Theory and Practice of Chunked Sequences

Umut A. Acar[2,1], Arthur Charguéraud[1,3], and Mike Rainey[1]

[1] Inria
[2] Carnegie Mellon University
[3] LRI, Université Paris Sud, CNRS

Abstract. Sequence data structures, i.e., data structures that provide operations on an ordered set of items, are heavily used by many applications. For sequence data structures to be efficient in practice, it is important to amortize expensive data-structural operations by *chunking* a relatively small, constant number of items together, and representing them by using a simple but fast (at least in the small scale) sequence data structure, such as an array or a ring buffer. In this paper, we present chunking techniques, one direct and one based on bootstrapping, that can reduce the practical overheads of sophisticated sequence data structures, such as finger trees, making them competitive in practice with special-purpose data structures. We prove amortized bounds showing that our chunking techniques reduce runtime by amortizing expensive operations over a user-defined chunk-capacity parameter. We implement our techniques and show that they perform well in practice by conducting an empirical evaluation. Our evaluation features comparisons with other carefully engineered and optimized implementations.

1 Introduction

Sequence data structures, i.e., data structures that store an ordered set of elements and support operations on them, are fundamental in computer science. There exist several variants of sequences, such as LIFO queues (stacks), FIFO queues, doubly-ended queues (deques), and more general data structures, such as finger-search trees. The common operations on sequences include push and pop operations at one or two ends, a split operation that partitions the data structure at a desired position, and a concatenation operation that joins two sequences.

Many asymptotically efficient data structures for sequences have been developed. Resizable circular arrays support constant-time push, pop and random access operations, but require linear time for concatenation and splitting. Doubly-linked lists improve the bound for concatenation to $O(1)$, but splitting at a given index requires linear time. More sophisticated data structures, such as Kaplan and Tarjan's functional catenable sorted lists, support push and pop operations in constant time, while also supporting splitting and concatenation in logarithmic time [15]. Their catenable sorted list is one instance of a finger search tree, a type of tree that has been studied extensively since the 1970s [9]. A

more recent functional finger-tree data structure by Hinze and Paterson achieves similar bounds and accepts a simple implementation [11].

Practical performance is a major concern for sequence data structures because of their widespread use in applications. While there has been much focus on developing asymptotically efficient sequence data structures, there is relatively little rigorous work on practical data structures that can guarantee small constant factors on modern computers. To understand the significance of practical concerns we implemented in C++ an optimized version of Hinze and Paterson finger tree data structure [11], and compared it to a resizable circular array, which is simpler but asymptotically efficient only for a narrower set of operations including push and pop. Our experiments show that the finger tree is over 20 times slower for push/pop operations than with circular arrays.[4] This is unfortunate, because such gaps in performance can prevent the use of these asymptotically efficient data structures in practice. It would be nice to have the best of both worlds by guaranteeing both theoretical and practical efficiency. We are therefore interested in the question: *can we design asymptotically and practically efficient data structures for sequences that can support a broad range of operations, including push/pop operations on both ends, concatenation, and split at a specified position?*

In practice, simpler data structures can out-class sophisticated, asymptotically efficient data structures because the latter tends to perform many more expensive operations, such as memory operations and manipulations of tree nodes, than their simpler counterparts. To reduce such overheads, practitioners represent a sequence data structure as a hierarchical data structure consisting of an *underlying sequence* data structure that stores *chunks* of items instead of individual items. Each chunk in turn is represented as an array, which is basically a fast sequence data structure (in small scale). The idea is to amortize the cost of expensive memory operations on the underlying sequence over the items in the chunks. This chunking technique can be applied to essentially any underlying sequence data structure. For example, the C++ Standard Template Library (STL) [19] includes a deque data structure represented as a resizable circular array of chunks of 512 single-word items. Similarly, the Haskell "yi" package [2] provides a chunked finger-tree data structure for character sequences. While chunking can be effective in practice, all applications of this technique known to us are merely heuristics: they provide no worst-case efficiency guarantees. In fact, as we describe in Section 2, their time and space efficiency can degenerate significantly on certain sequences of operations.

In this paper, we give chunking algorithms that yield tight amortized, worst-case bounds with small constant factors. To support splits and random accesses efficiently, we consider a slightly more general interface for sequences: we associate weights with the items and support a *weighted-split* operation. The weighted-split operation takes a sequence s and a weight w, and it decomposes s in three parts (s_1, x, s_2), in such a way that $|s_1| \leq w < |s_1| + |x|$, where $|x|$

[4] We specifically measured the time for pushing 100 million integers and then popping them in FIFO order.

denotes the weight of the item x, and $|s_1|$ denotes the sum of the weights of the items in s_1.

Given any underlying weighted sequence data structure, we show in Section 3 how to build a weighted (or unweighted) sequence data structure by using K-capacity chunks

- that guarantees constant-time push and pop operations with excellent constant factors, in particular such that every allocation operation is amortized over at least K push or pop operations,
- that supports concatenation and split efficiently by introducing an additive overhead proportional to K, and
- that requires approximately a factor-2 increase in space usage, thus ensuring reasonably good space utilization.

At a high level, our techniques speed up the push and pop operations (usually the most common operations) without significantly affecting the performance for the other operations. We note that in this paper, we consider ephemeral (as opposed to persistent) data structures only.

Since our techniques can be applied to any sequence data structure, including to a chunked sequence data structure, it can be applied recursively to permit bootstrapping. We describe such a bootstrapped data structure in Section 4, which uses structural decomposition [7, 5, 4] and recursive slowdown [14].

In our proofs, in addition to considering the chunk size as a parameter, we also differentiate between memory allocation and other operations. For memory allocation, we introduce a parameter A to denote the cost of allocating and later deallocating a structure of bounded-size (e.g., a chunk or a record), and reserve the $O(1)$ notation to account for the other (relatively cheaper) operations. We show that all allocation operations are well amortized. As we describe briefly, in our chunking technique, allocation correlates with other expensive memory operations. This approach thus gives us a good indication of practical overheads.

To understand the actual practical efficiency of our proposed techniques, we have implemented them all in C++. We perform an empirical evaluation by comparing our data structures to more specialized data structures that are optimized for a narrower set of operations such as STL deques and ropes, which are carefully engineered and highly optimized. Our practical results confirm our theoretical results showing that our data structures perform well in practice, usually within 10% of the actual run time of the best known data structure, while still supporting a broader set of operations.

The contributions of the paper include the chunking techniques that guarantee worst-case bounds, their analysing and the proofs, the bootstrapped data structure, and the implementation and its evaluation. Our implementations and test scripts are available for download at `http://deepsea.inria.fr/chunkedseq/`.

2 Challenges

We consider common chunking strategies used in prior implementations such as those employed by the Standard Template Library for C++ and identify two

limitations that can lead to significantly degraded performance and underutilization of memory (space) by breaking the amortization benefits of chunking.

Push-pop sequences. A common chunking strategy is to create and dispose of chunks on a need by need basis. For example, to push an item x to the front of a sequence, we first check if there is space in the first chunk. If so, we push x into that chunk. Otherwise, we create a new chunk, place x in it, and push this chunk to the front. Symmetrically, to pop an item from the front, we extract the first item stored in the first chunk. If the first chunk becomes empty as a result, then we pop the chunk from the front and dispose of it.

