Theory and Practice of Chunked Sequences

Umut Acar       Arthur Charguéraud       Mike Rainey
Inria & Carnegie Mellon University
Goal

Ephemeral deques (double-ended queues) with:
→ push and pop at the two ends, in $O(1)$ amortized
→ concat, and split at arbitrary indices, in $O(\log n)$

Weighted split operation: (e.g. split at 13)
Motivation

Application to parallel BFS and parallel DFS

→ each processor push/pop vertices from its frontier
→ split is used for dynamically distributing the load
→ concatenation is used to merge the new frontiers

Requirements:

→ push and pop must be very efficient
→ split and concatenation is sublinear time
Contribution

Amortized $O(1)$ push/pop and $O(\log N)$ concat/split

Prior work:

$\rightarrow$ Kaplan and Tarjan (1996)
$\rightarrow$ Hinze and Parterson (2006)

Purely functional data structures (confluently persistent). Yet, very large constant factors (even if made ephemeral).

This work: ephemeral catenable/splittable deques with small constant factors (e.g., not far from C++ STL deques).
Overview

(1) Chunked sequences
   → assume a potentially-slow deque data structure
   → construct a fast deque data structure, using chunks

   • amortize allocations in worst-case push-pop scenarios
   • ensure space efficiency in worst-case concat scenarios

(2) Bootstrapped chunked sequences
   → build a stand-alone, fast, catenable/splittable deque
   → use structural decomposition and recursive slowdown
      (Dietz 1982; Buchsbaum & Tarjan 1995; Kaplan & Tarjan 1996)
Challenges with chunks

A chunk = fixed-capacity ring buffer (repr. as an array)

A deque of chunks (e.g. C++ STL deques)

Challenge: (iterated push/pop operations)
Our chunked sequence

**Approach:** place two special chunks on each side

```
  front  front  middle sequence  back  back
outer  inner   of chunks    inner  outer
```

**Invariant:** the inner chunks must be either empty or full.
Implementation of push and pop

1. **push**
   - Insertion of elements into the stack.

2. **push***
   - Attempted insertion with failure.

3. **push**
   - Successful insertion after the attempt.

4. **pop**
   - Removal of elements from the stack.

5. **(fix)**
   - Rectification of errors in the stack operations.

6. **pop**
   - Continued successful removal of elements.

7. **pop***
   - Final successful removal of elements.

The diagram illustrates the implementation of push and pop operations on a stack, showing the sequence of actions and their outcomes.
Amortization with chunked sequences

**Theorem:** the amortized cost of push (including pop) is

\[ C + \frac{A+M}{K} + O(1) \]

where:

- **C** cost of push (including associated pop) in a chunk
- **A** cost of allocation (including deallocation) of a chunk
- **M** cost of push (including pop) in the middle sequence
- **K** capacity of a chunk
Challenges with concatenation

Concatenation of deques of chunks (with merge)

Worst-case scenario:
Implementation of concatenation

**Invariant:** any two consecutive chunks in the middle sequence must store a total number of more than $K$ items.

**Concatenation:** (up to 4 chunks need to be merged)
Implementation of split

Case 1:

Case 2:
**Towards bootstrapping**

**Summary:** given a potentially-slow catenable/splittable deque, we built a catenable/splittable deque structure with small constant factors, even in worst-case scenarios.

**Next step:** implement the middle sequence of our chunked sequence using... our chunked sequence. Do so recursively.
Bootstraped chunked sequence

_depth 0_
- front outer
- front inner
- back inner
- back outer

_depth 1_

_depth 2_

Efficiency analysis

**Theorem:** the depth is at most

$$\left\lfloor \log_{(K+1)/2} n \right\rfloor + 1$$

**Remark:** depth is bounded by 7 for all practical purposes.

**Theorem:** push/pop has cost $O(1)$, with a small constant

**Theorem:** concat and split have cost

$$O \left( K * \log_{(K+1)/2} \left( \min \left( n_1, n_2 \right) \right) \right)$$

where $n_1$ and $n_2$ denote the size of the two parts involved.

→ compare with:  $$O \left( \log_2 \left( \min \left( n_1, n_2 \right) \right) \right)$$
Space-usage analysis

**Theorem:** asymptotic space usage is

\[
\left( 2 + \frac{O(1)}{K} \right) \ast n
\]

**Alternative:** with concat twice slower, density is 3/4, thus

\[
\left( 1.33 + \frac{O(1)}{K} \right) \ast n
\]

**Alternative:** for bag semantics (unordered items), usage is

\[
\left( 1 + \frac{O(1)}{K} \right) \ast n
\]
Implementation and benchmarks

Two implementations:
→ OCaml (mechanized proof using CFML, in Coq)
→ C++ (carefully optimized code)

Performance of C++ code:
→ first layer with unweighted chunks of capacity 512
→ deeper layers with weighted chunks of capacity 32

<table>
<thead>
<tr>
<th>Experiment</th>
<th>STL deque</th>
<th>Bootstr. chunked</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFO ((10^6 \times 10^3))</td>
<td>5.46</td>
<td>+28%</td>
</tr>
<tr>
<td>LIFO ((10^3 \times 10^6))</td>
<td>9.15</td>
<td>+20%</td>
</tr>
<tr>
<td>LIFO ((10^0 \times 10^9))</td>
<td>12.07</td>
<td>+12%</td>
</tr>
<tr>
<td>FIFO ((10^6 \times 10^3))</td>
<td>5.51</td>
<td>+16%</td>
</tr>
<tr>
<td>FIFO ((10^3 \times 10^6))</td>
<td>9.16</td>
<td>+15%</td>
</tr>
<tr>
<td>FIFO ((10^0 \times 10^9))</td>
<td>12.32</td>
<td>+8%</td>
</tr>
</tbody>
</table>
Thanks!