Parallel Scan as a Multidimensional Array Problem

Artjoms Šinkarovs
A.Sinkarovs@hw.ac.uk
Heriot-Watt University
Edinburgh, Scotland

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl
Radboud University
Nijmegen, Netherlands
Heriot-Watt University
Edinburgh, Scotland

Abstract

For many algorithms, it is challenging to identify a suitable parallel version, as the design space is typically very large. In this paper we demonstrate how rank-polymorphic array languages can be used as a tool to explore such design spaces through concise high-level specifications. If input data can be organised into a multi-dimensional array, and the algorithm can be stated as a recursive traversal over sub-arrays, array languages offer a lot of expressive power. The reason for this is that array shapes can be used to guide recursive traversals. Conciseness of specifications comes from array reshapes that move the desired elements into canonical hyperplanes.

As a case study, we discuss several variants of implementing prefix sums (also known as scans) in SaC. We demonstrate how small code adjustments suffice to change the concurrency pattern exposed to the compiler. It turns out that variability that is typically achieved by generic inductive data types such as binary trees is a special case of what is offered by the array paradigm.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; **Concurrent algorithms**.

Keywords: array languages, algorithms, prefix sum, parallelism, rank polymorphism, functional programming

ACM Reference Format:
Artjoms Šinkarovs and Sven-Bodo Scholz. 2022. Parallel Scan as a Multidimensional Array Problem. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '22), June 13, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3520306.3534500

ARRAY '22, June 13, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9269-3/22/06.
https://doi.org/10.1145/3520306.3534500

1 Introduction

One of the key strengths of array programming languages is the abundance in concurrency that stems from the data-parallel nature of most array operations. This property empowers compilers to generate parallel code without requiring the programmer to specify any low-level details. Liberating the programmer from this burden has many advantages: program specifications can focus on the algorithm and are not littered with code that deals with necessities of the hardware, making them easier to write, easier to reason about, and easier to maintain. Another advantage is that the absence of target hardware specific code inherently renders them portable between systems that require different scaffolding for organising parallel executions.

The price to be paid for the absence of low-level details in the code has to be picked up by the compilers and code generators for such languages. They have to deal with the challenge of identifying how to expose which concurrency to parallel execution units. The complexity of this task typically is managed through a set of basic parallel operations such as map-like, fold-like or scan-like skeleton operations. Once a program has been specified as a composition of such basic building blocks, compilers usually apply various rewrite schemes in order to adjust a given composition to a form that suits well the properties of a given target hardware. Finally, bespoke implementations for the remaining individual skeleton operations are fine-tuned to the particulars of the executing system.

For many examples, it has been shown that this code-generation-based approach to parallel performance often rivals or even outperforms hand-optimised low-level parallel codes. However, no matter how good a given code generator is, in some situations, it will fail to generate high-performance code. This potential threat combined with the black-box nature of such highly code-transforming systems often leaves programmers with an uneasy feeling about committing to any one tool chain.

In this paper, we look at ways of how to help compilers and code generators without compromising the black-box nature of the overall setup. The central idea is to create alternative concurrency pattern through recursive function calls where we leverage argument shapes as guidance for the concurrency pattern. As it turns out, array languages

that support rank-polymorphic function definitions are ideally suited to express this.

As a running example, we consider an explicit implementation of scan operations in SaC — a strict functional array language. The main mechanism to control concurrency is a combination of data-parallel map-like operations and (recursive) functions. Maps evaluate all the elements concurrently, yet generation of the final array introduces one synchronisation point. Each function call introduces one synchronisation point, as per strict semantics it has to produce a value. Note that these assumptions do not necessarily translate directly into runtime behaviour. While they are realistic when assuming a direct code generation, compiler optimisations such as function inlining or loop-transformations may modify these patterns. Therefore, our study is not about the actual performance but rather about the concurrency patterns that can be exposed in specifications.

We implement different variations of scan whose computations are grouped by means of reshaping the data into multi-dimensional arrays and then computing scans through combinations of recursive function applications on hyperplanes of the multi-dimensional arrays. It turns out that this approach is not only expressive enough to obtain well-known specifications such as Blelloch's parallel scan [1], but that the formulations in terms of multi-dimensional arrays provide a better understanding of these algorithms and allow to vary them easily.

The contributions of this paper are:

- the observation that multi-dimensional array shapes can serve as guiding principle for concurrency pattern through recursive element traversals;
- a demonstration at the example of `scan`, how the ability to reshape data provides a powerful means to radically restructure such traversals;
- a systematic investigation how such algorithms can steer the relation between overall work and the number of required synchronisation steps solely through the chosen array shape.

We start the paper with a brief recap of SaC and its central language construct, the tensor comprehensions. Section 3 introduces the variant of `scan` that we are investigating and looks at its naive formulation. Section 4 explains how to control concurrency in scans. Sections 5 and 6 explore top-down and bottom-up scans. Section 7 presents related work and some real-world applications of scan, before Section 8 concludes.

2 Background

In this section we give the basic language constructions of SaC [5, 10] to help the reader understand the presented code snippets. While we use SaC as a concrete implementation vehicle, none of our examples are SaC-specific and can be ported to array languages that support rank-polymorphism, *e.g.* APL [8], J [13] or Remora [12].

All that is needed is an abstraction for n-dimensional arrays, with three basic primitives: selection, shape-enquiry and some form of n-dimensional map functionality. As well as the ability to write recursive functions that accept arrays of arbitrary ranks.

