Report

Sequences for Parallel Programming

Author(s):
Kourtis, Kornilios

Publication Date:
2015-05

Permanent Link:

https://doi.org/10.3929/ethz-a-010584487 →

Technical Report

Systems Group, Department of Computer Science, ETH Zurich

Sequences for Parallel Programming

by

Kornilios Kourtis

May 28, 2015

inf | Informatik
Computer Science

Sequences for parallel programming

Kornilios Kourtis
kkourt@inf.ethz.ch

Systems Group,
Department of Computer Science, ETH Zurich

May 28, 2015

Abstract

In this paper we build sequence data types that can efficiently support two important operations for parallel programming: *partition* and *concatenation*. We present a common chunk-based (instead of element-based) interface that enables good performance in practice, and consider two alternative implementations: ropes [2] and skip list arrays (based on skip lists [21]). We parallelize a run-length encoding algorithm as a motivating example, and show that, compared to dynamic arrays, the proposed data structures significantly increase the scalability of parallel algorithms that include both partitioning and concatenation operations.

1 Introduction

As parallel systems become the norm, programmers are forced to deal with parallelism if they want to take advantage of multicore machines. This requires rethinking algorithms and data structures. To this end, researchers have advocated abandoning the widely-used accumulator paradigm, and using divide and conquer algorithms instead [1, 24].

Implementing divide and conquer (D&C) algorithms that operate on sequences frequently requires partitioning to split up the data to form subproblems, and concatenation to combine partial solutions.

A simple illustrative example of this is the `filter` operation. It accepts a sequence `x` as input, and returns a sequence that contains the elements of `x` that satisfy a specified predicate as output. To implement a parallel `filter` we can split `x`, recurse until we reach a single element, and use the predicate to either return an empty sequence or a sequence with a single element. Finally, we concatenate partial results to form the output. In §2.1 we discuss another example, parallelizing a run-length encoding algorithm which also depends on partition and concatenation.

Traditionally, sequence data types are implemented using either arrays or linked lists. Linked lists are fundamentally problematic for parallel programming because they cannot be efficiently partitioned. Using arrays is better, but becomes problematic when concatenation is required.

In this paper, we investigate how to build data structures that can support both efficient partition and concatenation. We develop a common interface, which we implement with two data structures: one based on ropes [2] and one based on skip lists [21].

Performance is vital to parallel programming. Good scalability is not enough. If a parallel program does not outperform the sequential version, it is useless even if it is scalable. Achieving good parallel performance is not easy, and it usually depends on having a fast sequential computation at the bottom of the computation.

We discover that to support good performance, our

sequences interface needs to be chunk-based (instead of element-based). For data structure implementations this implies a two-level structure. At the top level, structure enables efficient concatenation and lookups. At the bottom level, efficient sequential computation is achieved using unstructured arrays.

In §2 we provide some background, parallelize a run-length encoding algorithm as an example, and discuss why linked-list and array sequence implementations are inadequate. In §3 we present an interface and discuss our implementations: ropes and skip-list arrays, which we evaluate in §4.

2 Background

Languages like Python, Perl, Ruby, and PHP, have become popular due to their ease-of-use. This can be largely attributed to their embedded data structures (e.g., lists and dictionaries in Python) that allow programmers to easily express algorithms, but also provide means for easy and efficient data exchange between different and independently-developed modules. Most of these languages implement two different data types: sequences and associative arrays. In this paper, we focus on sequences.

A sequence data type implements an ordered collection of values, mapping each ordinal key of a contiguous discrete range (normally starting from 0) to an element. Sequences are extensively used in functional programming where, combined with high-order functions like `map` and `reduce`, are able to express a rich set of algorithms. This methodology has greatly influenced imperative [25], and – more importantly for our purposes – parallel programming [6, 10, 23].

Most of these (imperative) languages implement sequences using dynamic arrays[1] because they are simple, easy to implement, and offer fast element access [22]. While dynamic arrays are a good match for sequential algorithms, parallel algorithms exhibit a different structure, and depend on different operations.

In the next section, we illustrate the differences between sequential and parallel algorithms using run-length encoding as an example.

[1] i.e., arrays that are dynamically resized as needed

```
1  class RLE(object):
2    def __init__(self,sym,cnt):
3      self.sym  = sym
4      self.cnt = cnt
5
6  def rle_encode(xs):
7    ret,curr,cnt = ([],xs[0],1)
8    for item in xs[1:]:
9      if item == curr:
10       cnt += 1
11     else:
12       ret.append(RLE(curr,cnt))
13       curr, cnt = (item,1)
14   ret.append(RLE(curr,cnt))
15   return ret
```

Listing 1: sequential run-length encoding using the accumulator paradigm.

2.1 A case study: run-length encoding

Run-length encoding (RLE) encodes consecutive instances of the same symbol s (called *runs*), as a tuple `(s,cnt)`, where `cnt` is the number of the instances. For example, the string `'aaaaabbbb'`, would be encoded as `[('a',5),('b',4)]`. RLE is a building block of many compression programs such as bzip2.

