POLITECNICO
MILANO 1863

Constant Time, Parallel Shuffling

095947 - CRYPTOGRAPHY AND ARCHITECTURES FOR COMPUTER SECURITY

Conti Alessandro

Table of Contents

Specification

Shuffling arrays in constant time is a common problem in modern cryptography. This project involves analysing the technique proposed by Daniel J. Bernstein in `https://cr.yp.to/2024/insertionseries-20240515.py`, and implementing it in C, possibly employing parallelization.

Why Does It Matter?

In many cryptographic applications, it is necessary to perform multiple insertions within an array or list

- construction of constant-weight words used in the McEliece cryptosystem;
- insertion of a blockchain transaction in the mempool.

However, naive implementations is very slowly and may expose data to side-channel attacks. If memory access depends on data, the adversary may infer sensitive information.

Inserting an Element into an Array

To insert an element at a specific position in an array of size n, it is necessary:

- increase the size of the array if all positions are full, $\mathcal{C}(n) = \mathcal{O}(n)$;
- move all elements to the right of the specified position by one position, $\mathcal{C}(n) = \mathcal{O}(n)$;
- insert the new element at the desired, free position, $\mathcal{C}(n) = \mathcal{O}(1)$.

This algorithm has the following computational cost

$$\mathcal{C}_{insert}(n) = \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$$

Inserting Multiple Elements into an Array

To insert m elements in specific positions in an array of size n, it is necessary to repeat for m times the insertion of a single element.

This algorithm has the following computational cost

$$\mathcal{C}_{multiple\ \ insertions}(n, m) = m \cdot \mathcal{C}_{insert}(n) = \mathcal{O}(m \cdot n)$$

which is highly inefficient if most of the entries in an array are multiple entries and not single entries.

Insertion Series Objective

We will now analyse an algorithm that attempts to optimise serial insertion within an array. Trying to go from having a quadratic complexity to a quasi-linear one.

insertionseries_merge_after_sort_recursive Function


```
def insertionseries_merge_after_sort_recursive(L, XY):
    L = list(enumerate(L))
    R = insertionseries_sort_recursive(XY)
    return [y for x, y in insertionseries_sort_merge(L, R)]
```

insertionseries_merge_after_sort_recursive Cost

Let $n :=$ `list->listSize` and $m :=$ `pairList->listSize`

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{pairlist\_init}(n) + n \cdot \mathcal{C}_{pairlist\_append}(n) + \mathcal{C}_{insertionseries\_sort\_recursive}(m) + \\
&\quad + \mathcal{C}_{insertionseries\_sort\_merge}(n, m) + \mathcal{C}_{intlist\_init}(n+m) + \mathcal{C}_{intlist\_reserve}(n+m) + \\
&\quad + m \cdot \mathcal{C}_{intlist\_append}(m) + \mathcal{C}_{pairlist\_free}(n) + \mathcal{C}_{pairlist\_free}(m) + \mathcal{C}_{pairlist\_free}(n+m) \\
&= 1 + n \cdot \log(n+1) + \mathcal{O}\left(m \cdot (\log(m))^2\right) + \mathcal{O}((n+m) \cdot \log(n+m)) + 1 + \\
&\quad + \log(n+m) + m \cdot \log(m+1) + 1 + 1 + 1 = \\
&= \mathcal{O}\left(m \cdot (\log(m))^2\right) + \mathcal{O}((n+m) \cdot \log(n+m)) + \\
&\quad + \log(n+m) + n \cdot \log(n+1) + m \cdot \log(m+1) + 5 = \\
&= \mathcal{O}\left((n+m) \cdot \log(n+m) + m \cdot (\log(m))^2\right)
\end{aligned}
$$

insertionseries_merge_after_sort_recursive Cost (2)