In SaC, arrays can be constructed out of individual elements by using square brackets:

$$a = [1,2,3] \qquad b = [[1,2],[3,4],[5,6]]$$
$$c = [[[1,2],[3,4]],[[5,6],[7,8]]]$$

All arrays are rectangular and all the nestings are homogeneous. Expressions like $[[1,2],[3]]$ are invalid.

Shape. Each array has a *shape* which is a vector (1-dimensional array) denoting the number of elements per axis. For the above examples, we have:

$$\texttt{shape}\,(a) = [3] \quad \texttt{shape}\,(b) = [3,2]$$
$$\texttt{shape}\,(c) = [2,2,2]$$

SaC is a strict first-order language, so higher-order functions are not supported. All expressions evaluate arrays — empty arrays as well as scalar values also have shapes:

$$\texttt{shape}\,([]) = [0] \qquad \texttt{shape}\,([[]]) = [1,0]$$
$$\texttt{shape}\,(42) = []$$

The shape of an empty vector is $[0]$; the shape of the two-dimensional array containing one row with no elements is $[1,0]$. The shape of a scalar value is the empty vector.

Indexing. Selections have C-like syntax:

$$\texttt{array[iv]}$$

Per convention, we call the variable that represents the index `iv`, which is a shorthand for `<index-vector>`. The following two constraints apply:

1. $\texttt{shape}\,(iv) \dot{\leq} \texttt{shape}\,(\texttt{shape}\,(array))$
   the length of the index vector can at most be as long as the array has axes, *i.e.* we are comparing two singleton vectors — the shape of the index vector must be element-wise less or equal (denoted as $\dot{\leq}$) to the shape of the shape of the array, and
2. $\texttt{iv} \dot{<} \texttt{shape}\,(array)$
   the values of the index vector must be in range, *i.e.* element-wise less ($\dot{<}$) than the corresponding shape elements.

In case `iv` has maximal length, the corresponding scalar element in `array` is selected. Otherwise, the selection pertains to the first axes of `array` only and returns a sub-array whose shape corresponds to those components of the shape of `array` for which no indices were provided. In case `iv` is empty, the entire array is selected.

Tensor Comprehensions. In the context of this paper, n-dimensional arrays are always constructed using the *tensor comprehension* [11] notation. Note that tensor comprehension is crucially important in SaC, as it gives a normal form to parallel operations. Most of the array combinators such as those found in APL can be expressed by means of tensor comprehensions.

An n-dimensional array can be specified by an expression of the form:

$$\{\ \texttt{idx-var -> elem-expr} \mid \texttt{idx-var < shp-expr}\ \}$$

where the shape of the result is determined by the value of `shp-expr`, and each element is computed by evaluating the expression `elem-expr`. SaC allows `elem-expr` to evaluate to non-scalar arrays, provided that all these expressions are of identical shape. The shape of the overall result is the concatenation of `shp-expr` and `shape(elem-expr)`.

Consider the following array definitions and the values they evaluate to:

$$\{\ \texttt{iv -> 1} \mid \texttt{iv < [3]}\ \} = [1,1,1]$$
$$\{\ \texttt{iv -> [1,2]} \mid \texttt{iv < [2]}\ \} = [[1,2],[1,2]]$$

The index variable can be referred-to in the element expression, and the upper bound specification can be omitted when it can be automatically inferred from the index use. For example, the expression

$$\{\ \texttt{iv -> a[iv] + 1}\ \}$$

computes an array that has the same shape as `a` but whose elements are incremented by one. Also, similarly to APL, arithmetic operations lift scalar arrays to scalar array elements of the other argument. For example, the above expression can be simply written as:

$$a + 1$$

Types. The type system of SaC is based on the type system of C that is extended with the following features:

1. Specification of array shapes;
   The types for scalar values are just like in C: `int`, `float`, `double`, *etc.* When specifying an array, we can chose how precise do we specify the shape. Here are several examples of arrays where elements are of type `int`:

   ```
   int[*]          // Any rank
   int[+]          // Any non-zero rank
   int[.] int[.,.] // 1-d, 2-d,...
   int[23,17]      // Statically known shape
   ```

2. Function overloading based on shape precision; Overloading helps to specify different functionality based on array shapes similarly to pattern-matching style specification often found in functional languages. For example:

   ```
   int foo(int)        { return 1; }
   int foo(int[.])     { return 2; }
   int foo(int[23,17]) { return 3; }
   int foo(int[+])     { return 4; }
   ```

   is a specification of a rank-polymorphic function `foo` that returns different values depending on the shape of the argument. Shapes form a partial order with respect to their precision. At dispatch the argument is matched against "the most precise" shape int the overloading hierarchy. The compiler will attempt to dispatch function applications statically, but this has no impact on the semantics of the program.
3. Automatic inference; This makes it possible to omit type declarations inside of function bodies:

   ```
   int foo(int a) {
     b = 1; c = 2; return a + b + c;
   }
   ```

4. Functions can return multiple values. For example, we can write a function:

   ```
   int, int[.] foo() {
     return (1, [1,2,3]);
   }
   ```

   which returns two values — the integer and the 3-element vector.

For more information on syntax, semantics, and code generation for various parallel hardware refer to [5, 10].

3 Introducing Scan

The computation of prefix sums of a vector is usually being referred to as `scan`. Given a vector `a` of values $\mathtt{a}_k$ with $0 \le k < n$, the result `s` of `scan (a)` is defined as

$$\mathtt{s}_i = \sum_{j<i} \mathtt{a}_j$$

for all i with $0 \le i < n$. Note that the first element of the result is always 0, and the sum of the entire vector is not contained in s. This definition sometimes is also referred to as "exclusive scan". "Inclusive scan" is almost the same — it does not contain the leading 0, but contains the overall sum as last element.