A sequential implementation of RLE is shown in Listing 1. This is probably the first algorithm one would write to solve this problem. To focus on algorithmic aspects instead of implementation details, we use Python syntax to write concise code, even though our implementation is in C.

The algorithm follows the accumulator paradigm: after program state is initialized, it is incrementally updated by consuming input. The state is the encoded output, the current symbol, and its count. As long as the same symbol appears in the input, the count is increased by one. When a different symbol appears, the current symbol and its count is appended to the output, and the new symbol is stored with a count of 1.

Dynamic arrays (or indeed linked lists) are a good match for this algorithm, since input is iterated sequentially and output is constructed by appending elements at the end of a sequence.

```
1  def prle_rec(xs):
2    if len(xs) == 1:
3      return [RLE(xs[0],1)]
4    mid = len(xs) / 2
5    rle1 = spawn prle_rec(xs[:mid])
6    rle2 = spawn prle_rec(xs[mid:])
7    sync
8    return rle_merge(rle1, rle2)
9
10 def rle_merge(rle1, rle2):
11   if rle1[-1].sym == rle2[0].sym:
12     rle2[0].cnt += rle1.pop().cnt
13   return rle1 + rle2
```

Listing 2: parallel run-length encoding using divide and conquer.

Although the accumulator paradigm is natural for sequential execution and generally achieves good performance, it cannot be efficiently parallelized due to the singular state it maintains.

To enable for parallelism, we express RLE using a D&C algorithm. We use the *spawn* and *sync* language constructs of the Cilk language [11] to create parallel tasks. The *spawn* keyword prepends a function call, specifying that the callee can (but does not have to) execute in parallel with the caller. The *sync* keyword acts as a barrier, and does not allow execution to proceed until all spawned functions have returned and have made their result available. Note that eliding *spawn* and *sync* results in a valid sequential program. Cilk parallel constructs can easily express divide and conquer algorithms, and more recent parallel languages have adopted similar features (e.g., OpenMP's tasks [17], and Chapel's `begin` statement [5]).

Our parallel D&C implementation is shown in Listing 2. We first split the problem into two sub-problems (*binary splitting*) by dividing the input (line 4), then recursively solve each sub-problem separately (lines 5-7), and finally combine the two partial solutions to get the final solution (line 8). We combine solutions by checking whether the last element of the first solution is the same as the first element of the second solution. If that is the case, we merge the two elements before concatenating the lists. We terminate the recursion by providing a unitary solution when the input is a single element (lines 2-3). The D&C algorithm can be easily parallelized by solving the two sub-problems in parallel.

It is worth noting that the same algorithm can be expressed using a data parallel `map`/`reduce`, by mapping each input element to the unitary solution and then performing a reduction on `rle_merge`.[2] Since `rle_merge` is an associative operation, the above `map`/`reduce` can be executed in parallel. Our discussion is based on the binary splitting method because it provides more transparency on the parallel execution, but our methods are applicable to the data parallel version as well.

As can be seen in our example, there are two basic operations used in the parallel RLE version: partitioning (lines `5-6`) and concatenation (line `13`).

In general, we argue that partitioning and concatenation are basic operations for parallel programming. On one hand, partitioning is used to split the input, allowing each parallel context can act on its own part independently. Parallel algorithms that produce lists, typically build them by combining partial solutions via concatenation. Reflecting the importance of these operations, Scala's parallel collection framework [18] is based on the similar concepts of splitters and combiners. Similarly, Cilk Reducers [10] implement reduction objects for Cilk and include a concatenation reduction object based on linked lists.

2.2 Implementing sequences

Traditional data structures used to implement sequence data types, i.e., linked lists and arrays, cannot implement both of these operations efficiently. While concatenating linked lists is extremely fast, partitioning is a $\Theta(n)$ operation, rendering linked lists utterly inapt for parallel programming. Contrarily, arrays can be (non-destructively) partitioned very fast and generally perform well because they completely lack structure and are very close to the data machine representation. Their lack of structure, however, makes it difficult to combine results. Concatenating arrays

[2] e.g., for Python: `reduce(rle_merge, map(lambda x: [RLE(x[0],1)], xs))`

is a $\Theta(n)$ operation because it requires copying at least the elements of one array, and maybe both if it is not possible to expand the other array using a function like `realloc`.

Overall, arrays and linked lists are antithetical with regard to partitioning and concatenation: arrays can be efficiently partitioned, but cannot be efficiently concatenated, while the opposite is true for linked lists.

Because, normally, partitioning is applied on the input and concatenation on the output, an alternative approach would be to use different data structures for the input and output. We see two problems with this approach. First, using two different sequence data type implementations significantly complicates the program and makes code maintainability difficult. As exemplified by embedded data structures in sequential languages, there are many usability benefits when using a single sequence data type across the whole program. Second, it is possible that the output of one component will act as input for another parallel component, blurring the input/output distinction for a particular sequence.