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{Pairlist}(n) + \mathcal{S}_{Pairlist}(m) + \mathcal{S}_{Pairlist}(n+m) + \mathcal{S}_{IntList}(n+m) = \\
&= \ (n \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + (m \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + \\
&\quad + ((n+m) \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 20\,\mathrm{B} \cdot (n+m) + 64\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

insertionseries_sort_recursive Function


```
def insertionseries_sort_recursive(XY):
    XY = list(XY)
    t = len(XY)
    if t <= 1: return XY
    s = t // 2
    L = insertionseries_sort_recursive(XY[:s])
    R = insertionseries_sort_recursive(XY[s:])
    return insertionseries_sort_merge(L, R)
```

insertionseries_sort_recursive Cost

Let m := `pairList->listSize`

$$\mathcal{C}(m) = \begin{cases} \mathcal{C}_{pairlist\_copy}(m) & m \leq 1 \\ 2 \cdot \left(\mathcal{C}_{pairlist\_init}(m/2) + \mathcal{C}_{pairlist\_reserve}(m/2)\right) + m \cdot \mathcal{C}_{pairlist\_append}(m/2) + & \\ \quad +2 \cdot \mathcal{C}(m/2) + \mathcal{C}_{insertionseries\_sort\_merge}(m/2, m/2) + & \\ \quad +2 \cdot \mathcal{C}_{pairlist\_free}(m/2) & m > 1 \end{cases}$$

$$= \begin{cases} m & m \leq 1 \\ 2 \cdot (1 + \log(m/2)) + m \cdot \log(m/2 + 1) + 2 \cdot \mathcal{C}(m/2) + & \\ \quad +\mathcal{C}_{insertionseries\_sort\_merge}(m/2, m/2) + 2 \cdot 1 & m > 1 \end{cases}$$

$$= \begin{cases} m & m \leq 1 \\ 2 \cdot \mathcal{C}(m/2) + 2 \cdot \log(m/2) + m \cdot \log(m/2 + 1) + & \\ \quad +\mathcal{C}_{insertionseries\_sort\_merge}(m/2, m/2) + 4 & m > 1 \end{cases}$$

What is currently shown is the time complexity of an iteration; to find the total complexity, the master theorem must be applied. On the next slide, we see the complexity by considering only the big-O time complexity of the insertionseries_sort_merge function.

insertionseries_sort_recursive Cost (2)

$$\begin{aligned}
\mathcal{C}(m) &= 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + 2 \cdot \log\left(m/2\right) + m \cdot \log\left(m/2 + 1\right) + \\
&\quad + \mathcal{O}\left(\left(\frac{m}{2} + \frac{m}{2}\right) \cdot \log\left(\frac{m}{2} + \frac{m}{2}\right)\right) = \\
&= 2 \cdot \mathcal{C}\left(\frac{m}{2}\right) + \mathcal{O}(m \cdot \log(m)) = \\
&= \mathcal{O}\left(m \cdot (\log(m))^2\right)
\end{aligned}$$

insertionseries_sort_recursive Cost (3)

$$\mathcal{S}_{iteration}(m) = \begin{cases} \mathcal{S}_{PairList}(m) & m \leq 1 \\ 2 \cdot \mathcal{S}_{size\_t} + 2 \cdot \mathcal{S}_{PairList}(m/2) + \mathcal{S}_{PairList}(m) & m > 1 \end{cases}$$

$$= \begin{cases} 8\,\mathrm{B} \cdot m + 16\,\mathrm{B} & m \leq 1 \\ 2 \cdot 8\,\mathrm{B} + 2 \cdot (8\,\mathrm{B} \cdot m/2 + 16\,\mathrm{B}) + (8\,\mathrm{B} \cdot m + 16\,\mathrm{B}) & m > 1 \end{cases}$$

$$= \begin{cases} 8\,\mathrm{B} \cdot m + 16\,\mathrm{B} & m \leq 1 \\ 16\,\mathrm{B} \cdot m + 64\,\mathrm{B} & m > 1 \end{cases}$$

$$\mathcal{S}(m) = \sum_{i=0}^{\log m} \mathcal{S}_{iteration}\left(\frac{m}{2^i}\right) =$$

$$= 16\,\mathrm{B} \cdot m \cdot \sum_{i=0}^{\log m} \frac{1}{2^i} + 64\,\mathrm{B} \cdot \sum_{i=0}^{\log m} 1 =$$

$$< 16\,\mathrm{B} \cdot m \cdot \left(2 - \frac{1}{m}\right) + 64\,\mathrm{B} \cdot (\log m + 1) =$$