This strategy can fail to amortize the cost of push/pop operations on chunks, which are expensive. For example, starting from a sequence whose front chunk is full, repeat the following pattern: push one item and pop it immediately. It is not difficult to see that each operation requires pushing/popping a chunk. This chunking strategy, employed by the C++ Standard Template Library (STL) Deques, runs 10 times slower in the worst case. To test this, we wrote a program that starting from an initial deque obtained by pushing a given number of items, performs a sequence of push and pop operations on 64-bit integers. The programs runs 10 times slower when the initial deque has a size equal to 511 modulo 512 than with a different initial deque. All chunked sequence data structures that we have seen (and their naive variants) suffer from the same or similar problems.

Sparse chunks. Chunking delivers efficiency improvements by amortizing the cost of slow operations over a number of fast operations. Such amortization works, of course, only if chunks are densely populated. When chunks are sparsely populated, then the amortization arguments breaks and performance and memory utilization drops. For example, if chunks have capacity K but store only 1 item, then amortization fails entirely and the memory footprint of the sequence is roughly K times bigger than necessary. It is not difficult to create sparse chunks by using concatenation operations. Consider for example a chunked sequence consisting of 2 chunks each containing a single item. Such a sequence can be obtained by pushing $K + 1$ items to the front, then popping $K - 1$ from the back, where K is the capacity of a chunk. Once we have two sequences each with two sparse chunks, we can create one with arbitrary number of chunks by repeatedly concatenating them.

The "yi" package of Haskell [2] implements a refinement of this strategy: to concatenate two sequences s_1 and s_2, first check whether the back chunk in s_1 and the front chunk in s_2 would fit into a single chunk; if so, merge these two chunks before concatenating the underlying sequences. This strategy does not prevent sparse chunks. For example, the concatenation of two sequences each made of two chunks of size 1 produces a sequence made of three chunks of size 1, 2, and 1. Concatenating two such sequences produces a sequence made of chunks of size 1, 2, 2, 2, and 1. By iterating the process, we obtain an arbitrarily-long sequence made of sparse chunks containing no more than 2 items each. This example demonstrates that a provably efficient chunking strategy requires techniques to prevent sparse chunks from being formed.

3 Efficient chunked sequences

One of our main results is a theorem (Theorem 1 below) that shows that chunking can be applied to any (underlying) sequence data structure. The theorem states the bounds for the resulting chunked sequence, parametrized by the bounds of the underlying sequence. To simplify the analysis, we combine the cost of push and pop. More precisely, we charge all the cost of a pop operation to the push operation associated with the corresponding item. Doing so is correct because we consider ephemeral sequences and conduct an amortized analysis. For the theorem, we define a *chunk* as a circular array of fixed capacity K and we assume that cost function for the underlying sequence (e.g., $\mathcal{C}_{split}(n)$) are nondecreasing functions of size.

Theorem 1 (Efficiency of chunked sequence). *Consider an underlying weighted sequence that supports the following operations:*

- *Push and pop, with cost $\mathcal{C}_{pushpop}$. For simplicity, we assume this cost to not depend on the number of items in the sequence.*
- *Concatenation, with cost $\mathcal{C}_{concat}(n)$, where n is the minimum of the sizes of the two input sequences.*
- *Weighted split, with cost $\mathcal{C}_{split}(n)$, where n is the minimum of the sizes of the two output sequences.*
- *Space usage bounded by $\mathcal{C}_{space}(n)$, where n is the number of single-word items stored in the sequence.*

Let $K \geq 2$ denote the capacity of a chunk, a value that may be freely chosen. Assume that chunks are implemented with a structure that supports $O(1)$ push and pop operations and that requires $K + 3$ words to store K single-word items —e.g., using fixed-capacity circular arrays. Recall that A denotes the cost of allocation, including subsequent deallocation.

Then, we can implement a (weighted or unweighted) sequence that achieves the amortized bounds shown below, where, for each operation, n is a size defined as above, and where $p_n = \lfloor \frac{2(n-1)}{K+1} \rfloor + 1$, for whatever the local definition of $n > 0$ is. Intuitively, p_n bounds the number of chunks stored in the underlying sequence.

- *Push and pop, with cost: $O(1) + \frac{1}{K}\big(A + \mathcal{C}_{pushpop}\big)$.*
- *Concatenation, with cost: $\mathcal{C}_{concat}(p_n) + O(K) + 4 \cdot \mathcal{C}_{pushpop}$.*
- *Split, or weighted split, with cost: $\mathcal{C}_{split}(p_n) + O(K) + 6A$.*
- *Space usage, bounded by: $2(1 + \frac{2}{K+1}) \cdot n + \mathcal{C}_{space}(p_n) + 5K + O(1)$ words.*

We present the representation and the invariants of the data structure that satisfies Theorem 1 and describe the implementation of the operations. The proof of the theorem can be found in Appendix A.

Representation. As discussed in Section 2, the main challenge in efficient chunking as required by Theorem 1 is to ensure that all operations on the underlying sequence data structure, which stores chunks are well amortized. To ensure

such amortization, we use a representation that keeps two chunks to store the items at the front of the sequence, and two chunks to store the items at the back. We refer to each of the special chunks stored at the two ends as a *buffer*. We then represent a sequence as a quintuple made of a *front-outer buffer*, a *front-inner buffer*, a *middle sequence*, which is an underlying sequence of chunks, a *back-inner buffer*, and a *back-outer buffer*. We write, e.g., (f', f, m, b, b') to denote such a quintuple.

Invariants. To guarantee efficiency, we maintain the invariant that the inner buffers are, at all time, either completely empty or completely full. Moreover, chunks in the middle sequence are never empty, and, to prevent sparse chunks from being formed, we ensure that any 2 consecutive chunks from the middle sequence have an average density of more than 50%. Our invariants are summarized as shown below, where $|c|$ denotes the number of items stored in a chunk c.

1. The front-inner and the back-inner buffers are either empty or full.
2. If c is a chunk from the middle sequence, then $0 < |c| \leq K$.
3. If c and c' are two consecutive chunks in the middle sequence, $|c| + |c'| > K$.