Hence our goal is to implement a data structure with which neither partitioning nor concatenation will dominate the parallel computation. We assume that the parallel computation operates over all input, so we require partitioning and concatenation being significantly better than $O(n)$.

```
1  def prle_rec(xs):
2    if len(xs) <= cutoff:
3      return rle_encode(xs)
4    mid = len(xs) / 2
5    rle1 = spawn prle_rec(xs[mid:])
6    rle2 = spawn prle_rec(xs[:mid])
7    sync
8    return rle_merge(rle1, rle2)
```

Listing 3: Optimized parallel RLE using cutoff.

The above requirements, however, are not enough. There is no point in writing parallel programs if they do not perform better than their sequential counterparts. One way to address this is to increase the time the program spends on (fast) sequential computations without hurting parallelism. A standard optimization to do that for D&C algorithms is to use a cutoff value to stop the recursion before reaching the unitary solution, and apply the sequential computation instead.

Applying this optimization for RLE encoding is shown in Listing 3. This leads to significantly improved performance because the sequential accumulator version is significantly faster than the recursive one.

Overall, achieving good performance for a parallel computation depends largely on efficient serial execution towards the leaves of the computation.

3 Design and Implementation

As discussed in the previous section, for a data structure to be effective in practice, it needs to facilitate fast sequential execution. This can be achieved by using contiguous areas of memory (we call them chunks), instead of elements, as the basic unit. We start by presenting a common interface for sequences based on chunks.

3.1 Interface

A summary of the interface shown in Table 1. The interface acts on *chunks*, i.e., contiguous memory areas represented as a pointer and their size, instead of individual elements. Chunks amortize the overhead of lookups and allow for fast sequential computations. Our initial implementation was element-based and performed worse from the sequential version for more than one order of magnitude. While for scalability and complexity this does not matter, in practice it makes a big difference because it requires a large number of cores (more than what many modern machines provide) to improve performance compared to the sequential case.

On the other hand, a chunk-based interface adds an additional abstraction for the programmer. At the lowest level, it makes program harder to write, especially if you using a chunk-based interface for both input and output. We argue, however, that this is a necessary evil: we cannot achieve acceptable per-

operation	description
`get(s, idx) -> chunk`	get chunk on `idx`
`append_prepare(s) -> chunk`	get a chunk to append elements to
`append_finalize(s, nelems)`	finalize append
`split(s) -> (s1, s2)`	split `s` into `s1` and `s2`, so that they have (roughly) the same number of elements
`concat(s1, s2)`	concatenate s2 to s1
`pop(s, nelems) -> chunk`	remove and return a chunk with no more than `nelems` elements from the end
`slice(s, idx, nelems) -> sl`	create a slice from `s`
`slice_slice(sl, idx, nelems) -> sl`	create a slice from `sl`
`slice_split(sl) -> (sl1, sl2)`	split slice
`slice_get(sl, idx) -> chunk`	get chunk on `idx` (`sl` must include `idx`)
`slice_next(sl) -> chunk`	return first chunk and move to next chunk

`chunks` are represented as a tuple of (pointer, number-of-elements).

Table 1: Interface summary

formance otherwise. Creating proper programming abstractions (e.g., specific iterators) can reduce the programmer's effort. Furthermore, with proper language support the programmer can use high-level operations (e.g., array operations, `map`/`reduce` operations, Chapel's promotions [5]) build on top of the proposed interface, but we do not tackle this here.

Next, we describe the operations supported by the interface. The `get(idx)` operation returns a chunk starting at element `idx`. Appending elements is a two step process: `append_prepare` returns a chunk for the new elements, and `append_finalize` completes the operation. Before finalization, the appended elements are not accessible using `get()`. The `split` operation destructively splits the array, so that the two pieces have (roughly) the same amount of elements. Finally, operations for concatenation (`concat`) and removing elements from the end of an array (`pop`) are also provided.

Although it would be possible to use the `split` operation for splitting input, it is not always efficient to do so because it creates two new lists out of the first, and in many cases (e.g., in the RLE example), this is not required. To avoid the unnecessary overhead, we introduce *slices*, which represent sub-regions of a sequence. Splitting a slice creates two new slices with the same number of elements.

Next, we discuss two implementations of this interface: ropes and skip list arrays.

3.2 Ropes

Tree data structures appear like a good candidate for providing efficient partition and concatenation [1, 24]. A tree data structure that is well suited for our purposes is Ropes [2]. Indeed, ropes have been used to implement sequence data types in parallel systems [8, 18].

Initially, ropes were proposed as an alternative to traditional unstructured string representations of programming languages, aiming to improve typical string operations such as concatenation.