In the remainder of the paper, we exclusively consider exclusive scan. The other variant can be derived straightforwardly. We use `int` type and the plus operation. However, any monoidal binary operation can be used instead. In languages that support higher-order functions and type polymorphism, one could generalise the scans. In SaC, the mathematical definition from above can be expressed as:

```
int[.] scan_highlevel(int[.] a) {                                  1
  return { [i] -> sum ({[j] -> a[j] | [j] < [i]})                  2
         | [i] < shape(a) };                                       3
}                                                                  4
```

We see two tensor comprehensions and a reduction in the form of the application of `sum`. All three of these are mapped to built-in skeletons; they expose all possible concurrency to the compiler. If and how this concurrency is mapped into

parallel code depends on the code generator that is being used and, therefore, is outwith the control of the programmer. In the current compiler `sac2c` it depends on the chosen target hardware what kind of code actually is being generated.

While it would be nice if a compiler would detect the above pattern and transform it into highly efficient parallel code, it is rather unlikely. A compiler would need to identify that the computation of each element can share the computation of the previous element, and it would need to identify that the associativity of + allows for sharing partial sums arbitrarily.

To help the compiler, we re-formulate `scan`. We explicitly sacrifice concurrency for ensuring the sharing of sums.

4 Throttling Concurrency

Enforcing a sequential execution of `scan`, can be achieved in SaC by replacing the concurrent constructs used in the previous section by a sequential loop construct, *e.g.* by a `for`-loop:

```
int[.], int scan (int[.] a)                                  1
{                                                            2
  s = 0;                                                     3
  for (i = 0; i < shape(a)[0]; i++) {                        4
    t = a[i];                                                5
    a[i] = s;                                                6
    s += t;                                                  7
  }                                                          8
  return (a, s);                                             9
}                                                            10
```

Here, the additions are being shared as much as possible, computing the result strictly from left to right. That way, we perform n additions overall. We use the scalar variable `s` to carry around the state of our computation, starting with 0 and ending up with the sum of all elements of the array `a`. By returning this value as well, we can use `scan` as basic building block for later versions.

This version of `scan` is purely sequential. We now want to start re-introducing some concurrency to it as to enable parallel executions again. In the sequel, we assume a naive implementation of the code generator which maps all concurrency into parallelism and which does not apply any sophisticated code optimisations. The easiest way to introduce concurrency is to divide the vector into chunks, perform scans over all these chunks concurrently and then use the carries[1] of all chunks to adjust all but the first chunk.

The key idea here is that we reshape our input vector into a higher-dimensional array and use the shape as a means to guiding the parallel execution. As in all array languages which implement higher-dimensional arrays in continuous memory, in SaC such reshapes comes at no runtime cost; it merely introduces different ways of computing offsets into memory. This idea can be straight-forwardly applied to specify the chunking of the vector: we reshape our vector of shape $[n]$ into a matrix of shape $[p, s]$ (for simplicity, we assume here that p divides n), yielding p chunks of size s. Conveniently, SaC allows us to select any of the p chunks by indexing the matrix with a single index only. Now, we can express a concurrent version of `scan` as:

```
int[.,.], int scan (int[.,.] a)                              1
{                                                            2
  a, m = { [i] -> scan (a[i]) };                             3
                                                             4
  s = m[0];                                                  5
  for (i = 1; i < shape(a)[0]; i++) {                        6
    a[i] += s;                                               7
    s += m[i];                                               8
  }                                                          9
                                                             10
  return (a, s);                                             11
}                                                            12
```

Line 3 exposes the concurrency when computing the sequential scans on the p chunks. Notice here, that this relies on the overloading mechanism which ensures that the previously defined version of `scan` will be executed whenever `scan` is applied to 1-dimensional arrays. Since scanning the rows delivers two return values, the scan of the row and the overall sum of the row, this tensor comprehension now returns a matrix of shape $[p, s]$ and a vector of shape $[p]$ that holds the sums of the individual chunks.

The `for`-loop in the lines 6–9 now applies the necessary adjustments. The addition in line 7 performs an adjustment of an entire row with the running sum `s` of the row-sums contained in `m`. Note that this addition is a data-parallel operation as it is defined through a tensor comprehension in the standard library.

Having a closer look at this loop, we can observe that it combines two things: a scan on the carries (line 8) and the adjustment of the rows of `a` (line 7). By taking these apart, we can not only reuse our function `scan` but we can also join all data-parallel additions into a single one:

```
int[.,.], int scan (int[.,.] a)                              1
{                                                            2
  a, m = {[i] -> scan (a[i])};                               3
  m, s = scan (m);                                           4
  a = a ^+ m;                                                5
  return (a, s);                                             6
}                                                            7
```

We capture the addition of the elements of `m` to all rows of `a` by a generic infix operation `^+` which expects the shape of `m` to be a prefix of the shape of `a`:

```
int[*] ^+(int[*] a, int[*] m)                                1
{                                                            2
   o = shape(m);                                             3
   return {iv -> a[iv] + m[iv] | iv < o};                    4
}                                                            5
```

[1] By carry we mean the sum of all the elements within the given chunk

Note, that the addition within the tensor comprehension in line 4 is data-parallel. Trivial with-loop-folding [9] renders this a single data-parallel operation on all elements of a.