Operations. We implement the sequence operations as described below.

push-front. Consider a sequence (f', f, m, b, b') and an item x to push to the front of this sequence. If f' is full, we make room as follows. If f is empty, we simply exchange f with f', by swapping pointers. Otherwise, if f is full, we update the sequence to (c, f', m', b, b'), where c is a fresh chunk and where m' is the result of pushing the full chunk f to the front of m. At this point, the front-outer buffer is not full, so we push x to the front of this buffer.

pop-front. Consider a sequence (f', f, m, b, b'). If f' is empty, we populate it as follows. If f is not empty, in which case it must be full, we swap f with f'. Otherwise, assume f to be empty. If m is not empty, we pop from m, obtaining a nonempty chunk c and a new middle sequence m'; we then update the sequence to (c, f, m', b, b'). Otherwise, assume m to be empty. If b is not empty, in which case it must be full, we swap b with f'. Otherwise, if b is empty, we swap b' with f'. (Alternatively, we may directly pop from the front of b'.) At this point, the front-outer buffer is not empty, so we can pop from this buffer.

push-buffer-back. This auxiliary function is used to implement concat. When applied to a middle sequence m and to a chunk c, the function push-buffer-back modifies m so as to concatenate the items from c at its back, proceeding as follows. If c is empty, there is nothing to do. Otherwise, we perform the following two steps. (1) If m is nonempty and has a back chunk c' such that $|c| + |c'| \leq K$, then we pop c' out of m and merge the items from c' into c. (2) We push the chunk c to the back of m.

push-back and **pop-back** and **push-buffer-front** are defined symmetrically.

concat. Consider two sequences $(f'_1, f_1, m_1, b_1, b'_1)$ and $(f'_2, f_2, m_2, b_2, b'_2)$. To concatenate them, we start by concatenating the chunks b_1, b'_1 at the back of m_1, by applying twice the function push-buffer-back. Symmetrically, we concatenate f'_2 and f_2 to the front of m_2, using push-buffer-front. If m_1 and m_2 are both nonempty at this point, let c_1 be the back chunk of m_1 and c_2 be the front

chunk of m_2. If $|c_1| + |c_2| \leq K$, then we pop c_1 and c_2, merge the items from c_2 into c_1, and push c_1 back into m_1. (Remark: the pop and push operations on c_1 may be factorized with the earlier calls to push-buffer-back.) At this point, we concatenate the two underlying sequences m_1 and m_2 to get a new middle sequence, call it m_{12}. The final result of the concatenation is $(f'_1, f_1, m_{12}, b_2, b'_2)$.

split. Consider a sequence (f', f, m, b, b') and an index i denoting the split position. There are five cases; we consider the first one that applies.

- Case $i \leq |f'|$. We return two sequences $(f'_1, \emptyset, \emptyset, \emptyset, \emptyset)$ and (f'_2, f, m, b, b'), where (f'_1, f'_2) is the result of splitting the chunk f' at index i. More precisely, f'_1 denotes f' restricted to its items stored at index less than i, and f'_2 denotes a fresh chunk into which we move the items at index i or more in f'.
- Case $i \leq |f'|+|f|$. We return two sequences $(f', \emptyset, \emptyset, \emptyset, f_1)$ and $(f_2, \emptyset, m, b, b')$, where (f_1, f_2) is the result of splitting the chunk f at index $i - |f'|$.
- Case $i \leq |f'|+|f|+|m|$, where $|m|$ denotes the total number of items stored in all the chunks of m. Let j be equal to $i-|f'|-|f|$. We invoke the weighted split operation on the middle sequence to split m into a triple (m_1, c, m_2), such that the chunk c contains the item located at index j in m. Let (c_1, c_2) is the result of splitting the chunk c at index $j - |m_1|$, where $|m_1|$ denotes the weight of m_1 (i.e., the sum of the weights of the chunks in m_1). We then return the two sequences $(f', f, m_1, \emptyset, c_1)$ and $(c_2, \emptyset, m_2, b, b')$.
- The remaining two cases, $i \leq |f'|+|f|+|m|+|b|$ and $i > |f'|+|f|+|m|+|b|$ are essentially symmetrical to the first two cases.

4 Bootstrapped chunked sequences

The construction presented in Section 3 shows that, we can build a chunked sequence data structure on top of an underlying weighted sequence data structure. We can thus build a *bootstrapped* weighted sequence data structure by instantiating the underlying sequence to the structure produced by the theorem. To initiate the bootstrapping process, we can use a single chunk. The resulting bootstrapped chunked sequence data structure is a weighted sequence that, for a fixed value of K, achieves the asymptotic bounds as finger trees: constant time push and pop operations at the two ends, and logarithmic time concatenation and split. Unlike finger trees, however, our structure achieves constant factors amortized over K for push and pop operations, without significantly increasing the constant factors in concatenation and split. The precise bounds for our bootstrapped structure are as follows.

Theorem 2 (Efficiency of bootstrapped chunked sequence). *A bootstrapped chunked sequence has depth zero when $n \leq 1$, and has depth $d \leq \lfloor \log_{(K+1)/2} n \rfloor + 1$ otherwise. It achieves the following bounds:*

- *Push and pop, with cost:* $O(1) + \frac{4A}{K-1}$.
- *Concatenation, with cost:* $(d+1) \cdot \left(O(K) + \frac{16A}{K-1}\right)$.

– *Weighted split, with cost:* $(d+1) \cdot \big(O(K) + 6A\big)$.
– *Space usage, with a bound asymptotically equivalent to:* $2(1 + \frac{4}{K-1}) \cdot n$.

At first approximation, our bootstrapped data structure implements push and pop in $O(1) + \frac{A}{K}$, and concatenation and weighted split in $O(K \cdot \log_{K/2} n)$. Since $\log_{K/2} n$ is a rather small value the concatenation and split operations are competitive with the corresponding operations on finger trees, of cost $O(\log_2 n)$, with small values of K.

We note that since the bootstrapped data structure stores chunks of chunks (of chunks and so on), its nodes have high fanout, like some other data structures such as B+ trees [16]. A benefit of large fanout is that it decreases depth. Unlike B+ trees, however, our structure stores both ends of the sequence very close to the root, achieving constant-time access to the ends of the sequence.

We present the representation and the invariants of the data structure that satisfies Theorem 2 and describe the implementation of the operations. The proof of the theorem can be found in Appendix B.

Representation. We represent a bootstrapped chunked sequence as a list of *levels*. The deepest level is a *shallow* level that consists of a single weighted chunk. Every other level is a *deep* level that consists of a weight field and of pointers to the front-outer, front-inner, back-inner and back-outer weighted chunks. Chunks attached at depth 0 store individual items, chunks attached at depth 1 one store chunks of items, chunks at depth 2 store chunks of chunks of items, and so on...

We may choose different chunk capacities for different levels. However, our goal is to minimize both the product of the chunk sizes (to reduce the depth) and the sum of the chunk sizes (for fast split and concatenation). It therefore makes sense to select the same chunk capacity at every level.

Invariant. We enforce that if a level stores zero or one element (which may be items or chunks, depending on the level), then it is shallow. For all but the last level, we enforce the same invariants as those presented previously in Section 3.

Operations. We implement the sequence operations as described below. Operations on deep levels are similar to those described in Section 3, making recursive calls on the lower levels of the bootstrapped structure when operating on the middle sequence. Operations on deep levels also require updating the weight field. Below, we only focus on the treatment of shallow levels and the transitions between shallow and deep levels.

check. The purpose of this auxiliary function is to enforce the invariant that if a level contains zero or one element, then it is shallow. To that end, if the sequence is deep, we execute the following two steps, in order. (1) If all four buffers are empty and the middle sequence is nonempty, we pop a chunk from the front of the middle sequence and set it as new front-outer buffer. (2) If the sequence has an empty middle sequence, and all four buffers contain zero or one item in total, then we change the representation of the sequence to shallow (reusing one of the four buffers as chunk to represent the shallow level).

push-front. First, if the sequence is shallow and is made of a full chunk, we change its representation to deep, setting the chunk as back-outer buffer. Then, we push the incoming item to the front of the (shallow or deep) level.

pop-front. We pop an item from the structure, which may be shallow or deep. If the structure is deep, then we call check to possibly make it shallow.

concat. If both structures are deep, we call the concatenation procedure described in Section 3, then call check on the result. Else, we pop the items of the shortest sequence one by one and push them into the other one.

split. If the structure is shallow, we split its chunk at the appropriate position in order to isolate the targeted item, and we produce two shallow structures. If the structure is deep, we split it and then call check on both subsequences.