As shown in Figure 1, ropes maintain chunks at their leaves, while their internal nodes contain the length of the string represented by the respective subtree. In general, for a rope to maintain its good performance properties for lookups and partitioning it needs to be well-balanced. In a perfectly balanced rope, each children would split the range of their parent in two equal parts. Maintaining good balance, however, is not easy in the face of concatenations. A basic concatenation operation can be implemented by creating a new root node with the two operands as children. Even if th the two tree operands are in-

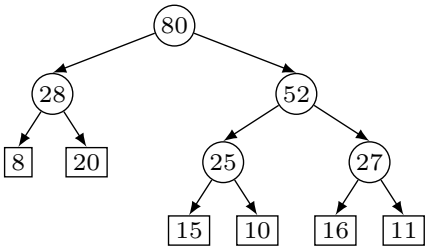


Figure 1: Example of a rope data structure. It includes 80 items. Internal nodes are represented with circles, while leaves are represented with rectangles.

dividually balanced, there are no guarantees for the balance of the result without assuming that the concatenation operands have similar sizes. To deal with this issue, ropes include a balance check, which can cause a rebalancing operation which effectively reconstructs the tree so that it is balanced [2].

Next, we describe how we implement slices and concatenation on ropes.

Slices/partition A naive approach to implement slices is to maintain an index and a length for each slice and simply use the sequence interface, ignoring any knowledge about the internal rope implementation. This, however, would result in always starting from the root node, which can be inefficient (especially for large sequences). Hence, we implement rope-specific slices by having each rope slice maintain two pieces of information. First a slice contains the smallest rope node that contains the whole slice (slice root), along with offset of the slice in the node. Second, a slice maintains the first leaf of the slice along with the corresponding offset in the leaf. This enables lookups to be performed starting from the slice root, while operations such as `slice_next()` return the chunk of the first leaf, and starting from there they traverse the tree upwards and then downwards to find the next first leaf. Although this does not change the complexity of the operations (e.g., $O(\log n)$ steps for partitioning), it offers significantly improved performance.

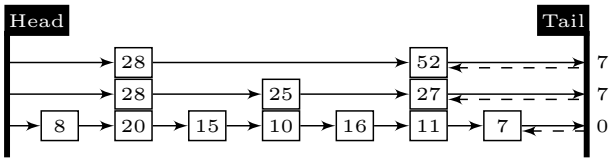


Figure 2: Example of a skip-list array. It includes 87 items in 7 nodes. Tail backward pointers are represented with dashed lines.

Concatenation We implement concatenation by creating a new root node and attach the operands as children. As an optimization, if both ropes include a single leaf and there is sufficient space, we copy the contents of the first second leaf to the first (and avoid creating a new root).

3.3 Skip List Arrays

Skip list arrays (SLAs) are based on skip lists [21], a probabilistic data structure, developed as an alternative to balanced trees. Each skip list node is assigned a random *level*, up to a maximum value, so that nodes with level $n+1$ are a fraction $\mathtt{p}$ of nodes with level n. This property is probabilistic and achieved via randomized level selection when a new node is created. Each node contains a number of *forward pointers* equal to its level, forming a linked list for each level so that the higher-level lists are sparser than the lower-level lists. To lookup a key we traverse the lists starting from the top level. Traversing high-level lists skips a large number of lower-level nodes. As we get closer to the desired key, we descend to lower levels. The probabilistic expected cost for the lookup operation is $O(\log n)$ [21].

In many ways, skip list arrays are to skip lists what ropes are to balanced binary trees. To implement SLAs, we augment traditional skip lists in three ways (see Figure 2):

(i) In each SLA node we store an arbitrary-sized *chunk* of elements, instead of a single element.

(ii) In addition to the *head* sentinel node, we also use a *tail* sentinel node with backward pointers to enable efficient concatenation. Using the

tail node, concatenation can be performed in a number of steps equal to the highest level of the two SLA operands. The tail node can also be exploited in two other SLA operations: appending items to the end, and retrieving the elements from the end.

(iii) Instead of just a pointer to the next node, we also maintain *partial element counts* for each level.[3] The partial count is equal to the number of items between the current and the previous node for each level. Hence, the partial count for the bottom level is the number of items in the chunk of the node. Maintaining item counts enables effective partitioning, while keeping them partial allows modifications (e.g., concatenation) with only local changes (no need to traverse all the nodes). No partial counts for the head node are needed. The tail count for the bottom level is always zero, but we keep it because it simplifies our algorithms.

Slices/partition Similarly with ropes, we implement SLA-specific slices using SLA pointers. An *SLA pointer* is a special forward pointer structure that allows to start a lookup from a previously defined point. SLA pointers act similarly to head nodes, with the addition of a (meaningful) count field that is set to the total number of elements from the start of the array to (but not including) the node they point to.

Concatenation The concatenation of `sla1` to `sla2` will have the head node of `sla1`, the tail node of `sla2`, and the its level will be eqal to the the maximum of `sla1` and `sla2` current levels. Initially, to avoid having large gaps in chunks, we copy all elements from the first node of `sla2` to the last node of `sla1` if enough space exists. Then we iterate over all possible levels and set up forward pointers and partial counts accordingly. There are three different cases depending on the level: (i) for a level that did not exist in `sla1`, we set its head forward pointer and partial count, (ii) for a level that did not exist in `sla2`, we set its tail backward pointer and partial count, and (iii) for a level that existed in both `sla1` and `sla2`, we link the first node of `sla2` to the last node of `sla1` and adjust the formers' partial count accordingly.