This leads to the following concurrency of our 2-dimensional version of scan: the tensor comprehension in line 3 of scan performs ps additions offering p-fold concurrency. The scan on the carries in line 4 performs p additions sequentially, while the application of ^+ in line 5 executes ps additions fully concurrent. Overall, we have $2ps + p$ additions, $s + p + 1$ steps when assuming full parallelism, and 2 synchronisation events from the two tensor comprehensions we eventually execute.

From these considerations, it becomes evident that we achieve a minimal number of steps for this approach when choosing p and s as $\sqrt{n}$. As the number of sequential steps constitutes the lower limit for any possible parallel execution, we need to modify our specification of scan further, if we want to get beyond $n/(2\sqrt{n}+1)$ as conceptual upper limit for a possible speedup.

5 Top-Down Recursive Scan

In the 2-dimensional scan from the previous section, concurrency is mainly throttled through the calls to sequential scans on vectors. A large inner axis enforces long sequential scans on the chunks and a large outer axis enforces a long sequential scan on the carries. The only way to reduce both of these at the same time is to introduce further dimensions. For this purpose, we can turn our 2-dimensional specification into a rank-polymorphic one. Here the underlying array algebra of SaC turns out to be very helpful: all that is needed to make scan applicable to arbitrarily ranked arrays is to change the argument type declaration:

```
int[+], int scan (int[+] a)                    1
{                                              2
  a, m = { [i] -> scan (a[i]) };               3
  m, s = scan (m);                             4
  a = a ^+ m;                                  5
  return (a, s);                               6
}                                              7
```

As an example, consider application of scan to an array of shape $[p, q, r]$. The concurrent call in line 3 will scan all sub-arrays of shape $[q, r]$, returning scanned versions of these into a new array a of shape $[p, q, r]$. Furthermore, it returns a vector m of shape $[p]$ containing the carries of the scans of the sub-arrays of shape $[q, r]$. The scan of carries works in the same way as in the previously considered version. Thereafter, we add the carries to the corresponding elements of the sub-arrays of a. Note that our generic definition of ^+ from the previous section is still up for the task. Instead of adding the carries to individual rows, it now adds them to sub-arrays of shape $[q, r]$.

Let us consider the behaviour of this specification in terms of additions, steps, and synchronisations. First, we assume an argument of shape $[p, q, r]$. From the previous section, we know that the tensor comprehension in line 3 performs p-fold $2qr + q$ additions, using $q + r + 1$ steps and 2 synchronisations each. One more synchronisation derives from the tensor comprehension itself. The scan in line 4 yields p additions, p steps and no synchronisations. Finally, we have pqr additions in 1 step and 1 synchronisation in line 5. Overall, we obtain $3pqr + pq + p$ additions, $p + q + r + 2$ steps and $2p + 2$ synchronisations.

By induction over the dimensionality, we obtain for an argument of shape $[p_1, \ldots, p_m]$:

$$m \times \prod_{i=1}^{m} p_i + \sum_{i=1}^{m-1} \left(\prod_{j=1}^{i} p_j \right) \quad \text{additions}$$

$$\sum_{i=1}^{m} p_i + m - 1 \quad \text{steps, and}$$

$$\sum_{i=1}^{m} \left(2 \times \prod_{j=1}^{i-1} p_j \right) \quad \text{synchronisations.}$$

Again, we see that the overall parallel performance that we expect depends on the chosen shape. If we choose $m = 1$, we obtain a sequential executions with n steps and n additions. If we want to decrease the overall number of steps, we have to introduce further dimensions and we have to choose as small as possible shape components p_i, ideally 2.

In order to get an idea of the best possible asymptotic behaviour with respect to the number of steps, we look at the extreme case, where we assume that we can reshape our original vector of length n into an m-dimensional array of shape $[2, \ldots, 2]$. For such a scenario, we obtain the following characteristics: we perform $O(n \log n)$ additions, require $O(\log n)$ steps and we need $O(n)$ synchronisations.

6 Bottom-Up Parallel Scan

The scan from the previous section now has the best possible complexity in terms of steps, allowing for a maximal speedup of $n/\log n$. However, this has come at the expense of factor $\log n$ more work and $O(n)$ many synchronisations.

While the increase in overall work might be acceptable if it is possible to hide the additional work through parallel executions, the very high number of synchronisations is likely to introduce a huge overhead.

When looking at the runtime considerations of the previous sections we can see that the vast amount of synchronisations stems from the recursive mapping of the scan onto sub-arrays with a dimensionality decreased by 1. While compiler optimisations like with-loop-scalarisation [6] in principle could elide such nested synchronisations, this is rather challenging in the context of recursive definitions.

Instead of relying on compiler smartness, we can modify our definition of scan in a way that avoids these nested synchronisations in the first place. Here, we can leverage

the benefit of the array-based setting. Instead of unfolding the concurrency dimension by dimension, we can operate directly on the last axis. All we need to change in our specification is our explicit tensor comprehension:

```
int[+], int scan (int[+] a)                                    1
{                                                              2
  a, m = { iv -> scan (a[iv])                                  3
           | iv < drop([-1], shape(a)) };                      4
  m, s = scan (m);                                             5
  a = a ^+ m;                                                  6
  return (a, s);                                               7
}                                                              8
```

In lines 3-4 we now concurrently map the recursive call to scan over all innermost vectors. Since `scan` on vectors is done sequentially, there are no more synchronisations happening inside those scans. The consequence of this choice is a potentially larger shape on the returned array `m` of carries. Let us consider the 3-dimensional case again with shape $[p, q, r]$. Whereas the top-down version from the previous section returned a vector of shape $[p]$ here, now, we obtain an array of shape $[p, q]$ as carries. Consequently, the scan in line 5 is a recursive call to this instance of `scan`. It results in the scanned carries, now stored in an array of shape $[p, q]$. As in the previous cases, our generic function ^+ is perfectly capable to deal with higher-dimensional second arguments, delivering the correct overall result.