5 Benchmarks

To evaluate our chunking techniques, we wrote an implementation in C++ consisting of a few generic classes and two data structures that we benchmark. The first class is a generic C++ class that implements our chunking technique of Section 3. This chunked-sequence class is a templated class that is parameterized over the representation of its underlying sequence. Recall that we define the underlying sequence as any underlying sequence data structure that provides the full set of operations for maintaining a sequence of chunks. For the first data structure we benchmarked, we used an instantiation of our chunked-sequence class for which the underlying sequence is represented by our own ephemeral C++ implementation of Hinze and Patterson's finger tree. In addition, we coded a C++ class that implements our bootstrapped chunked sequence of Section 4. For the second data structure we benchmarked, we used an instantiation of our chunked-sequence class for which the underlying sequence is represented by our bootstrapped chunked sequence.

We ran all of our experiments with the same settings for K (i.e., chunk capacity) that we found to deliver good performance overall. For our chunked finger tree, we used 512; for our bootstrapped chunked sequence, we used 512 and 32 for the chunk-capacity settings of the outer and underlying sequences, respectively. We compiled all programs with GCC version 4.9.0, using optimizations `-O2 -march=native`. For the measurements we report in the abstract, we considered an Ubuntu Linux machine with kernel `v3.2.0-58-generic` and an 2.4GHz Intel Xeon 4870 processor with 1TB of RAM. We have obtained similar results on an AMD machine.

Our first study is a comparison between our chunked data structures and the STL deque, which as discussed earlier is also a chunked data structure that uses a chunk-capacity setting of 512 items. To measure the relative efficiency of long sequences of similar accesses to the ends of the sequence, we ran two simple benchmarks, namely LIFO and FIFO. Our LIFO benchmark proceeds in two steps: the first step is to fill a previously empty target sequence by pushing on the back end n 64-bit items and the second is to empty the target sequence by popping repeatedly from the back of the sequence. Our FIFO benchmark does the same thing as LIFO but pops from the front instead of the back end.

Table 5 shows the data from our experiments. The results in the first six rows of the table show that our two chunked data structures are at worst a few percent slower than the STL deque.

To measure the relative efficiency of interleaved sequences of pushes and pops, we ran experiments involving the depth-first and breadth-first search of a directed graph. Our depth-first and breadth-first codes are serial implementations of DFS and BFS that are each parameterized by C++ template parameter over the representation of their respective frontiers (i.e., lifo stack and fifo queue ADTs). We considered three graphs that each demonstrates key characteristics of our sequence data structures. Each graph is represented in adjacency-list format and uses 64-bit integer values to represent vertex ids. Looking at DFS and BFS, we see that, in every case except for DFS on tree, our chunked data structures are competitive with STL deques — sometimes slower and sometimes faster, but never differing by more than a few percent. In the case of DFS on tree, our chunked finger tree and bootstrapped chunked sequence are each nearly 40% slower than STL deque. This benchmark demonstrates a weakness of our implementations: the empty check is relatively costly because of the need to frequently check the emptiness of the two inner buffers and the middle sequence each time around main the DFS loop. The cost of the empty check is so pronounced in this particular case because the cost is not well amortized by sufficiently many push operations: the peak size of the DFS frontier is just a few tens of items. Although it affects implementations, this weakness is not inherent to our general technique. If performance on such small sequences is important, one can adjust the code to sacrifice a few instructions on each push and pop operation to cache the size of the structure. We plan to experiment with such optimizations in future work.

We ran an experiment involving single-processor executions of Leiserson and Schardl's parallel BFS algorithm (PBFS) [17]. The original PBFS uses a special-purpose bag data structure to manage the frontier of the graph traversal. During a given round, PBFS traverses its frontier in a divide-and-conquer fashion, using push and pop in the sequentialized leaves and split and concat in the divide and conquer stages, respectively. Their bag data structure is represented by a chunked binomial tree that bears some resemblance to our chunked representations. Despite the similarities, Leiserson and Schardl's structure provides access only to the front and supports only an approximate split-in-half operation. In our experiment, we consider the same chunk capacity as in the original PBFS paper, namely 128, and we applied to our data structure a few basic optimizations exploiting the fact that sequence order needs not be maintained —in particular, the back buffers become unnecessary. We see from the results table that our (bag-specialized) chunked data structures perform either better, or at worst a few percent slower, than the PBFS bag structure.

In the appendix, we report on two additional experiments to more thoroughly evaluate performance in scenarious that mix push, pop, split and concatenate, comparing in particular against the STL rope data structure [18].

Experiment	Seq. length	Nb. repeat	PBFS bag	STL deque	Our chunked finger tree	Our bootstr. chunked
LIFO	10^3	10^6		**5.46**	6.40	6.99
	10^6	10^3		**9.15**	10.95	10.97
	10^9	10^0		**12.07**	**13.28**	13.47
FIFO	10^3	10^6		**5.51**	6.34	6.40
	10^6	10^3		**9.16**	10.96	10.52
	10^9	10^0		**12.32**	**13.53**	**13.31**
DFS on grid 2D				**4.84**	**5.17**	**5.27**
DFS on tree				**11.25**	15.53	15.46
DFS on friendster				**63.43**	**64.67**	**65.28**
BFS on grid 2D				**39.89**	**36.74**	**36.68**
BFS on tree				**15.23**	20.54	21.08
BFS on friendster				**72.84**	**72.76**	**72.68**
PBFS on grid 2D			**39.87**	**38.17**	**38.71**	**38.67**
PBFS on tree			**19.00**	75.53	**21.53**	**20.46**
PBFS on friendster			**117.11**	137.36	**117.45**	**117.04**

Table 1. Measurements of benchmark runs. All measurements were taken from our Intel machine. Each data point represents wall-clock time in seconds. For each data point in the table, we made five runs and took the mean. The amount of noise that we observed between runs of the same application was below 1%. All data points that are no more than 10% slower than the best time are displayed in boldface. Our grid 2D graph is a grid graph in two-dimensional space, where each vertex is connected to each of its four neighbors in two dimensions. We used number of vertices $n = 2$ billion and number of edges $m = 4$ billion. Our tree graph is a perfect binary tree of 2^{29} nodes. Our friendster graph is a social networking graph that has $n = 65$ million vertices and $m = 1.8$ billion edges [6]. For LIFO and DFS, we use the stack optimization and for PBFS we use the bag optimization as described in the Appendix C, while for the other benchmarks we use the plain double-ended sequence-ordered chunk representation. For PBFS, we use linear-time split and concat for STL deque (only).