[3] a similar modification has been proposed by Pugh [20] to implement rank operations on skip lists.

3.4 Discussion

The main difference between ropes and SLAs is how balancing is achieved. In ropes, imbalance might happen which will lead to a rebalance operation that will have to reconstruct the tree, which can cause significant delays when it happens. On the other hand, SLAs do not require rebalance, but depend on statistical properties of individual nodes for achieving balancing.

4 Experimental Evaluation

4.1 Setup and methodology

We use four systems for our evaluation: a 2-way 10-core system (20 cores) Intel Ivy Bridge system, a 4-way eight-core system (32 cores) Intel Beckton system, a 4 way 12-core system (48 cores) AMD Magny-Cours system, and a 4-way 6-core system (24 cores) AMD Istanbul system. Table 2 provides more details.

All systems are running a 64-bit version of linux (kernel version 4.0). We use the gcc (version 4.9) implementation of Cilk Plus [4], which is the evolution of the original Cilk language that supports C++ and provides additional constructs. Our code is written in C, and to retain control over parallel execution we only use *spawn* and *sync*, ignoring more advanced Cilk Plus constructs like reducers [10]. In our initial experiments, the default GNU libc `malloc` did not scale well. Although ropes and SLAs can benefit from custom allocators (e.g., by exploiting object caching [3]), we did not want those potential benefits to be a part of our evaluation. Thus, we use tcmalloc [12], a generic allocator with the goal of reducing contention for multi-threaded programs.

We compare ropes and SLAs against dynamic arrays (*DAs*) because DAs offer the best performance and is what most languages use to imple-

	Ivy Bridge	Beckton	Magny-Cours	Istanbul
#cores	$2 \times 10 = 20$	$4 \times 8 = 32$	$4 \times 12 = 48$	4×6
Model	E5-2670v2	L7555	6174	8431
Frequency (GHz)	2.5	1.86	2.2	2.4
L1 (data/instr.)	32K/32K (×20)	32K/32K (×32)	64K/64K (×48)	64K/64K (×24)
L2 (unified)	256K (×20)	256K (×32)	512K (×48)	512K (×24)
L3 (unified)	25M (×2)	24M (×4)	6M (×8)	6M (×4)

Hyperthreading on all machines is disabled.

Table 2: Experimental platforms.

ment their embedded sequence data type. DAs consist of a single contiguous memory chunk, and are resized on demand using the `realloc` function on a parametrized granularity (`da_grain`). Thus, appending x elements in total requires $\lceil x/\texttt{da\_grain} \rceil$ calls of `realloc`. The `realloc` function will try to resize the current buffer. If that is not possible, it will allocate a new buffer and copy the data. We implement DA concatenation by performing a `realloc` on the first array, and then copying the second array to the newly allocated space.

For SLAs, we use a `p` value of 0.5 and define a maximum level equal to 5. These values were chosen conservatively based on the discussion in [21].

4.2 Sequential recursive sum

Our first experiment aims to quantify the overhead introduced by the added structure of ropes and SLAs compared to DAs. For that, we examine a worse-case scenario for ropes and SLAs, where DAs are optimal.

We use a *sequential recursive sum* as our benchmark. Initially, an array of 10^7 integers is allocated and filled with random values. Subsequently, we measure the time to recursively compute the sum of all array elements, by splitting the array into two parts until the size of the array reaches a `cutoff` value. When `cutoff` is reached, we apply the sequential accumulator algorithm.

Since the only operations used are splitting and lookup, DAs will perform optimally, while SLAs and ropes will have additional overhead due to their structure. Furthermore, the aforementioned two operations dominate execution time since addition induces negligible overhead.

We use a `cutoff` value of 10^4 integers: large enough to experiment with large chunks, and small enough to be orders of magnitude less than the array size.

