Refining the Parallel Prefix Sum Algorithm

Ernie Heyder
Wittenberg University
105 West McCreight
Springfield, Ohio 45504
s13.eheyder@wittenberg.edu

ABSTRACT
This paper focuses on the prefix sum algorithm. The goal of the paper is to develop a parallel prefix sum algorithm that is faster than the sequential algorithm. Takes parts of an algorithm created by Dan Grossman [1] and combined them with a new step in the algorithm that was developed by the author to create a different approach to solving the prefix sum algorithm. There are concurrency problems that can occur when trying to write this algorithm. The solution to these problems will be given as they appear, along with a new algorithm developed by the author.

1. INTRODUCTION

Inspiration for this paper comes from Dan Grossman's paper on parallel algorithm [1]. In particular, the prefix sum algorithm that he describes will be analyzed here, considering many issues in much greater detail. The idea for the prefix sum algorithm was taken from Dan Grossman's Java code [1], and was then developed, through extensive testing and modifications.

The overall goal is to create a prefix sum that will be significantly faster than the simple sequential algorithm. The prefix sum algorithm is used in an implementation of the quick sort algorithm and *pack* operations. *Pack* operations solve problems like: "Given an array input, produce an array output containing only those elements of input that satisfy some property, and in the same order they appear in input" [1]. Along with applications in algorithms the prefix sum algorithm demonstrates important issues in parallel programming.

The paper will start out by introducing what the Prefix sum algorithm is and defining the two different types of summing that can occur. Presented next will be the sequential, recursive, and parallel algorithms, concluding with results on how the parallel function compares to the sequential function.

2. PREFIX SUM ALGORITHM

This section starts by introducing the exclusive and inclusive prefix sum. It then analyzes the difference between using an in-place algorithm and creating one or more new arrays of non-constant size. We will see in all cases that the in-place algorithm is better than creating additional arrays.

Prefix sum takes an array and computes a running total (or prefix) of every element in the given array. There are two types of prefix sums. One behaves as follows:

Original: [5, 6, 7, 8]
Prefix Sum: [0, 5, 11, 18]

This is known as an *exclusive prefix sum*, in which position *i* in the prefix sum array is the sum of all the values in the original array up to ***but not including*** position *i*. The second prefix sum behaves as follows:

Original: [5, 6, 7, 8]
Prefix Sum: [5, 11, 18, 26]

This is known as an *inclusive prefix sum*, in which position *i* in the prefix sum array is the sum of all the values in the original array up to ***and including*** position *i*. The two sums are related by a simple shift of the values in the array. To change the inclusive prefix sum into the exclusive, shift all of the values to the right in the array, and define the first value as 0.

One option to consider when designing any algorithm is whether additional structures (e.g. arrays) will be required that are larger than Θ(1) in the problem size. Algorithms that require no such structures, but rather modify the original structures as needed, are called *in-place* algorithms. Algorithms that require additional structures often involve many expensive memory allocation operations. Typically parallel algorithms are used on very large amounts of data, meaning that memory allocation is an important issue to consider.

2.1 Sequential Algorithm

The sequential algorithms for both the exclusive and inclusive prefix sum will be presented in this section. The sequential algorithm is the base case that later algorithms will be tested against. The easier of the two algorithms is the inclusive prefix sum as you will see below. As mentioned before both of these algorithms are in-place algorithms, meaning they will mutate the array. The Java code for the inclusive prefix sum is as follows:

```
1 inclusivePrefixSum(array) {
2       for ( i = 1; i < array.length; i++) {
3               array[i] += array[i-1];
4       }
5 }
```

The Java code for the exclusive prefix sum is:

```
1 exclusivePrefixSum(array) {
2       tempOld = array[0];
3       array[0] = 0;
4       for ( i = 1; i < array.length; i++) {
5               tempNew = array[i];
6               array[i] = tempOld + array[i-1];
7               tempOld = tempNew;
8       }
9 }
```

2.2 Recursive Algorithm

The recursive algorithm presented here is a slight modification of the algorithm taken from Dan Grossman [1]. This algorithm works in two parts; the first part is called the up-pass and the second is called the down-pass. The recursive algorithm does not create a speedup from the sequential, but is rather an intermediate step that is used to get to the parallel algorithm. The exclusive and inclusive algorithm will both use the same up-pass, but will have slightly different down-pass algorithms.

2.2.1 Up-pass

The up-pass is a recursive algorithm that can be thought of as a two stage process, the breakdown and the buildup. In the following Java code, the breakdown is simply the recursive calls of the function. The buildup is the popping of the implicit stack created by those recursive calls.

```
1  upPass(array, lo, hi) {
2       //Base Case
3       If (hi-lo <= 0) {
4               return array[hi];
5       } else {
6               upPass(array, lo, (hi+lo)/2);
7               right = upPass(array,
                        (hi+lo)/2+1, hi);
8
9               array[hi] = array[(hi+lo)/2] +
                        right;
10              return array[hi];
11      }
12  }
```

The breakdown will take an array and break it down into each of its elements individually. In the diagram below the array starts at size 8 then is broken into two arrays of 4 and so on. Note that the "splitting" of the array depicted here is simply for visualization purposes. In practice, no new arrays are created. Rather, the algorithm simply references different ranges of the original array using an upper and a lower index.