Our experiments show that our chunked data structures deliver excellent performance relative to the state-of-the-art data structures that we considered, even though each of these other data structures are highly tuned for a strictly narrower set of operations. Moreover, in contrast to the other state-of-the-art chunked data structures, ours come along with strong guarantees against worst case behavior. Furthermore, our benchmarks show promise for our chunked data structures to serve in roles that were previously not filled. On the one hand, for many sequential-programming applications, our data structures can be used in place of STL deque, and as a bonus, offer fast logarithmic-time split and concatenate operations. On the other, the PBFS application demonstrates potential of our chunked data structures in multicore applications as generic sequence containers and as splittable work-queue data structures in load-balancing algorithms.

6 Conclusion and future work

We presented algorithmic and implementation techniques for designing practically efficient sequence data structures that amortize expensive operations over a collection of items arranged as a chunk. We proved tight bounds by parameterizing our analysis by the cost of memory allocations, which, in our approach, correlate with expensive operations, and by counting such operations separately. We show that the proposed techniques perform well in practice. In future work, we plan to investigate the use of stronger invariants on consecutive chunks for increased space utilization, and consider persistent data structures.

Acknowledgements

This research is partially supported by the European Research Council under grant number ERC-2012-StG-308246 and the National Science Foundation under grant number CCF-1320563.

References

1. Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32:1260–1279, May 2003.
2. Jean-Philippe Bernardy. The Haskell `yi` package. `http://hackage.haskell.org/package/yi-0.6.2.3/docs/src/Data-Rope.html`.
3. Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
4. Adam L. Buchsbaum and Robert Endre Tarjan. Confluently persistent deques via data-structural bootstrapping. *J. Algorithms*, 18(3):513–547, 1995.
5. Adam Louis Buchsbaum. *Data-structural bootstrapping and catenable deques*. PhD thesis, Princeton University, 1993.
6. Stanford Large Network Dataset Collection. Friendster graph. `http://snap.stanford.edu/data/com-Friendster.html`.
7. Paul F. Dietz. Maintaining order in a linked list. In *STOC '82*, pages 122–127, Baltimore, USA, May 1982. ACM Press.
8. James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. SC '09, pages 53:1–53:11. ACM, 2009.
9. Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. STOC '77, pages 49–60, New York, NY, USA, 1977. ACM.
10. Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC '02*, pages 280–289, 2002.
11. Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *JFP*, 16(2):197–218, 2006.
12. Intel. Cilk Plus. `http://www.cilkplus.org/`.
13. Intel. Intel threading building blocks. 2011. `https://www.threadingbuildingblocks.org/`.
14. Haim Kaplan and Robert E Tarjan. Persistent lists with catenation via recursive slow-down. In *TOC'95*, pages 93–102. ACM, 1995.

15. Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. STOC '96, pages 202–211, New York, NY, USA, 1996. ACM.
16. Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*, chapter 6, pages 481–489. Addison-Wesley, 2 edition, 1998.
17. Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm. In *SPAA '10*, pages 303–314, June 2010.
18. SGI. STL rope. `http://www.sgi.com/tech/stl/Rope.html`.
19. Alexander Stepanov and Meng Lee. *The Standard Template Library*, volume 1501. HP Laboratories, 1995.

A Proof of Theorem 1

We now prove that the construction described in the previous section satisfies the bounds announced in the statement of Theorem 1. We begin the analysis with a key lemma related to the density of the chunks, showing that we need no more than approximately $\frac{n}{K/2}$ chunks for storing n items.

Lemma 1 (Maximal number of chunks in the middle sequence). *If a middle sequence stores n items then it contains no more than $\lfloor \frac{2(n-1)}{K+1} \rfloor + 1$ chunks.*

Let p be the number of chunks. If $n = 0$ or $p \leq 1$, then the result is trivial. Otherwise, let the sequence of values $(x_i)_{i \in 1..p}$ describe the size of the chunks in the middle sequence. By definition, we have $\sum_{i \in 1..p} x_i = n$. Chunks in the middle sequence are nonempty, so $x_i \geq 1$ for all i. Moreover, by the invariant on consecutive chunks, we have $x_i + x_{i+1} \geq K + 1$ for all $i \in 1..(p-1)$. We have:

$$\begin{aligned} 2n &\geq \left(\sum_{i \in 1..p} x_i\right) + \left(\sum_{i \in 1..p} x_i\right) \geq \left(\sum_{i \in 1..(p-1)} x_i\right) + \left(\sum_{i \in 2..p} x_i\right) + 2 \\ &\geq 2 + \sum_{i \in 1..(p-1)} (K+1) \geq 2 + (p-1)(K+1) \end{aligned}$$

It follows that $p \leq \frac{2(n-1)}{K+1} + 1$. Since p is an integer, we consider only the integer part of the bound, that is, $\lfloor \frac{2(n-1)}{K+1} \rfloor + 1$. □

Proof of Theorem 1. We define the potential of a sequence (f', f, m, b, b') to be $((\mathsf{if}\,|f| = 0\,\mathsf{then}\,0\,\mathsf{else}\,|f'|) + (\mathsf{if}\,|b| = 0\,\mathsf{then}\,0\,\mathsf{else}\,|b'|)) \cdot \frac{1}{K}(A + \mathcal{C}_{pushpop})$. We show that the amortized cost of each operation is equal to the real cost of the operation plus the variation in the total potential of the sequences(s) involved in the operation considered. Note also that we describe operations for the case of unweighted sequences; the generalization to weighted sequences is straightforward.

push-front. The only expensive case is when the front-inner buffer is full. We then perform a push operation on the middle sequence, for a cost of $\mathcal{C}_{pushpop}$, and allocates a new buffer, for a cost A. Because the outer buffer goes from full to empty, the potential decreases by $K \cdot \frac{1}{K}(A + \mathcal{C}_{pushpop})$. The variation in potential therefore pays for the expensive operations. Assume now that we push in the front-outer buffer. If the front-inner buffer is empty, the potential does not increase, so the cost is only $O(1)$. Otherwise, the potential increases by $\frac{1}{K}(A + \mathcal{C}_{pushpop})$, therefore the amortized cost of push is $O(1) + \frac{1}{K}(A + \mathcal{C}_{pushpop})$.

pop-front. The only potentially-expensive case is when both front buffers are empty and the middle sequence is not empty. In this case, we pop a chunk c from the middle sequence and deallocate a chunk. Since we charged the pop operation on the underlying sequence to the corresponding push, and the deallocation to the corresponding allocation, the amortized cost is zero. Note that the potential does not increase through the operation because because the front-inner buffer remains empty. So, in all cases, the amortized cost of pop is $O(1)$.

push-buffer-back. The operation $\mathsf{push\text{-}buffer\text{-}back}(m, c)$ requires accessing the back chunk of the middle sequence, for a cost of $O(1)$, may involve transferring up to K items into this chunk, for a cost of $O(K)$, and may involve pushing

the chunk c into the middle sequence, for a cost of $\mathcal{C}_{pushpop}$. So, the amortized cost of push-buffer-back is $O(K) + \mathcal{C}_{pushpop}$.

