Chapter 6

Algorithm-Design Technique: Contraction

Contraction, an inductive technique for designing parallel algorithms, is probably one of the most important algorithm-design techniques. Like divide-and-conquer algorithms, contraction algorithms involve solving a smaller instance of the same problem, but unlike in divide-and-conquer, there is only one subproblem to solve at a time.

A contraction algorithm for problem P has the following structure.

Base Case: If the problem instance is sufficiently small, then compute and return the answer, possibly using another algorithm.

Inductive Step: If the problem instance is sufficiently large, then apply the following three steps (possibly multiple times).

1. ***Contract:*** "contract", i.e., map the instance of the problem P to a smaller instance of P.
2. ***Solve:*** solve the smaller instance recursively.
3. ***Expand:*** use the solution to solve the original instance.

Contraction algorithms have several nice properties. First, their inductive structure enables establishing correctness and efficiency properties in a relatively straightforward fashion using principles of induction. For example, to prove a contraction algorithm correct, we first prove correctness for the base case, and then prove the general (inductive) case by using strong induction, which allows us to assume that the recursive call is correct. Similarly, the work and span of a contraction algorithm can be expressed as a recursive (inductive) relation, which essentially reflects the structure of the algorithm itself. We then establish the bounds for work and span by using known, well-understood techniques for solving recursive relations as briefly discussed in Chapter 4. Second, contraction algorithms can be efficient, especially if they can reduce the problem size geometrically (by a constant factor greater than 1) at each contraction step and if the contraction. Similarly, if the contraction and expansion steps have low spans, and if the problem size can be decreased by geometrically, then the algorithm will likely have a low span, making it a good parallel algorithm.

In this chapter, we will consider several applications of the contraction technique and analyze the resulting algorithms.

6.1 Example 1: Implementing Reduce with Contraction

As our first example, we describe how to perform reduction on a sequence by using contraction. Recall that the type signature for *reduce* is as fallows.

```
reduce (f: α * α → α) (I: α) (S: α sequence): α
```

where f is an associative function, S is the sequence, and I is the identity element of f. Even though we have defined *reduce* broadly for both associative and non-associative functions, throughout this section, we only consider the case with associative functions.

The crux in designing a contraction algorithm is to design an algorithm for reducing an instance of the problem to a significantly smaller instance of the same problem by performing a parallel contraction step.

Question 6.1. *How can we reduce an instance of the "reduce" problem to a smaller instance of the same problem?*

As it turns out, there are many ways to perform such a contraction. Recall, for example, that we can perform a reduction by using iteration (`iterate`). In fact, iteration technique can be viewed as a way of reducing the problem to a smaller instance of the problem, which is smaller than the original problem only by one element. To see how we might achieve a more significant reduction in the problem size, consider applying the function to consecutive pairs of the input. For example if we wish to compute the sum of the input sequence

$$\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$$

by using the addition function, we would contract the sequence to

$$\langle\, 3, 5, 7, 5 \,\rangle\,.$$

By using this contraction step, we have reduced the input size by (nearly) a factor of two.

Question 6.2. *Can we perform this contraction step in parallel?*

Note also that the contraction step can be performed in parallel, because each pair can be considered independently in parallel.

Having reduced the size of the problem with the contraction step, we next solve the resulting problem by invoking the same algorithm and apply expansion to construct the final result.

Question 6.3. *What computations should the expansion step perform?*

It is not difficult to see that by solving the smaller problem, we have actually solved the original problem, because the sum of the sequence remains the same as that of the original. Thus, the expansion step requires no additional computation.

We can thus express our algorithm as follows; for simplicity, we assume that the input size is a power of two.

Algorithm 6.4 (Reduce with contract)**.**

```
(* Assume for simplicity that $|A|$ is a power of 2 *)
fun reduce_with_contract f A =
  B = $\langle f(A[i], A[i+1]) : 0 \leq i < \lfloor |A|/2 \rangle$ (* Contraction *)
  reduce_with_contract f B
  (* Expansion not needed. *)
```

One nice thing about contraction algorithms is that ther work and span can be written using a recursive relation.

Question 6.5. *What is the work of this algorithm?*

Assuming that the function being reduced over performs constant work, parallel tabulate in the contraction step requires linear work, we can thus write the work of this algorithm as follows.

$$W(n) = W(n/2) + n.$$

It is not difficult to solve this recursive formula to prove that the algorithm therefore performs $O(n)$ work.

Question 6.6. *How about the span?*

Assuming that the function being reduced over performs constant span, parallel tabulate in the contraction step requires constant span, we can thus write the work of this algorithm as follows.

$$S(n) = S(n/2) + 1.$$

The algorithm therefore performs $O(\log n)$ span.

6.2 Example 2: Implementing Scan with Contraction

As an even more interesting example, we describe how to implement the *scan* sequence primitive efficiently by using contraction. Recall that the *scan* function has the type signature

`scan` $(f\texttt{:}\ \alpha{\star}\alpha \rightarrow \alpha)$ $(I\texttt{:}\ \alpha)$ $(A\texttt{:}\ \alpha$ `sequence`$)$ `:` $(\alpha$ `sequence`$\star\alpha)$

where f is an associative function, A is the sequence, and I is the identity element of f. When evaluated with a function and a sequence, *scan* can be viewed as applying a reduction to every prefix of the sequence and returning the results of such reductions as a sequence.

