $2Z$ deepest blocks (padded with fake blocks if fewer) from the pool, and route to each subtree (via OSORT)
   Each subtree performs eviction on $4Z$ random paths.

6. Pool clean-up (OSORT and truncate pool at fixed $C \cdot m$ length)

4.4 Oblivious Route

Assume we have the following input,

1. Source: an array of key-value pairs $(k_i, v_i)_i$;

2. Destination: $(t, \tilde{k}_t)$ destination t requesting for key $\tilde{k}_t$.

To perform this in parallel,

1. OSORT an array concatenated with Source and Destination by key, and assume for each key, the source entry comes first.

2. Segmented Prefix Scan: each source entry propagates the value to all destination entries that request for it (bucketed into a single group by OSORT)

3. Obliviously sort such that all destination entries come first, and remove the source array by truncating at length m.

4.5 Cost of OPRAM

Let α denote a quantity that subsumes some super-constant factor of n, for every batch of m requests, we have work as the following

1. Fetch: Each CPU performs the read phase of these $O(\alpha \log N)$ requests sequentially, which involves reading up to $O(\alpha \log N)$ paths in the tree. Since each path is $O(\log n)$ in length, we obtain a total-work bound of $O(\alpha \log^2 N)$.

2. Route: The oblivious routing step is simply $O(\log m)$ parallel steps.

3. Remap: Assign each fetched block to a random new subtree and a random leaf within that subtree. By concentration, each subtree has at most $O(\alpha \log N)$ blocks, and we then adopt an oblivious routing procedure to route these blocks back to each subtree, such that each tree receives blocks destined for itself together with padded dummy blocks. This incurs a bound of $O(\alpha \log m \log N)$ parallel steps.

4. Evict: Since each CPU performs $O(\alpha \log N)$ evictions on its own subtree, this is again in total $O(\alpha \log^2 N)$ parallel steps.

References

[CS16] T-H. Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. Cryptology ePrint Archive, Paper 2016/1084, 2016. `https://eprint.iacr.org/2016/1084`.

[CSLN20] T-H. Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) ram revisited, and improved constructions. Cryptology ePrint Archive, Paper 2020/604, 2020. `https://eprint.iacr.org/2020/604`.