Secure MPC Protocol for Randomized Shell Sort

April 16, 2019

Randomized shell sort is an oblivious sorting algorithm, developed by Michael Goodrich [1]. While the paper should be referred to for a full description of the algorithm and its analysis, the following is designed to be a general introduction to the algorithm, based on its more thorough description in Goodrich[1]. It additionally describes how it is implemented in JIFF in a secure manner.

1 Randomized Shellsort

Recall that insertion sort is an $O(n^2)$ sorting algorithm with the following (simplified) pseudocode:

Given input array A, let n be the length of A and let $B = A[0]$.
for i=1 to $n-1$ **do**
  Check each element of B against $A[i]$ and insert $A[j]$ into B so that the B preserves its order.
end for

The traditional Shellsort sorting algorithm relies on the idea that insertion sort performs better on partially sorted lists. (Note, for example, that shell sort on a fully sorted list will have $O(n)$ runtime since each element only needs to be compared to the adjacent element). A traditional Shellsort algorithm uses this idea, and does a series of insertion sort algorithms on subsets of the array, where each subset is chosen by taking every "o^{th}" element of the array (o stands for "offset"). In pseudocode, given array A indexed from 0 to $n-1$ and offset values $o_1, \ldots, o_p$ such that $o_i < n$,

for $i = 1$ to p **do**
  for $j = 0$ to $o_i - 1$ **do**
    Sort the subarray of A with indices $j, j + o_i, j + 2o_i, ...$ with insertion sort
  end for
end for

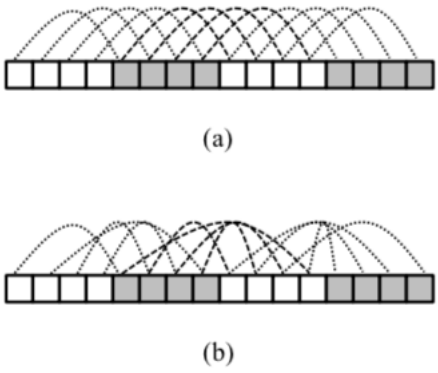

Figure 1: Compare-exchange operation. The dotted lines indicate the items that will be compared to one another, under (a) the identity permutation and (b) a random permutation for each pair of adjacent regions. Image from Goodrich [1].

Whether or not this is efficient depends on what series of offsets $o_1, \ldots, o_p$ is chosen. The idea behind *randomized* Shellsort is to fix the offset sequence as $O = n/2, n/4, n/8, \ldots, 1$, then compare pairs of regions A_1 and A_2 by randomly permuting the order and then doing comparisons and exchanges based on this random permutation, as shown in Fig. 1.

Which pairs of regions are to be compared are chosen based on a schedule as follows:

for $o = n/2, n/2^2, n/2^3, \ldots, 1$ **do**
  Let A_i denote subarray $A[io \,..\, io + o - 1]$, **for** $i = 0, 1, 2, \ldots, n/o - 1$.
  do a shaker pass:
    Region compare-exchange A_i and A_{i+1}, **for** $i = 0, 1, 2, \ldots, n/o - 2$.
    Region compare-exchange A_{i+1} and A_i, **for** $i = n/o - 2, \ldots, 2, 1, 0$.
  do an extended brick pass:
    Region compare-exchange A_i and A_{i+3}, **for** $i = 0, 1, 2, \ldots, n/o - 4$.
    Region compare-exchange A_i and A_{i+2}, **for** $i = 0, 1, 2, \ldots, n/o - 3$.
    Region compare-exchange A_i and A_{i+1}, **for** even $i = 0, 1, 2, \ldots, n/o - 2$.
    Region compare-exchange A_i and A_{i+1}, **for** odd $i = 0, 1, 2, \ldots, n/o - 2$.

Figure 2: Randomized Shellsort pseudocode from Goodrich [1].

As shown by Goodrich, this randomized Shellsort has, with high probability, a runtime of $O(n \log n)$ [1].

2 Our Secure Implementation

We securely implement randomized Shellsort on an array which is the element-wise sum of some number of input arrays. Since the algorithm is oblivious in the sense that everything other than the compare-exchange operation do not require knowledge of the underlying data, the majority of the algorithm may be performed without any special tools. Thus, the only portion of our implementation that requires use of the MPC library is the initial element-wise summation of the arrays and the compare-exchange operation.

The only other nuance to this implementation is in the randomization. Since the secret-sharing that underlies the compare-exchange operations occurs in parallel for all parties, all parties must do compare-exchanges on the same elements. In other words, the random permutations that the parties do must be fixed before the computations begin. In this demo, this is handled by having the first party send out a random string to all parties, who listen for that message. All parties then use this shared random string to determine the comparisons.

References

[1] Goodrich, M. T. Randomized shellsort: A simple oblivious sorting algorithm. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms* (2010), Society for Industrial and Applied Mathematics, pp. 1262–1277.