Suppose we are to run, *scan* `'+'`, i.e., "plus scan" on the sequence $\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$. What we should get back is

$$(\langle\, 0, 2, 3, 6, 8, 10, 15, 19 \,\rangle\, , 20)\,.$$

We will use this as a running example.

Based on its specification, a direct algorithm for *scan* is to apply a reduce to all prefixes of the input sequence.

Question 6.7. *What it the work of such an algorithm?*

Unfortunately, this easily requires quadradic work in the size of the input sequence.

Question 6.8. *Do you see what makes this algorithm inefficient? Is there a better algorithm?*

We can see that this algorithm is inefficient by noting that it performs lots of redundant computations. In fact, two consecutive prefixes overlap significantly but the algorithm does not take advantage of such overlaps at all, computing the result for each overlap essentially independently. By taking advantage of the fact that any two consecutive prefixes differ by just one element, it is not difficult to give a linear work algorithm (modulo the cost of the application of the argument function) by using iteration. Such an algorithm may be expressed as follows

$$\overline{\int}^{f\ I}_{x\in A} x = h\left(\overline{\prod}^{g\ (\langle\,\rangle, I)}_{x\in A}\ x\right),$$

where $g((B, v), x) = ((\mathit{append}\,\langle\, v \,\rangle\, B), f(v, x))$
and
$h(B, v) = (\texttt{reverse}\ \texttt{B}), v)$ (*reverse* reverses a sequence).

Question 6.9. *What is wrong with this algorithm?*

This algorithm, while correct, is almost entirely sequential, leaving no room for parallelism.

Question 6.10. *Considering that it has to compute some value for each prefix, can we even perform a* `scan` *in parallel?*

Beyond the wonders of what can be done with scan, an interesting fact about `scan` is that it can be accomplished efficiently in parallel, although on the surface, the computation it carries out appears to be sequential in nature. At first glance, we might be inclined to believe that any efficient algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. It is this apparent dependency that makes `scan` so powerful. Indeed, we often use `scan` when it seems we need a function that depends on the results of other elements in the sequence.

To implement `scan` efficiently using contraction, we need a way to reduce the problem to a significiantly smaller instance of the same problem by applying a contraction step.

Question 6.11. *Any ideas about how we might be able to do this?*

As a starting point, Let's apply the same idea as we used for reduction in Algorithm 6.4 see and how far we might get with that. Applying the contraction step from the `reduce` algorithm described above, we would reduce the input sequence

$$\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$$

to the sequence

$$\langle\, 3, 5, 7, 5 \,\rangle\,,$$

which if recursively used as input would give us the result

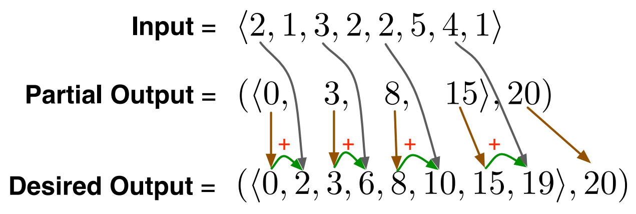
$$(\langle\, 0, 3, 8, 15 \,\rangle\,, 20).$$

Notice that in this sequence, the elements in the first, third, etc., positions are actually consistent with the result expected:

$$(\langle\, 0, 2, 3, 6, 8, 10, 15, 19 \,\rangle\,, 20).$$

The reason for why half of the elements is correct is because the contraction step which pairs up the elements and reduces them, does not affect, by associativity of the function being used, the result at the position that do not fall in between a pair. Thus, what we thus need is an expansion step that can fill those missing items. It is actually quite simple: all we have to do is to compute the missing elements by applying the function element-wise to the elements of the input at odd positions in the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recursive call:


Input = ⟨2, 1, 3, 2, 2, 5, 4, 1⟩
Partial Output = (⟨0, 3, 8, 15⟩, 20)
+
+
+
+
Desired Output = (⟨0, 2, 3, 6, 8, 10, 15, 19⟩, 20)

This leads to the following code. The algorithm we present works for when n is a power of two.

Algorithm 6.12 (Scan Using Contraction, for powers of 2)**.**

```
(* Assume that the length of the sequence is a power of two. *)
function scanPow2 f i A =
  case |A| of
    0 ⇒ (⟨ ⟩, i)
  | 1 ⇒ (⟨ i ⟩, A[0])
  | n ⇒ let
          val A' = ⟨ f(A[2i], A[2i+1]) : 0 ≤ i < n/2 ⟩
          val (r, t) = scanPow2 f i A'
        in
          (⟨ p_i : 0 ≤ i < n ⟩, t), where p_i = { r[i/2]  even(i);  f(r[i/2], A[i−1])  otherwise. }
        end
```

Question 6.13. *What is the work and span of the contraction algorithm for* `scan`*?*

Let's assume for simplicity that the function being applied has constant work and constant span. We can write out the work and spane for the algorithm as a recursive relation as follows.

$$W(n) = W(n/2) + n, \text{ and } S(n) = S(n/2) + 1,$$

because 1) the contraction step which tabulates the smaller instance of the problem performs linear work in constant span, and 2) the expansion step that constructs the output by tabulating based on the result of the recursive call also performs linear work in constant span.

These recursive relations should look familiar. Indeed, they are the same as those that we ended up with when we analyzed the work and span of our contraction-based implementation of `reduce`. They yield $O(n)$ work and $O(\log n)$ span.