$$= 32\,\mathrm{B} \cdot m + 64\,\mathrm{B} \cdot \log(m) + 48\,\mathrm{B}$$

$$= \mathcal{O}(m)$$

insertionseries_sort_merge Function


```python
def insertionseries_sort_merge(L, R):
    L = [(x, 1, 0, y) for j, (x, y) in enumerate(L)]
    R = [(x - j, 0, j, y) for j, (x, y) in enumerate(R)]
    M = merge(L, R)
    offsets = prefixsums(1 - fromL for _, fromL, _, _ in M)
    return [(x + offset, y) for (x, _, _, y), offset in zip(M,
    ↪  offsets)]
```

insertionseries_sort_merge Cost

Let n := `firstList->listSize` and m := `secondList->listSize`

$$\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{malloc}(n) + \mathcal{C}_{malloc}(m) + n + m + \mathcal{C}_{merge}(n, m) + \mathcal{C}_{intlist_init}(n + m) + \\
&+ \mathcal{C}_{intlist_reserve}(n + m) + (n + m) \cdot \mathcal{C}_{intlist_append}(n + m) + \\
&+ \mathcal{C}_{prefixSum}(n + m) + \mathcal{C}_{pairlist_init}(n + m) + \mathcal{C}_{pairlist_reserve}(n + m) + \\
&+ (n + m) \cdot \mathcal{C}_{pairlist_append}(n + m) + \mathcal{C}_{free}(n) + \mathcal{C}_{free}(m) + \mathcal{C}_{free}(n + m) + \\
&+ \mathcal{C}_{intlist_free}(n + m) + \mathcal{C}_{intlist_free}(n + m + 1) = \\
&= 1 + 1 + n + m + \mathcal{C}_{merge}(n, m) + 1 + \log(n + m) + (n + m) \cdot \log(n + m + 1) + \\
&+ \mathcal{C}_{prefixSum}(n + m) + 1 + \log(n + m) + (n + m) \cdot \log(n + m + 1) + \\
&+ 1 + 1 + 1 + 1 + 1 = \\
&= 9 + n + m + 2 \cdot \log(n + m) + (n + m) \cdot \log(n + m + 1) + \\
&+ \mathcal{C}_{merge}(n, m) + \mathcal{C}_{prefixSum}(n + m)
\end{aligned}$$

This is the complexity of the function, but the function calls two functions that can be parallelised. On the next slide, we see the two complexities in the case of choosing either not to parallelise or to parallelise.

insertionseries_sort_merge Cost (2)

$$
\begin{aligned}
\mathcal{C}_{serial}(n, m) &= 9 + n + m + 2 \cdot \log(n+m) + (n+m) \cdot \log(n+m+1) + \\
&\quad + (n + m + (n+m) \cdot \log(n+m) + 1) + \\
&\quad + ((n+m+2) \cdot \log(n+m+1) + 1) = \\
&= 2n + 2m + (n+m+2) \cdot \log(n+m) + (2n+2m+2) \cdot \log(n+m+1) + 11 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_{parallel}(n, m, t) &= 9 + n + m + 2 \cdot \log(n+m) + (n+m) \cdot \log(n+m+1) + \\
&\quad + \left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t} + 1 \right) + \\
&\quad + \left(2 \cdot \frac{n}{t} + n + t + \log(n+1) + 7 \right) = \\
&= \left(2 + \frac{2}{t} \right) \cdot n + m + t + \log(n+1) + \log\left(n^{m/t} \cdot m^{n/t} \right) + 2 \cdot \log(n+m) + \\
&\quad + (n+m) \cdot \log(n+m+1) + 17 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

insertionseries_sort_merge Cost (3)

$$
\begin{aligned}
\mathcal{S}(n, m) &= 3 \cdot \mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n) + \mathcal{S}_{*Quadruple}(m) + \mathcal{S}_{*Quadruple}(n+m) + \\
&\quad + \mathcal{S}_{IntList}(n+m) + \mathcal{S}_{IntList}(n+m+1) + \mathcal{S}_{PairList}(n+m) = \\
&= 3 \cdot 8\,\mathrm{B} + n \cdot 16\,\mathrm{B} + m \cdot 16\,\mathrm{B} + (n+m) \cdot 16\,\mathrm{B} + \\
&\quad + ((n+m) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + ((n+m+1) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + \\
&\quad + ((n+m) \cdot 8\,\mathrm{B} + 16\,\mathrm{B}) = \\
&= 48\,\mathrm{B} \cdot (n+m) + 76\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

merge Function


```python
def merge(L, R):
    return sorted(L + R)
```

merge Cost

Let $n :=$ `firstListSize` and $m :=$ `secondListSize`.