The interesting question now is: does this change in the traversal through our multi-dimensional array actually improve its overall characteristics with respect to the number of additions, step, and synchronisations? Surely, the number of synchronisations due to lines 3-4 has improved but the scan on the carries in line 5 is now more complex.

As this version is identical to the previous version when applied to 2-dimensional arrays, let us consider the case for arguments of shape $[p, q, r]$ again. The concurrent calls to `scan` in line 3 require pqr additions in total, r steps, and 1 synchronisation, as they build on pq sequential scans. The scan of the carries in line 5 is a 2-dimensional scan on an array of shape $[p, q]$ requiring $2pq + p$ additions, $p + q + 1$ steps and 2 synchronisations (see Section 4). Finally, the addition in line 6 performs pqr additions in 1 step using 1 synchronisation. Overall, this leads to $2pqr + 2pq + p$ additions, $p + q + r + 2$ steps, and 4 synchronisations.

Inductively, we obtain for an argument of shape $[p_1, \ldots p_m]$:

$$\begin{aligned} 2 \times \sum_{i=2}^{m} \prod_{j=1}^{i} p_j + p_1 \quad & \text{additions} \\ \sum_{i=1}^{m} p_i + m - 1 \quad & \text{steps, and} \\ 2m - 2 \quad & \text{synchronisations.} \end{aligned}$$

We can see a clear improvement over the top-down version from the previous section. When we look at reshaping a vector of length n into m-dimensional array of shape $[2, \ldots, 2]$, we now obtain the following characteristics: we still require $O(\log n)$ steps but do fewer additions, coming down to $O(n)$, and, most importantly, we bring the required synchronisations down to $O(\log n)$. At least asymptotically, this is the best we can hope for.

6.1 Alternative Bottom-Up

There are two important differences between the top-down and bottom-up specifications: the amount of work and the number of synchronisations. The bottom-up version from the previous section builds on scans of higher-dimensional arrays of carries which cannot be straight-forwardly mapped into the overall result array. This raises the question whether compilers are able to optimise away allocations of intermediate arrays which could be very difficult in the context of code generation for GPUs. As another point in the design space, let us investigate an alternative formulation of the bottom-up approach which, when compared to top-down, improves the number of synchronisations, keeps the amount of work, and explicitly uses only two arrays of the original size.

For simplicity we start with arrays of shape $[2, \ldots, 2]$. Such arrays are isomorphic to perfect binary trees. If we carry on with this analogy, top-down scans traverse the tree from the root down to the leaves, and it makes a recursive call on every node. This is why the number of synchronisations is comparable with the number of nodes. However, dependencies of the algorithm make it possible to reverse the direction of the traversal: from the leaves down to the root. In this case, all the nodes at the given depth can be processed concurrently.

In array nomenclature, bottom-up traversals are characterised by iterating array shapes right-to-left. Sometimes, expressing operations on the right-most axis can be simplified by applying an array transpose. Consider adding all the pairs of the following array of shape $[3, 2, 2]$:

$$\begin{pmatrix} \langle 1, 2\rangle & \langle 3, 4\rangle \\ \langle 5, 6\rangle & \langle 7, 8\rangle \\ \langle 9, 10\rangle & \langle 11, 12\rangle \end{pmatrix}$$

The operation happens on the right-most axis, so we can write an expression:

```
{iv -> a[iv][0]+a[iv][1] | iv < [3,2]}                         1
```

However, if we transpose the array by bringing the last axis into the front, we will get the following arrays at indices $[0]$ and $[1]$ respectively:

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix}$$

This means that we can get the same sum by simply computing $a^T[0] + a^T[1]$, where a^T is the transposed array.

Coming back to our example, if we transpose the array of elements and the array of carries at every iteration, we should be able to access the necessary sub-arrays by indexing on the first axis:

```
int[*], int scan(int[*] a)
{
  m = a; a *= 0;
  for (i = 0; i < dim(a); i++) {
    a, m = pair(rot(a), rot(m));
    a[1] = a[1] +^ m[0];
    m = m[0] + m[1];
  }
  return (a, m);
}
```

Notice that the first dimension of `a` and `m` always match. However, as `m` shrinks at every iteration, the updates of the remaining axes of `m` have to be propagated into the right-hand-side of the shape. This operation (denoted here as `+^`) is commonly referred as ranked addition. A transpose that brings the last axis to the front here is denoted as `rot`. One could expect to find such functions in an array library, but their implementations in SaC are straight-forward:

```
int[*] rot(int[*] a)
{
  i, o = takedrop(shape(a), -1);
  return { iv -> {jv -> a[jv][iv] | jv < o}
         | iv < i};
}

int[*] +^(int[*] a, int[*] b)
{
  i, o = takedrop(shape(a), -dim(b));
  return {iv -> a[iv] + b | iv < o};
}
```

Observe what happens with shapes of a and m through the iterations, assuming that the initial shape of a is $[p, q, r]$:

Iter	a	m	$\sum_i m[i]$
1	$[p, q, r]$	$[r, p, q]$	$[p, q]$
2	$[q, r, p]$	$[q, p]$	$[p]$
3	$[p, q, r]$	$[p]$	$[]$