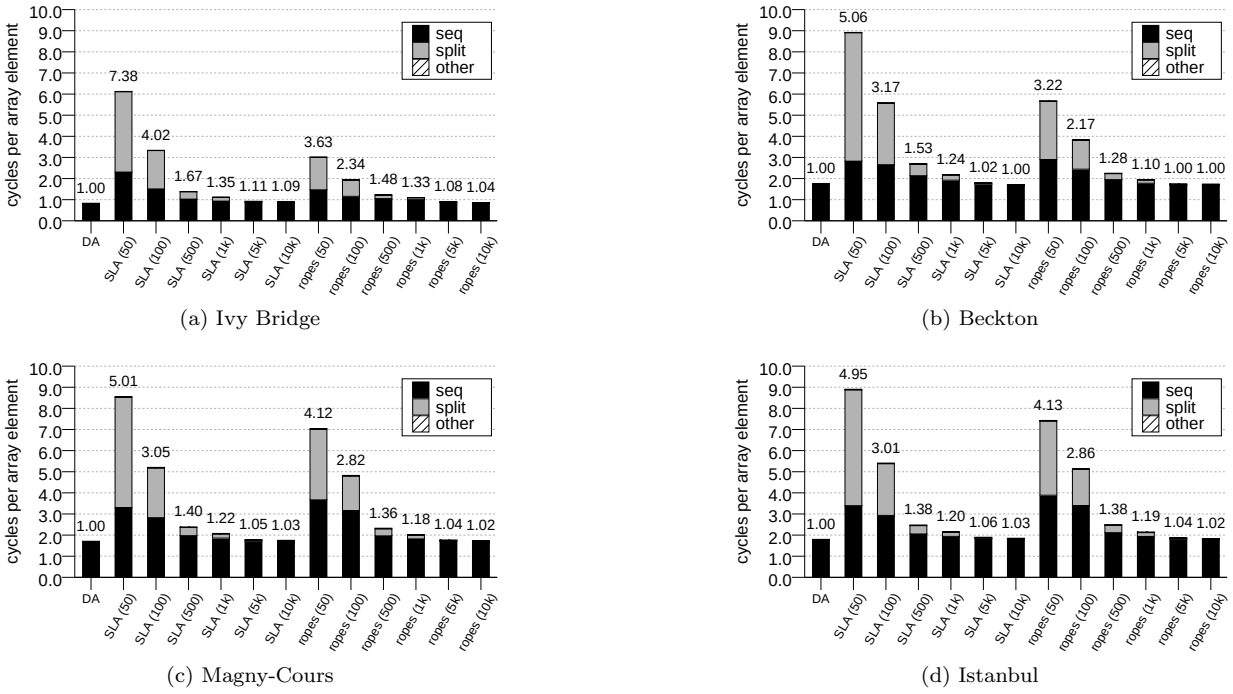
Figure 3 shows the execution time (normalized in cycles per array element) when using DAs, ropes, and SLAs for various chunk sizes. The execution time is broken into three parts: *seq*: time spent in the sequential accumulator algorithm, *split*: cost of splitting the array, and *other*: everything else. Additionally, the execution time normalized to the DA execution time is shown at the top of the bar.

One observation here is that the chunk-based interface is necessary to achieve good performance. Even a chunk size of 50 leads to a ×5 (or worse) execution time for our benchmark. Our initial implementation without chunks led to one or two orders of magnitue worse performance than DAs. Note that, in this case, core counts on most modern machines would not be able to reach a better performance than the sequential case.

For all machines, SLAs induce a higher overhead than ropes due to increased *split* time. Increasing the chunk size, improves both *seq*, and *split* performance. For sufficiently large chunks (5-10 thousand elements), the overhead becomes negligible.

4.3 Run-length encoding

Next, we use the RLE algorithm to evaluate SLAs on a scenario where both partitioning and concatenation are needed. Our goal is to compare the different sequence type implementations in such a scenario, and



(a) Ivy Bridge

(b) Beckton

(c) Magny-Cours

(d) Istanbul

Figure 3: Performance of a recursive sum of 10^7 integers, using a cutoff value of 10^4. For SLAs and ropes, the chunk size in integers is shown in parenthesis.

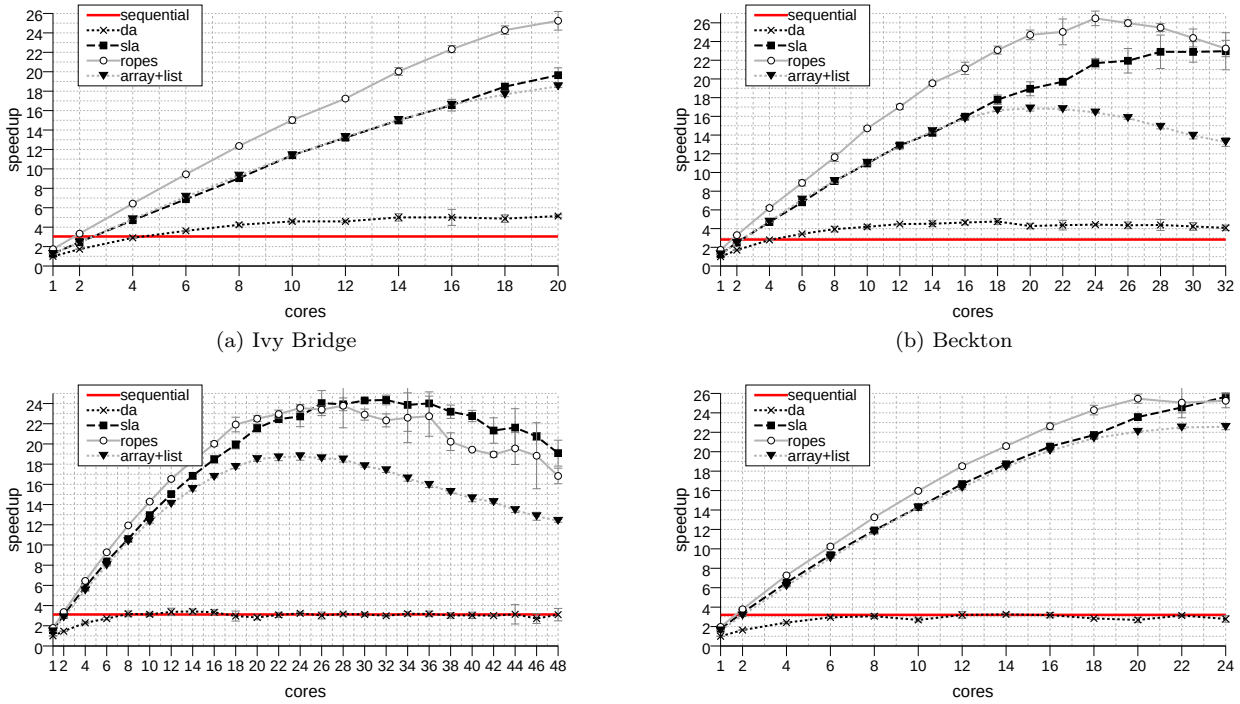
study the factors that impact performance. For input we use a symbol array such that the encoding result is $5 \cdot 10^6$ RLE pairs with a uniformly random count between 1 and 100. We use a `char` type for the symbol and a `size_t` type for the count of a pair. For the parallel RLE algorithm we use binary splitting and a cutoff value of 512 symbols for applying the sequential algorithm.