$$
\begin{aligned}
\mathcal{C}(n, m) &= \mathcal{C}_{malloc}(n+m) + \mathcal{C}_{memcpy}(n) + \mathcal{C}_{memcpy}(m) + \mathcal{C}_{qsort}(n+m) = \\
&= 1 + n + m + (n+m) \cdot \log(n+m) = \\
&= (n+m) + (n+m) \cdot \log(n+m) + 1 = \\
&= \mathcal{O}((n+m) \cdot \log(n+m))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}(n, m) &= \mathcal{S}_{size\_t} + \mathcal{S}_{*Quadruple}(n+m) = \\
&= 8\,\mathrm{B} + (n+m) \cdot 16\,\mathrm{B} = \\
&= 16\,\mathrm{B} \cdot (n+m) + 8\,\mathrm{B} = \\
&= \mathcal{O}(n+m)
\end{aligned}
$$

mergeParallel Function


```c
Quadruple *mergeParallel(Quadruple *firstList, size_t firstListSize, Quadruple *secondList, size_t secondListSize) {
    size_t resultSize = firstListSize + secondListSize;
    Quadruple *result = malloc(resultSize * sizeof(Quadruple));
    if (!result)
        return NULL;

#pragma omp parallel
    {
#pragma omp for schedule(static) nowait
        for (size_t i = 0; i < firstListSize; i++) {
            result[i + binarySearch(secondList, 0, secondListSize, firstList[i])] = firstList[i];
        }

#pragma omp for schedule(static)
        for (size_t i = 0; i < secondListSize; i++) {
            result[i + binarySearch(firstList, 0, firstListSize, secondList[i])] = secondList[i];
        }
    }

    return result;
}
```

mergeParallel Cost

Let n := `firstListSize`, m := `secondListSize` and t := number of threads used.

$$\begin{aligned}
\mathcal{C}(n, m, t) &= \mathcal{C}_{malloc}(n+m) + \frac{n}{t} \cdot \mathcal{C}_{binarySearch}(m) + \frac{m}{t} \cdot \mathcal{C}_{binarySearch}(n) = \\
&= 1 + \frac{n}{t} \cdot \log(m) + \frac{m}{t} \cdot \log(n) = \\
&= \frac{n \cdot \log(m) + m \cdot \log(n)}{t} + 1 = \\
&= \mathcal{O}\left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t}\right)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(n, m, t) &= \mathcal{S}_{*Quadruple}(n+m) = \\
&= (n+m) \cdot 16\,\mathrm{B} = \\
&= 16\,\mathrm{B} \cdot (n+m) = \\
&= \mathcal{O}(n+m)
\end{aligned}$$

prefixsums Function


```
def prefixsums(L):
    result = [0]
    for b in L:
        result += [result[-1] + b]
    return result
```

prefixsums Cost

Let $n :=$ `list->listSize`.

$$\begin{aligned}
\mathcal{C}(n) &= \mathcal{C}_{intlist_init}(n+1) + \mathcal{C}_{intlist_reserve}(n+1) + (n+1)\cdot\mathcal{C}_{intlist_append}(n+1) = \\
&= 1 + \log(n+1) + (n+1)\cdot\log(n+1) = \\
&= (n+2)\cdot\log(n+1) + 1 \\
&= \mathcal{O}(n\cdot\log n)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(n) &= \mathcal{S}_{IntList}(n+1) + \mathcal{S}_{int} = \\
&= ((n+1)\cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + 4\,\mathrm{B} = \\
&= 4\,\mathrm{B}\cdot n + 32\,\mathrm{B} = \\
&= \mathcal{O}(n)
\end{aligned}$$