For arrays of general shapes, updates of `a` have to happen on all the sub-arrays on the first axis. The same holds for the computation of carries. This can be expressed as follows:

```
int[+], int
buscan(int[+] a)
{
  m = a; a *= 0;
  for (i = 0; i < dim(a); i++) {
    a, m = pair(rot(a), rot(m));
    a, m = scan_fold(a, m);
  }
  return (a, m);
}
```

Where `scan_fold` is defined as:

```
int[+], int[*] scan_fold(int[+] a, int[+] m)
{
  t = m[0];
  for (j = 1; j < shape(a)[0]; j++) {
    a[j] = a[j] +^ t;
    t += m[j];
  }
  return (a, t);
}
```

Complexity-wise, there is the same work and depth as in the top-down scan, but there are significantly fewer synchronisations. For an array of shape $[p_1, \ldots, p_m]$, the iterations of the `for`-loop make $2p_m$, $2p_{m-1}$, $2p_{m-2}$+, ...steps. In total this results in $2\sum_i p_i$, which yields on asymptotic complexity of $O(\log n)$.

6.2 Fine-Tuning

With generalised scans operating on multi-dimensional arrays, we can fine-tune the operation that we apply to one-dimensional arrays. We reshape one-dimensional vector in a suitable shape, then we run one of the above scans, and we reshape the result back:

```
int[.] reshaped_scan(int[.] a, int[.] s) {
  a    = reshape(s, a);
  a, _ = scan(a);
  return (reshape(prod(s), a));
}
```

The shape that we provide to the above function controls the level of concurrency we suggest in our specification. Typically, the number of parallel units on the actual hardware is limited. While, with unlimited resources it always make sense to build a perfect binary tree (shape $[2, \ldots, 2]$), on real architecture the optimal choices may be different. Generalised scans allow one to experiment with exactly these choices. An array of 64 elements could be reshaped into shapes $[4, 16]$, $[4, 4, 4]$, $[16, 2, 2]$, *etc.*

The ability of SaC to support shape-based overloaded functions makes it possible to provide additional control on how `scan` is being computed when recursion reaches sub-array of a certain shape.

```
int[128,128], int scan(int[128,128] a) { /* ... */ }
int[*],       int scan(int[*] a)       { /* ... */ }
```

This means that concurrency of scan may not only be controlled by the choice of array shape, but also by providing additional overloadings for specific shapes. This could be useful when arrays fitting certain cache sizes benefit from special treatment.

While the above behaviour can be acheived by introducing explicit conditionals, the overloading mechanism ensures

that any (recursive) use of scan will include such an instance. Explicit conditionals may be useful though when dispatching on non-shapes, *e.g.* symmetric matrices, sorted array, *etc.*

7 Related Work

In this section we review some existing approaches to handling scan in programming languages. We give two examples of using scans to specify real-world parallel algorithms.

Application (removing tags). Scan is being used in surprisingly many applications including common algorithms like sorting, text processing, *etc.* One folklore example from APL is removal of quoted objects from the text. Consider removing html tags from a well-formed document that is stored in the variable *b*:

```
  b ← '<b>Hello <i>world</i></b>'
```

In parallel we can find a mask indicating where the tag symbols < and > begin and end. Values 0 and 1 represent false and true respectively.

```
  m ← b∊'<>'
1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 1
```

We can use (inclusive) scan to compute the regions between the tags. First we run scan with "not equals" operation, obtaining the mask:

```
  ≠\m
1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0
```

this mask excludes the last symbol of each tag, which we can fix by disjunction with the original mask:

```
  m∨≠\m
1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1
```

Now, if we want to remove the tags, we have to take negation of the above mask and leave all the letters in the string that correspond to values 1:

```
  b/⍨~m∨≠\m
Hello world
```

As it can be seen, the use of scan made it possible to compute the mask in parallel.

Application (permuting indices). Another example of using scan is when computing index permutations of some array. For example, consider an array stored in the variable *a*:

```
  a ← 1 6 0 7 9 8 1 7 6 3
```

and the mask that results from computing some predicate:

```
  m ← a > 3
0 1 0 1 1 1 0 1 1 0
```

Store negation of the mask in *M*:

```
  M ← ~m
1 0 1 0 0 0 1 0 0 1
```

We can use scan with plus, to enumerate the elements that satisfy the predicate:

```
  m1 ← m∧+\m
0 1 0 2 3 4 0 5 6 0
```

if we compute the maximum of m1, the remaining indices in the permutation can be computed with:

```
  m2 ← M ∧ (⌈/m1) + +\M
7 0 8 0 0 0 9 0 0 10
```

we scan the mask negation, then we add the maximum, and we remove the artifacts of scan. Finally, we compute the permutation with disjunction of m1 and m2:

```
  ix ← m1 ∨ m2
7 1 8 2 3 4 9 5 6 10
```

This assumes 1-based indexing, however if were to compute non-inclusive scan we were to get the answer for 0-based case.

For many more examples on how to use scans, refer to [1].

Builtin. Many array and list languages find scan to be an important primitive that deserves a separate built-in operation. For example, in APL [8] scan is expressed with backslash:

```
  +\1 2 3 4 5
1 3 6 10 15
```

The plus (+) is a binary operation that we use, and [1, 2, 3, 4, 5] is the array we scan over. By default, APL computes inclusive scan. Exclusive scans can be computed by prepending 0 and dropping the last element:

```
  0,¯1↓+\1 2 3 4 5
0 1 3 6 10
```

Similarly to APL, Futhark [7] introduces scan as a primitive in the same way:

```
scan (+) 0 [1, 2, 3, 4] == [1, 3, 6, 10]
```

In MPI [4], non-inclusive scan is a primitive that can be used when reducing results computed by individual threads.