concat. The concatenation of two sequences of size n_1 and n_2 first involves 4 calls to push-buffer-back, for a total cost $O(K) + 4 \cdot \mathcal{C}_{pushpop}$. If the front chunk of the first middle sequence and the back chunk of the second middle sequence need to be merged, we need to pay an additional $O(K)$ for the merge operation. However, we don't need charge for an additional push operation, as an implementation may combine it with the push operation performed in the earlier calls to push-buffer-back. Then, there remains to pay for the concatenation of the two underlying sequence, for a cost bounded by $\mathcal{C}_{concat}(\min(r_1, r_2))$, where r_1 and r_2 denote the maximal number of chunks in the two middle sequences. By Lemma 1, $\min(r_1, r_2) \leq \min(\lfloor \frac{2(n_1-1)}{K+1} \rfloor + 1, \lfloor \frac{2(n_2-1)}{K+1} \rfloor + 1) = \lfloor \frac{2(\min(n_1,n_2)-1)}{K+1} \rfloor + 1 = p_n$, where $n = \min(n_1, n_2)$ and $p_n = \lfloor \frac{2(n-1)}{K+1} \rfloor + 1$. Hence, the total amortized cost of concat is $\mathcal{C}_{concat}(p_n) + O(K) + 4 \cdot \mathcal{C}_{pushpop}$.

split. Let n be the minimum of the sizes of the two sequences produced. For implementing split, we need to allocate a sequence structure where to extract the carved out part, that is, to allocate one record, 4 buffers, and possibly an empty middle sequence. So, we first need to pay a cost $6A$. Then, if the split position falls in one of the side buffers, the cost if $O(K)$. Otherwise, assume the split position to fall in the middle sequence. In this case, we need to pay for the weighted split operation on the middle sequence, for a cost of $\mathcal{C}_{split}(p_n)$, where $p_n = \frac{2(n-1)}{K+1} + 1$ is a bound on the number of chunks in the two middle sequences produced (using again Lemma 1). We then need to migrate data into one of the freshly allocated chunks, for a cost of $O(K)$. In all cases, the sum of the potential of the two sequences produced by the split does not exceed that of the original sequence. In summary, the amortized cost of a split operation is $\mathcal{C}_{split}(p_n) + O(K) + 6A$.

space. We need 5 words to represent the main record, and $\mathcal{C}_{space}(p_n)$ to represent the middle sequence. Moreover, we need to represent at most $p_n + 4$ chunks, each of which requires $K+3$ words. Using the fact that $p_n \leq \frac{2(n-1)}{K+1} + 1$, we deduce that the space usage is bounded by: $(\frac{2(n-1)}{K+1} + 5)(K+3) + \mathcal{C}_{space}(p_n) + 5$, which is less than $2(1 + \frac{1}{K+2}) \cdot n + \mathcal{C}_{space}(p_n) + 5K + O(1)$ words. □

B Proof of Theorem 2

We prove that our bootstrapped (weighted) sequences described in the previous section satisfy the bounds stated in Theorem 2. We begin with a bound on the depth and on the number of items involved at each level. Given a bootstrapped sequence, we define its depth, written d, as the number of deep levels. We write n_r the number of elements (items or chunks) stored in the levels at depth r. In particular, for the first level, we have $n_0 = n$, where n is the number of individual items stored in the sequence, and the last level stores n_d items.

Lemma 2 (Bound on the depth). *Let n be the number of elements in the bootstrapped sequence. If $n \leq 1$, then the depth d is zero. Otherwise, we have $d \leq \lfloor \log_B n \rfloor + 1$, where $B = \frac{K+1}{2}$. Moreover, for $r \in [0, d]$, we have $n_r \leq \frac{n}{B^r} + 1$.*

If the sequence at level r contains $n_r \leq 1$ elements, then it must be the last level, and have depth zero. If the structure is deep, that is, $r < d$, then, by Lemma 1, we have $n_{r+1} \leq \frac{2(n_r-1)}{K+1} + 1$, which is equivalent to $n_{r+1} - 1 \leq \frac{n_r-1}{B}$. Thus, for any $r \in [0..d]$, we have $n_r - 1 \leq \frac{n_0-1}{B^r}$. We deduce $n_r \leq \frac{n}{B^r} + 1$. To deduce a bound on the depth, consider $r \equiv 1 + \lfloor \log_B n \rfloor$. For this value of r, we have $r > \log_B n$ and therefore $\frac{n}{B^r} < 1$. The inequation $n_r \leq \frac{n}{B^r} + 1$ then implies $n_r < 2$. Since n_r is an integer, we have $n_r \leq 1$, meaning that there cannot be a level $r+1$. Therefore r is an upper bound for the depth, i.e., $d \leq \lfloor \log_B n \rfloor + 1$. □

Proof of Theorem 2. We write $|s|$ the number of elements (items or chunks) in a structure s. To formally define the potential, we introduce a constant u, which will serve as an upper bound on constant-time basic manipulations involved in the operation. We define the potential of a shallow structure with chunk c as $\max(0, |c| - 1) \cdot \frac{4A+u}{K-1}$, and define the potential of a deep structure of the form (f', f, m, b, b') as $((\mathsf{if}\,|f| = 0\,\mathsf{then}\,0\,\mathsf{else}\,|f'|) + (\mathsf{if}\,|b| = 0\,\mathsf{then}\,0\,\mathsf{else}\,|b'|)) \cdot \frac{4A+u}{K-1}$. Remark: the term $\frac{4A+u}{K-1}$ corresponds to the limit of a geometric series.

check. The operation is $O(1)$, because the pop operation it may perform on the middle sequence has been charged to the corresponding push, and because it creates a shallow structure with at most one item, hence of potential zero.

push-front. Let $\mathcal{C}_{pushpop}$ be the cost of push-front. We prove by induction over the levels of the structure that $\mathcal{C}_{pushpop} \leq u + \frac{4A+u}{K-1}$ for some $u \in O(1)$. This result suffices to derive $\mathcal{C}_{pushpop} \leq O(1) + \frac{4A}{K-1}$. There are four cases. (1) If the structure is shallow and not full, we pay some constant cost u_1, plus the increase in potential $\frac{4A+u}{K-1}$, and we are done. Picking a constant u greater than u_1 ensures $\mathcal{C}_{pushpop} = u_1 + \frac{4A+u}{K-1} \leq u + \frac{4A+u}{K-1}$. (2) If the structure is shallow and full, we transform it into a deep structure, an operation that requires 4 allocations —we reuse one chunk and we need to allocate 3 buffers and 1 shallow middle sequence. To pay for these $4A$, we consume all the potential associated with the shallow structure, that is, $(K-1)\frac{4A+u}{K-1}$. Note that we thereby obtain a structure with zero potential. We then push the incoming item as described next. (3) If the structure is deep and at least one of the two front buffers is not full, we push a value for some constant cost, which may be assumed smaller than u. Moreover, if the front-inner buffer is full, the potential increases by $\frac{4A+u}{K-1}$, a cost charged to the current operation. (4) If the structure is deep and both front buffers are full, we first rearrange the structure so as to make room, for a cost of $u + \frac{4A+u}{K-1}$ to push into the middle sequence (by induction hypothesis), plus a cost A for allocating a new chunk. For these two operations, we consume the potential associated with the front, that is $K \cdot \frac{4A+u}{K-1}$, a value greather than $A + u + \frac{4A+u}{K-1}$. Once done, we are ready to push the incoming item into the empty front-outer buffer, charging the cost of this operation to the current operation.