prefixSumParallel Function


```c
IntList prefixSumParallel(const IntList *list) {
    IntList result;
    intlist_init(&result);
    intlist_reserve(&result, list->listSize + 1);

    size_t listSize = list->listSize;
    int *output = malloc((listSize + 1) * sizeof(int));
    output[0] = 0;

    int numberThreadUsed = 0;
    int *partialSumList = NULL;
#pragma omp parallel
    {
        int threadID = omp_get_thread_num();
        int numberThread = omp_get_num_threads();
#pragma omp single
        {
            numberThreadUsed = numberThread;
            partialSumList = calloc(numberThreadUsed, sizeof(int));
        }
        size_t chunk = (listSize + numberThread - 1) / numberThread;
        size_t start = threadID * chunk;
        size_t end = (start + chunk < listSize) ? start + chunk : listSize;

        int localSum = 0;
        for (size_t i = start; i < end; ++i) {
            localSum += list->list[i];
            output[i + 1] = localSum;
        }
        partialSumList[threadID] = localSum;
#pragma omp barrier
        int offset = 0;
        for (int i = 0; i < threadID; ++i)
            offset += partialSumList[i];
        for (size_t i = start + 1; i <= end; ++i)
            output[i] += offset;
    }
    for (size_t i = 0; i <= listSize; ++i)
        intlist_append(&result, output[i]);
    free(output);
    free(partialSumList);
    return result;
}
```

prefixSumParallel Cost

Let $n :=$ `list->listSize` and $t :=$ number of threads used.

$$\begin{aligned}
\mathcal{C}(n,t) &= \mathcal{C}_{intlist_init}(n+1) + \mathcal{C}_{intlist_reserve}(n+1) + \mathcal{C}_{malloc}(n+1) + \mathcal{C}_{omp_get_thread_num}(t) + \\
&\quad + \mathcal{C}_{omp_get_num_threads}(t) + \mathcal{C}_{calloc}(t) + 2 \cdot \frac{n}{t} + t + n + \mathcal{C}_{free}(n+1) + \mathcal{C}_{free}(t) = \\
&= 1 + \log(n+1) + 1 + 1 + 1 + 1 + 2 \cdot \frac{n}{t} + t + n + 1 + 1 = \\
&= 2 \cdot \frac{n}{t} + n + t + \log(n+1) + 7 = \\
&= \mathcal{O}(n)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(n,t) &= \mathcal{S}_{IntList}(n+1) + 3 \cdot \mathcal{S}_{size_t} + \mathcal{S}_{*int}(n+1) + 5 \cdot \mathcal{S}_{int} + \mathcal{S}_{*int}(t) = \\
&= ((n+1) \cdot 4\,\mathrm{B} + 16\,\mathrm{B}) + 3 \cdot 8\,\mathrm{B} + (n+1) \cdot 4\,\mathrm{B} + 5 \cdot 4 + t \cdot 4\,\mathrm{B} = \\
&= 8\,\mathrm{B} \cdot n + 4\,\mathrm{B} \cdot t + 68\,\mathrm{B} = \\
&= \mathcal{O}(n+t)
\end{aligned}$$

Comparison Between Naive and DJB Versions

Let $n :=$ size of the array into which new elements are to be inserted and $m :=$ list of elements to be inserted.

$$\mathcal{C}_{naive}(n, m) = \mathcal{O}(m \cdot n)$$
$$\mathcal{S}_{naive}(n, m) = \mathcal{O}(m \cdot n)$$

$$\mathcal{C}_{DJB}(n, m) = \mathcal{O}\Big((n + m) \cdot \log{(n + m)} + m \cdot (\log{(m)})^2\Big)$$
$$\mathcal{S}_{DJB}(n, m) = \mathcal{O}(n + m)$$

Thus for large sizes of n, m the algorithm devised by DJB speeds up the insertion of multiples of elements into an array at the same time.
Obviously if you only want to insert only a single element, the best algorithm is the naive one which has $\mathcal{O}(n)$ complexity and not DJB which has $\mathcal{O}(n \cdot \log{(n)})$ instead.

Using Parallelisation

In order to try to gain some computation time, it is possible to use the technique of parallelisation.
In this algorithm, the functions that can be parallelised are essentially the *merge* and the *prefixSum*. Parallelising these two functions resulted in an improvement:

- *prefixSum* went from having complexity $\mathcal{O}(n \cdot \log(n))$ to $\mathcal{O}(n)$, so a change from quasi-linear to linear complexity was achieved, and a very good asymptotic improvement was also obtained;
- *merge* function with the parallelisation has at least the same complexity as without the parallelisation, this similarity is due to the fact that they both have similar components, but for $t > 1$ there is a gain proportional to the number of threads. The two *merge* complexities are $\mathcal{O}((n+m) \cdot \log(n+m))$ and $\mathcal{O}\left(\frac{n \cdot \log(m) + m \cdot \log(n)}{t}\right)$.