Introducing scan as a built-in gives a lot of freedom to the compiler engineers. The implementation of the primitive can be tuned to the particular needs such as hardware, array sizes, and many others. Unfortunately, by doing so, programmers have very little control on how exactly the operation is going to be executed.

Algebraic Data Types. Many functional languages do not offer built-in support for multi-dimensional arrays. Data types are often defined inductively, and operations are expressed by means of recursive traversals. While this approach keeps the underlying mechanisms compact there is a following catch. Recursive traversals have to follow the inductive structure of data. For example, consider two different ways to specify lists in Haskell:

```
data Llist a = Lnil | Lcons a (Llist a)
data Rlist a = Rnil | Rcons (Rlist a) a
```

Consider now two encodings of the list [3, 4, 5]:

```
l = Lcons 3 $ Lcons 4 $ Lcons 5 Lnil
r = Rcons (Rcons (Rcons Rnil 3) 4) 5
```

These lists are isomorphic, and it is easy to write a function that converts between the two representations. However, these two encodings of a list offer different traversal orders when eliminator follow the structure of the type. For `Llist` it is natural to traverse elements left-to-right and for `Rlist` right-to-left. Consider adding natural numbers $1, 2, \ldots$ to the list elements in the order of traversal:

```
lfold Lnil n        = Lnil
lfold (Lcons x l) n = Lcons (x+n) $ lfold l $ n+1

rfold Rnil n        = Rnil
rfold (Rcons l x) n = Rcons (rfold l $ n+1) $ x+n
```

Function applications `lfold l 1` and `rfold r 1` evaluate to:

```
Lcons 4 $ Lcons 6 $ Lcons 8 Lnil
Rcons (Rcons (Rcons Rnil 6) 6) 6
```

Note that traversals in the other directions are surely expressible in both cases. However, it may take more work than simply unfolding the constructors.

When modeling multi-dimensional arrays, the shape (a list of natural numbers) has to become the index of the array type. This means that the choice of representation of shapes would affect what sub-arrays are "naturally" accessible. In case of `Llist` we get access to the first axis of the shape; in case of `Rlist` — to the last one.

Recall that in case of scan, the top-down scan traverses the shape left-to-right, and in the bottom-up case right-to-left. In [3] the authors use two different data structures to represent arrays of left-nested and right-nested shapes so that top-down and bottom-up scans could be expressed naturally:

```
data Nat = Z | S Nat
data P a = P a a

data Td :: Nat -> * -> * where
  Ld :: a -> Td Z a
  Bd :: P (Td d a) -> Td (S d) a

data Bu :: Nat -> * -> * where
  Lu :: a -> Bu Z a
  Bu :: Bu d (P a) -> Bu (S d) a
```

The `Td` type corresponds to top-down arrays/trees and `Bu` to bottom-up ones. As authors are only interested in perfect binary trees (arrays of shape $[2, 2, \ldots]$) they do not keep the elements of the shape, they only store the rank of the array (first argument to `Td` and `Bu` types). The key difference lies in constructing arrays of dimensions that are greater than zero. For a $d+1$-dimensional array, `Td` gives access to two d-dimensional sub-arrays; whereas `Bu` gives access to all the pairs. These structures guide the traversal over the array shape in the desired order.

Generally, using data structures to guide recursion is a very nice and powerful approach. In case of scans, one potentially has to invent a new data structure for every array shape. Multi-dimensional arrays could liberate one from doing this, offering a generic mechanism of shape-based traversals. The reason for this is that shapes offer random access to its components, and any shape permutation gives a rise to the array transpose. Unfortunately, encoding true multi-dimensional arrays that support rank-polymorphism, non-static shapes and guarantee lack of out-of-bound indexing is a really difficult problem that lies at the boundary of what Haskell type system can offer. Languages with full dependent types are clearly capable of capturing such properties. Unfortunately, dependent types put a lot of burden on programmers requiring them to write explicit proofs.

NESL approach. In [2] authors demonstrate an elegant list-based formulation of the scan algorithm:

```
function scan (a) =
  if #a == 1 then
    [0]
  else
    let e = even_elts(a);
        o = odd_elts(a);
        s = scan({e + o: e in e; o in o})
    in interleave(s,{s + e: s in s; e in e});
```

This can be immediately translated to array-based or list-based languages. All the operations computing even and odd elements, adding two arrays and interleaving two arrays are concurrent. The only synchronisation is happening at the recursive call of `scan`; so the algorithm implements the bottom-up version of scan.

Blelloch (single array). In the formulations of scan presented in sections 3–6 we always use a separate array m to store carries. In general, this represents a situation when recursive calls have to carry some state. Such states can be of complex types, and require non-trivial update operations. Our scans show that when recursion happens over array shapes, state updates can happen concurrently by keeping all the states in the array of a suitable shape.

In case of scan, the element type of the result and the type of the state coincide. Therefore, we can try to store the state and the result in the same array. This technique is demonstrated in [1] and can be expressed in SaC as follows.