Our experiments include four implementations: three using the same data structure for both the input and output (*DAs*, *SLAs*, *ropes*), and one that uses an array for input and a linked list of RLE pairs for the output (*array+list*). In the SLA case, we use a chunk of 512 symbols for the symbol array and a chunk of 32 pairs for the output, so that both cases result in the same chunk size in bytes (512) on our systems. We do not explore the SLA parameter space in this work, but preliminary experimentation with parameters close to what we report here led to qualitatively similar results. For DAs, we also use 512 bytes for the allocation grain (`da_grain`).

An overview of our results is presented in Figure 4 which shows the speedup of each implementation over the single-threaded DA case as the number of threads increase. For each point in the graphs we executed the experiment 4 times, used the average value for the point, and the minimum/maximum values for the error bars. The *sequential* line shows the performance of the sequential accumulator algorithm using DAs.

The DA implementation scales poorly, having a maximum parallel speedup of no more than 5 for all architectures. As a result, it does not significantly outperform the sequential implementation. Note that the sequential implementation that uses the accumulator approach is roughly 3 times faster than the recursive DA implementation for 1 thread.

Our purpose for including measurements for the `array+list` case was to offer a comparison with



(a) Ivy Bridge

(b) Beckton

(c) Magny-Cours

(d) Istanbul

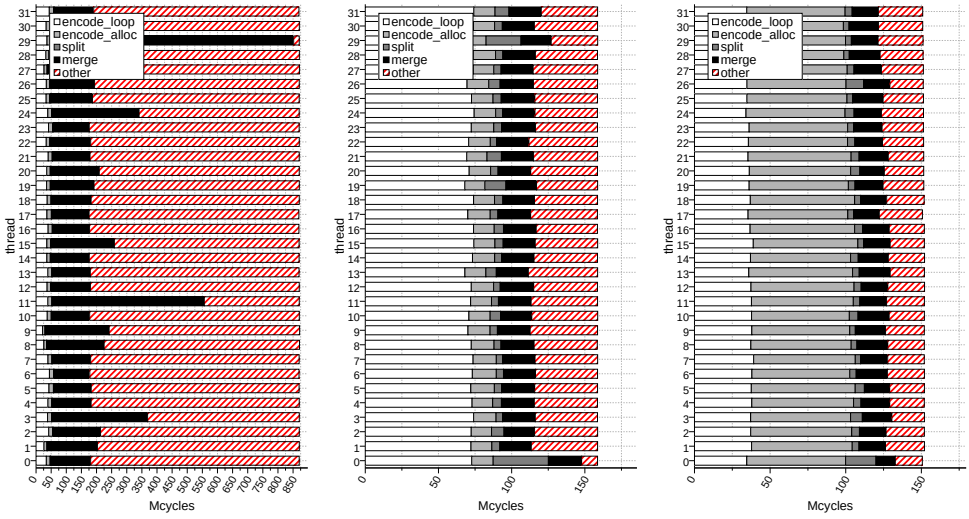
Figure 4: speedup of RLE for different sequence implementations compared to the single-threaded DA case.

the optimal case of using different data structures for input and output. In practice, however, this implementation can perform worse than ropes and SLAs. We attribute this to memory allocation scalability problems because each RLE element is allocated separately. A remedy for this issue would be to have multiple RLE nodes for each list element, whose logical conclusion is a structure similar to ropes and SLAs.