pop-front. If the structure is shallow, the cost is $O(1)$. If it is deep, the cost is $O(1)$, since all expensive operations are already charged to the corresponding push operation. The call to check is $O(1)$.

concat. First, assume that one of the two sequences is shallow, and let n denote the size of the shortest. We have $n \leq K$, and we assume $n > 0$ otherwise the concatenation is trivial and costs $O(1)$. Pushing its items into the other sequence costs $n(O(1) + \frac{4A}{K-1})$. To pay for this cost, we charge $O(K) + \frac{16A}{K-1}$ to the current operation, and consume the potential of the shallow sequence, greater than $(n-1)\frac{4A}{K-1}$. We can check that this amount suffices: $O(K) + \frac{16A}{K-1} + (n-1)\frac{4A}{K-1} - n(O(1) + \frac{4A}{K-1}) \geq O(K) - n \cdot O(1) + \frac{16A}{K-1} - \frac{4A}{K-1} \geq 0$.

Assume now that both sequences are deep. Let n_r denote the minimum of the sizes of the length of the two sequences at level r. According to Appendix A, $\mathcal{C}_{concat}(n_r) \leq O(K) + 4\mathcal{C}_{pushpop} + \mathcal{C}_{concat}(n_{r+1})$. So, $\mathcal{C}_{concat}(n_r) \leq O(K) + \frac{16A}{K-1} + \mathcal{C}_{concat}(n_{r+1})$. Summing up on all levels until reaching a shallow level gives: $\mathcal{C}_{concat}(n) \leq (d+1) \cdot \big(O(K) + \frac{16A}{K-1}\big)$.

split. For a shallow structure, we have $\mathcal{C}_{split}(n_d) \leq O(K) + A \leq O(K) + 6A$. For a deep structure (i.e., for $r < d$), according to Appendix A, we have $\mathcal{C}_{split}(n_r) \leq \mathcal{C}_{split}(n_{r+1}) + O(K) + 6A$. Summing up on all levels gives: $\mathcal{C}_{split}(n) \leq (d+1) \cdot (O(K) + 6A)$.

space. For a shallow structure, we have $\mathcal{C}_{space}(n_d) \leq K + O(1)$. For a deep structure (i.e., for $r < d$), according to Appendix A, we have $\mathcal{C}_{space}(n_r) \leq 2\frac{K+3}{K+1} \cdot n_r + 5K + O(1) + \mathcal{C}_{space}(n_{r+1})$. In order to sum up on all levels, we exploit the fact that $n_r \leq \frac{n}{B^r} + 1$ implies $\sum_{r=0}^{d-1} n_r \leq d + n(1 + \frac{1}{B} + \frac{1}{B^2} + \ldots) \leq d + \frac{n}{1-1/B} \leq d + \frac{K+1}{K-1}n$. Summing up gives: $\mathcal{C}_{space}(n) \leq 2\frac{K+3}{K+1} \cdot (d + \frac{K+1}{K-1}n) + (5d+1)K + O(d)$, therefore we deduce: $\mathcal{C}_{space}(n) \leq 2(1 + \frac{4}{K-1})n + (5d+1)K + O(d)$. Note that the first term dominates the expression, since $d \leq \lfloor \log_B n \rfloor + 1$. □

C Optimizations for stack and bag semantics

Our data structure can be simplified when accesses only take place at one end (catenable stack use), or when the order of the items is not relevant (bag use). When the sequence is used as a catenable stack (i.e., when push-back and pop-back are never used), the back buffers are not needed at all; they can be removed. Moreover, chunks can be represented as stacks, which are slightly simpler than circular arrays. Note that, in concatenation operations, the number of calls to push-buffer-back is reduced, thanks to the removal of the back buffers.

When the sequence is used as a bag, we can make one further simplification: we maintain the invariant that chunks stored in the middle sequence are always full. To implement concatenation, we push the items from the front buffers of the second sequence to the front of the first sequence. To implement split, we push all the items contained in the chunk extracted from the middle sequence to the front buffers of the two output sequences. In our benchmarking, we used these optimizations wherever we found an advantage.

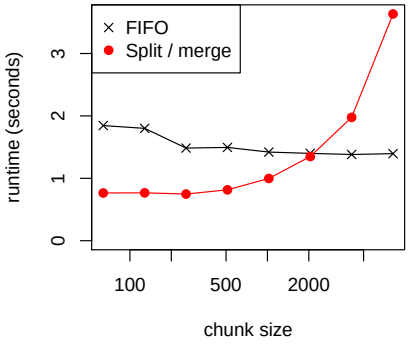

Fig. 1. Chunk-size benchmark.

D Additional benchmarking

D.1 Selecting a good chunk capacity

Because all the constant factors depend on the choice of K (the capacity of the chunks), we want to select a suitable value for K. We expect that higher values of K will be more favorable to push and pop operations, but that too high values of K will significantly increase the cost of split and concatenation operations.

To measure these effects, we considered two worst-case benchmarks: FIFO and split merge. The execution times, reported in Figure 1, are in accordance with the asymptotic bounds and confirm our predictions. Note that very small values of K are also suboptimal for the split-merge benchmark because they make the underlying tree too deep. From the shape of the curves, we can see that there is a range of values of K which are very close to being optimal both for split and merge operations and for push and pop operations. This range spans roughly between $K = 256$ and $K = 1024$. In what follows, we take $K = 512$, which is the same value as that used by STL deques.

D.2 PBFS detailed description

PBFS performs a level-order traversal using dual frontiers: one to represent the vertices to visit in the current level and one to build for the next level. The traversal of each level is a parallel traversal that proceeds in a divide-and-conquer fashion. For each level of recursion, the frontier corresponding to the current level is split approximately in half, and the two halves are processed recursively (usually in parallel, although for our purposes we consider just serial execution). When the recursive calls return, the results of the two calls are

Experiment	Seq. length	Nb. repeat	STL deque	STL Rope	Our chunked finger tree	Our bootstr. chunked
Filter	10^3	10^6	7.50	787.72	12.00	16.21
	10^6	10^3	71.75	914.78	12.92	14.52
	10^9	10^0	176.01	1114.90	14.26	15.62
Split merge + 0 push pop	$10^8/5$	200k		3.61	4.14	1.70
Split merge + 10 push pop	$10^8/5$	200k		4.05	4.16	1.73
Split merge + 100 push pop	$10^8/5$	200k		5.75	4.36	2.00
Split merge + 1000 push pop	$10^8/5$	200k		22.63	6.61	3.35

Table 2. Measurements of benchmark runs. All measurements were taken from our Intel machine. Each data point represents wall-clock time in seconds. For each data point in the table, we made five runs and took the mean. The amount of noise that we observed between runs of the same application was below 1%. An empty cell indicates not applicable. A value ∞ means that the run either timed out or was too slow to merit consideration. For our grid 2D graph: $n = 2$ billion and $m = 4$ billion; for our tree graph, $n = 2^{29}$ and $m = 2^{29} - 2$; for our friendster graph, $n = 65$ million and $m = 1.8$ billion.

concatenated to build the frontier for the next level. When the recursion reaches a frontier of fewer than 2048 vertices, the function branches to a base-case code that operates by repeatedly accessing frontier items individually by executing push and pop operations. The base case terminates when the frontier of the current level becomes empty.