Firstly, we iterate over the array shape right-to-left, and for each $a[iv]$ we compute the sum of carries of its subarrays at the first axis: $last(a[iv][0]) + last(a[iv][1]) + \cdots$. We know that we will find carries at the last (highest) index by inductive assumption. The base of the induction is a scalar, where the last element is trivially the last element. We store this result in the last element $a[iv][i-1]$ (the element at the largest index within its shape). We call this pass `upprop` and we lift the sum of last elements in the function `sum_max`:

```
int[*] upprop(int[*] a) {
  for (k = 1; k <= dim(a); k++) {
    i, o = takedrop(shape(a), -k);
    a = { iv -> a[iv][i-1] = sum_max(a[iv]) | iv < o };
  }
  return a;
}
```

Next, we propagate updates by iterating sub-arrays in the decreasing order (traversing the shape left to right). Each iteration computes the scan over last elements of each sub-array. We lift this computation in the function `last_fold`. The initial last element of the entire array a is set to zero.

```
int[*] downprop(int[*] a) {
  a[shape(a)-1] = 0;
  for (i = 0; i < dim(a); i++)
    a = { iv -> last_fold(a[iv]) | iv < take([i], shape(a)) };
  return a;
}
```

The `last_fold` function is a sequential scan over the first axis of the array a, where only last elements of each sub-array are updated:

```
int[*] last_fold(int[*] a) {
  o, i = takedrop(shape(a), 1);
  s = last(a);
  for (k = 0; k < o[0]; k++) {
    t = last(a[k]);
    a[k][i-1] = s;
    s += t;
  }
  return a;
}
```

Finally, the scan can be expressed by composing `downprop` and `upprop`:

```
int[*] scan(int[*] a) {
  return downpop(upprop(a));
}
```

While we have managed to blend the state and the result into the same data structure, the specification became more complex. We have traded space for the necessity to traverse the array twice. While this did not increase the complexity in terms of big O notation, we doubled the amount of synchronisation points. We lost in expressiveness when updating carries in the chosen sub-arrays. Such an operation requires to update a single element of each sub-array that is located at its maximal index. Unfortunately, there is no reshape that would bring such elements to the front axis of the array.

8 Conclusions

In this paper we have demonstrated how rank-polymorphic array programming can be used to specify parallel algorithms at the example of scan. The algorithm updates certain parts of the data structure in a recursive manner and propagates some state through recursive calls. When input data to such algorithms can be organised in multi-dimensional arrays, the structure of the shape can be used to guide recursion over sub-arrays in a non-trivial way.

The shape of the array is a list of natural numbers, and it prescribes canonical arrangement of hyperplanes of the given array. Shape changes that preserve the number of array elements (product of the shape) lead to reversible rearrangements (often called transposes) of the canonical hyperplanes within the array. In practice, this means that we can transpose the array so that desired elements move to the "front" (or back) of the array, then perform the operation, and then transpose the array back.

At runtime, such transposes are typically very cheap. This is because, at runtime, all the arrays are flat, and transposes only change the computation of the offset into the flat representation.

With rank polymorphism and generalised transposes, it becomes possible to express (recursive) traversal through arbitrary permutation of the array shape. Reshapes are only limited by the divisibility of the array length. While reshape does not give access to all the possible array hyperplanes, it gives a powerful tool to experiment with the algorithm specifications without the necessity to introduce additional data structures. Moreover, any such specification does not add extra complexity to data representation at runtime.

While the proposed techniques give a lot of power to explore the design space, there is no precise way to choose the variant that is best-suited for the given hardware. We strongly believe that the proposed algorithm formulation which uses array shapes to guide the traversal is a suitable basis for automatic compiler heuristics when mapping computations to parallel hardware. Unfortunately, we do not yet have satisfying answers on how exactly this process should be organised for the given class of hardware. The search space is incredibly large: distributed memory, GPU caches, memory transfers, threading models, memory consistencies, *etc.*. It is not even clear whether there exists a satisfying heuristics that does not require programmer annotations. To this day, the best implementations are found either by brute-force benchmarking or via applying in-depth knowledge of the underlying hardware. Therefore, a lot of exciting research is still ahead.

For now, we can surely say that array languages do not necessarily help machines to do a better job, but they definitely help humans do their job better.

Acknowledgments

We would like to thank Conal Elliot, Jeremy Gibbons and Peter Braam for productive disucssions that inspired us to write this paper. This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

References

[1] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. Carnegie Mellon University, USA.

[2] Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CS-92-103. Carnegie Mellon University, USA.

[3] Conal Elliott. 2017. Generic Functional Parallel Algorithms: Scan and FFT. *Proc. ACM Program. Lang.* 1, ICFP, Article 7 (aug 2017), 25 pages. https://doi.org/10.1145/3110251

[4] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. University of Tennessee, USA.

[5] Clemens Grelck. 2005. Shared Memory Multiprocessor Support for Functional Array Processing in Sac. *Journal of Functional Programming* 15, 3 (2005), 353–401. https://doi.org/10.1017/S0956796805005538

[6] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahner. 2005. With-Loop Scalarization – Merging Nested Array Operations. In *Implementation of Functional Languages*, Phil Trinder, Greg J. Michaelson, and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 118–134. https://doi.org/10.1007/978-3-540-27861-0_8

[7] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language.* Ph. D. Dissertation. University of Copenhagen, Universitetsparken 5, 2100 København.

[8] Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA.

[9] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. https://doi.org/10.1007/BFb0055425

[10] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[11] Sven-Bodo Scholz and Artjoms Šinkarovs. 2019. Tensor Comprehensions in SaC. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 13 pages. https://doi.org/10.1145/3412932.3412947

[12] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

[13] Roger Stokes. 15 June 2015. Learning J. An Introduction to the J Programming Language. http://www.jsoftware.com/help/learning/contents.htm. [Accessed 2020/02].