To gain additional insights on our results, we instrumented our code to take per-thread time measurements for different operations. The time breakdown results for the Beckton machine are shown in Figure 5. The time to execute the sequential encode (i.e., after the cutoff) value is broken down to the time to allocate a new sequence (`encode_alloc`) and the main loop of iterating input and appending to the output (`encode_loop`). `split` and `merge` depict the time to split the array an merge the subsolutions, respectively. Finally, `other` shows the remaining execution time until the program ends.

We can see in the breakdown graph that DAs perform poorly because the merge operation dominates execution time due to copying in concatenation. Furthermore, we notice significant load imbalance between threads, which we attribute to the merging operations of the final stages where most data are copied.

The ropes and SLAs implementations behave similarly. Ropes seem to perform better, but it is not clear what happens in the general case because our workload leads to fairly balanced ropes so the rebalancing operation is not triggered.



(a) DA (b) SLA (c) ropes

Figure 5: Beckton RLE execution time breakdown per thread for 32 threads.

5 Related Work

The sequence data type is fundamental in functional programming and, along with functions, the basic building block of functional algorithms. Many functional languages, inspired by lisp, implement lists via the `cons` operation, which is fundamentally problematic for parallel computing [24]. Researchers working on the fortress parallel language advocate using concatenation instead of `cons` as a fundamental operation for functional lists [16, 24]. Finger trees [14] are a functional data structure that supports efficient concatenation and splitting, but requires lazy evaluation. Data parallel Haskell [15] introduces a new array type in Haskell for supporting nested data parallel operations. Most imperative languages (e.g., Python and Perl) use dynamic arrays because they are simple, easy to implement, and offer fast element access which is the dominant operation in most sequential algorithms [22].

Reductions offer an attractive way to combine subsolutions in parallel programming. OpenMP [17] provides a reduction clause that can be used with some hard-wired operators on scalar values. Many languages (e.g., chapel [7]) offer support for user-defined reduction operations, that can be automatically parallelized if the operation is assumed associative.

In this paper we focused on programming patterns where each task operates on a distinct and well-defined subset of the data. These patterns can offer good performance and ease of use, but are not applicable to all programs. In more generic settings, it is not known beforehand what part of the array a task will access so synchronization between different tasks is required. Pugh [19] describes a synchronization method for skip-lists based on pointer reversal. That is, when a node x is deleted, its forward pointer is set to point to x's predecessor, so that the lookup operations that have reached x can continue. Herlihy et al. [13] describe a concurrent skip list implementation, while Fraser and Harris [9] apply a generic methodology for object-based software transactional memory to skip lists.

6 Conclusion

In this paper we discussed two implementations (ropes and skip list arrays) for building a sequence data type aimed for parallel programming. Both offer efficient concatenation and partition and thus can be paired with popular parallel programming methods such as divide and conquer algorithms and data parallel operations. Furthermore, a chunk-based interface allows them to perform good in practice.

7 Code availability

Our implementation has been made available at: https://github.com/kkourt/xarray/.

References

[1] Guy Blelloch. Parallel thinking. Keynote talk at the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009.

[2] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software Practice and Experience*, 25(12), December 1995.

[3] Jeff Bonwick. The Slab allocator: an object-caching kernel memory allocator. In *USENIX 1994 Summer Technical Conference*, 1994.

[4] Intel cilk plus. http://cilkplus.org/.

[5] Cray. Chapel language specification (ver. 0.92). http://chapel.cray.com/spec/spec-0.92.pdf, 2012.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[7] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[8] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

[9] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[10] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st ACM Symposium on Parallelism in Algorithms and Architectures*, August 2009.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[12] Sanjay Ghemawat. TCMalloc: Thread-caching malloc. http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html, 2007.

[13] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list (brief anouncement). In *10th International Conference on Principles of Distributed Systems*, December 2006.

[14] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2), March 2006.

[15] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science*, December 2008.

[16] Jan-Willem Maessen. Bulk parallel operations in Fortress. Slides for one-hour colloquium presented at Portland State University, 2010.

[17] Openmp application programming interface (ver. 3.1). http://www.openmp.org/mp-documents/OpenMP3.1.pdf, 2011.

[18] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *17th international conference on Parallel and Distributed Computing*, Euro-Par'11, August 2011.

[19] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, 1990.

[20] William Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, 1990.

[21] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 1990.

[22] Python design and history faq: How are lists implemented? http://docs.python.org/2/faq/design.html#how-are-lists-implemented.

[23] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *IEEE 13th International Symposium on High Performance Computer Architecture*, February 2007.

[24] Guy Steele. Organizing functional code for parallel execution; or, foldl and foldr considered slightly harmful. Invited talk at the 14th ACM SIGPLAN International Conference on Functional Programming, August 2009.

[25] Guido van Rossum. Origins of python's "functional" features. http://python-history.blogspot.ch/2009/04/origins-of-pythons-functional-features.html, April 2009.