D.3 Supplementary benchmarks

We performed a two additional experiments to more thoroughly evaluate performance in scenarious that mix push, pop, split and concatenate. In these experiments, we compared with the STL rope data structure [18], which is a production implementation that is based on a chunked tree representation [3]. Although designed for representing large strings, STL rope can be used as a generic sequence representation, providing push and pop at both ends of the sequence along with split and concatenate. The underlying representation of the STL rope is a self-balancing tree in which items are stored in fixed-capacity chunks of 28 items. We elided STL rope from the experiments reported in Section 5 because the STL rope was considerably outperformed in scenarios involving many push and pop operations.

To evaluate our implementations, we developed a synthetic "split merge" benchmark. We developed this benchmark to simulate, to a rough approximation, the work performed by online load-balancing algorithms that use the "steal-half" strategy for moving work items between processors [1, 10, 8]. In the benchmark, we start by creating 5 chunked sequences, each containing $n/5$ items, and we then repeat 200k times: merge two randomly chosen sequences, then split one randomly chosen sequence at an index in the sequence that is close

to the middle, then push a given number $h \in \{0, 10, 100, 1000\}$ of 64-bit items into the randomly chosen sequence. The expected execution time is of the form $200,000 * (C \log(n/K) + O(K) + O(h))$. In our experiments, we take $n = 10^8$. (Results are similar with any other reasonably-large value for n.)

We show the results in Table 2. On the one hand, the run times show that, when the number of pushes and pops are small, the STL rope outperforms our chunked finger tree by at most 15%. On the other, our bootstrapped chunked always performs significantly better than the other structures. Moreover, the performance of the STL rope degrades significantly as the number of pushes and pops increases. The relatively small chunk size of the STL rope is at least partly to blame for its poor push and pop operations, but we could find no parameter by which we could adjust the chunk size of the STL rope.

For our next experiment we consider a fork-join parallel algorithm for filtering items from a given sequence. Our benchmark program consists of a "filter" function of two arguments, namely a predicate function p and a sequence s, that applies p to each item in s, from left to right, and returns the sequence of those x for which $p(x)$ returned true. The filter function uses a divide-and-conquer strategy to expose parallelism high in the call tree and a serial filtering process to accumulate results in the serialized leaves of the call tree. The recursion cuts to serial code when the input sequence contains fewer than 2048 items. In the divide step, we split the input sequence into roughly equal halves and recur on the halves and, in the conquer step, we combine sequences returned from the two recursive calls by concatenating. The serialized base case repeatedly pops an item x from the input sequence and, if $p(x)$ returns true, pushes x on the output sequence. The base case repeats this process until the input sequence becomes empty, at which point it returns the output sequence. For this study, we consider only the single-processor execution of this parallel filter algorithm, although we note that parallelizing this function in a parallel dialect of C++, such as Cilk Plus [12] or TBB [13], is trivial because, by definition, there is no possibility of race conditions on accesses to the sequence. For this benchmark, we extended the STL deque with trivial split and concatenate operations that each have linear-time complexity in the size of the input. We used a predicate function p that always returns false, thereby maximizing the number of items being returned by the function.

From our results table, we can see that, for small input sizes, the STL deque has superior performance because, for small sequences, copying data is cheaper than performing a tree split. However, as the input size grows large, the performance of the STL deque degrades sharply due to copying overhead, whereas the performance of our chunked data structures grows linearly with the size of the input. Despite having log-time split and concat, the STL rope is well outperformed by all the other data structures, because the push and pop operations are extremely slow.

D.4 Results from a different platform

We ran our benchmarks on a different machine to increase confidence in our results. Our auxiliary machine is an AMD machine running Ubuntu Linux with kernel `v2.6.32-57-server` and an 2.1GHz AMD Opteron 6172 processor with 128GB of RAM. Table 3 shows our results. The results from this machine are not significantly different from those of our Intel machine.

Experiment	Seq. length	Nb. repeat	PBFS bag	STL deque	STL Rope	Our chunked finger tree	Our bootstr. chunked
LIFO	10^3	10^6		6.22	∞	5.44	6.19
	10^6	10^3		13.27	∞	13.50	12.59
	10^9	10^0		13.60	∞	14.64	13.60
FIFO	10^3	10^6		6.29	∞	5.81	5.86
	10^6	10^3		14.24	∞	14.82	14.01
	10^9	10^0		13.69	∞	13.74	14.35
Filter	10^3	10^6		10.06		14.82	18.38
	10^6	10^3		103.63		17.34	17.33
	10^9	10^0		222.69		17.59	17.43
Split merge + 0 push pop	$10^8/5$	200k			5.63	5.29	2.25
Split merge + 10 push pop	$10^8/5$	200k			5.89	5.33	2.29
Split merge + 100 push pop	$10^8/5$	200k			7.84	8.16	2.48
Split merge + 1000 push pop	$10^8/5$	200k			30.92	5.63	4.29
DFS on grid 2D				5.54	∞	5.69	5.76
DFS on tree				4.68	∞	15.66	18.66
DFS on friendster				86.31	∞	85.80	89.08
BFS on grid 2D				52.58	∞	44.09	44.75
BFS on tree				21.75	∞	21.18	20.84
BFS on friendster				110.24	∞	108.44	108.76
PBFS on grid 2D			49.35	49.63	∞	48.23	49.51
PBFS on tree			25.42	111.03	∞	28.55	30.17
PBFS on friendster			120.56	126.64	∞	118.09	118.27

Table 3. Measurements of benchmark runs. All measurements were taken from our AMD machine. Each data point represents wall-clock time in seconds. An empty cell indicates not applicable. A value ∞ means that the run either timed out or was too slow to merit consideration. For our grid 2D graph: $n = 2$ billion and $m = 4$ billion; for our tree graph, $n = 2^{29}$ and $m = 2^{29} - 2$; for our friendster graph, $n = 65$ million and $m = 1.8$ billion. For LIFO and DFS, we use the stack optimization and for PBFS we use the bag optimization as described in Appendix C, while for the other benchmarks we use the plain double-ended sequence-ordered chunk representation.

D.5 Comparison with STL vector

We considered the STL vector, which is an implementation of a resizeable array that supports fast access to the back end of the sequence. Internally, the STL

vector is represented by a contiguous array. When resizing is demanded by a push or pop operation, the respective operation has to reallocate an array and move the contents to the freshly allocated array. The run times show that, for a sequence of length 10^3, our chunked data structures perform just a few percent slower, for length 10^6 STL vector is twice faster than STL deque and our chunked sequences, but that for length 10^9, the STL vector degrades significantly. It is well known that this spike in run time is typical of STL vector and is caused by the memory overhead involved in resize operations that move large numbers of items.