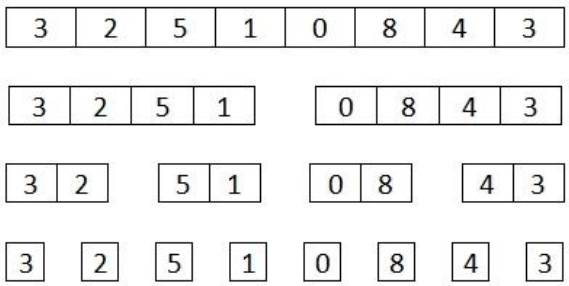

Figure 2.1 Breakdown Step

For the buildup, which involves popping of the recursive stack, it starts with the individual element and returns that element. This step is the base case for our recursive function. At every step other than the base case, we add the two numbers that are given to us by the left and the right recursive call, and store that value at the highest position in the current range. The following figure illustrates the process. Again, though the array appears to be split, this is simply a visual aid. The "splitting" is actually handled through the *lo* and *hi* parameters. Remember that both the exclusive and inclusive prefix sum use the same up-pass.

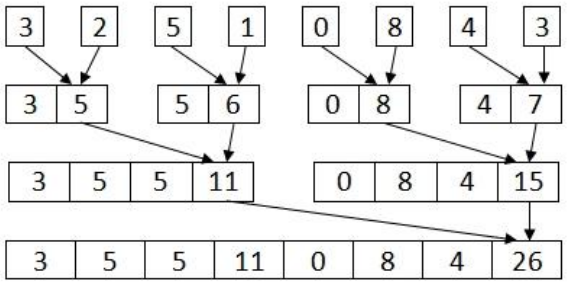

Figure 2.2 Buildup Step

2.2.2 Exclusive Down-pass

The down-pass takes the array that is modified in the up-pass and creates the final prefix sum. The down-pass is the part of the algorithm that determines if the function is an exclusive or inclusive sum. First I will show the exclusive algorithm, since it is easier to understand. The Java code for the exclusive sum algorithm is as follows.

```
1  exclusiveDownPass(array, lo, hi) {
2          //Recursive Case
3          If (hi-lo != 0) {
4                  temp = array[hi];
5                  array[hi] += array[(hi+lo)/2];
6                  array[(hi+lo)/2] = temp;
7
8                  exclusiveDownPass(array, lo,
                           (hi+lo)/2);
9                  exclusiveDownPass(array,
                           (hi+lo)/2+1, hi);
10         }
11 }
```

For the exclusive sum the down-pass has to do less work than the up-pass. It only has the breakdown step. There is no need for the buildup step because all of the work is done before the recursive calls are made. Just like the up-pass, the array is broken down into different ranges with each recursive call; this will be represented by the blocks in the diagram that follows. As before, these are just ranges of the original array, not new arrays. The first step before the down-pass is to change the last value in the array to zero. This step is very important for the algorithm to work.

The first step of the down-pass itself is to store the value at the high position in the range (exclusiveDownPass line 4). Then add the value at the halfway point to the high position's value (exclusiveDownPass line 5, Figure 2.3 short dashed black arrow). The halfway point is determined by the equation (hi+lo)/2, where the hi and lo value are the upper and lower index of the range. The division used here is integer division. This value is stored in the high position (exclusiveDownPass line 5, Figure 2.3 solid black arrow). Next, take the earlier value stored from the high position and replace the value at the half way point with it (exclusiveDownPass line 6, Figure 2.3 long dashed black arrow). Repeat this process until there is just one element in the range.

2.2.3 Exclusive Prefix Sum: Summary

The Java code for the exclusive prefix sum is as follows:

```
1 exclusivePrefixSum(array) {
2       upPass(array, 0, array.length-1);
3       array[array.length-1] = 0;
4       exclusivedownPass(array, 0,
                array.length-1);
5 }
```

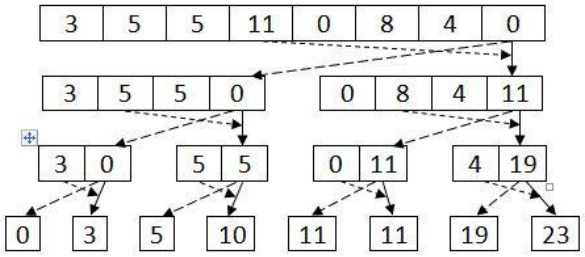

Figure 2.3 Exclusive Down-pass

At this point the array is the exclusive prefix sum of the original array.

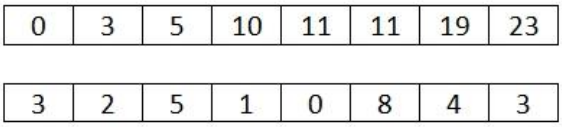

Figure 2.4 Exclusive Down-pass Results

2.2.4 Inclusive Down-pass

Next I will look at the algorithm for the inclusive prefix sum. Since there is a relationship between the inclusive and exclusive prefix sum, we could use it here to make a slight change to the exclusive algorithm. To do this, simply store the last value in the array after the up-pass. Next add a for loop after the exclusive down-pass that shifts all the elements to the left. Then put the stored value at the end of the array to solve this problem. But this would be very inefficient because we have already iterated through the array once and now have to do it again. On large amounts of data this will cause a large slowdown in the speed.

Instead we will change the base case. The base case will now store the value to the left of where it did in the exclusive algorithm. The new Java code is:

```
1  inclusiveDownPass(array, lo, hi) {
2       //Base Case
3       If (hi-lo == 0) {
4             If (hi != 0) {
5                   array[hi-1] = array[hi];
6             }
7       } else {
8             temp = array[hi];
9             array[hi] += array[(hi+lo)/2];
10            array[(hi+lo)/2] = temp;
11
12            inclusiveDownPass(array, lo,
                    (hi+lo)/2);
13            inclusiveDownPass(array,
                    (hi+lo)/2+1, hi);
14      }
15 }
```

The following diagram shows how the inclusive down-pass works. It is exactly the same as the exclusive down-pass except it has a different base case that is added to the end. The 26 is added using the following inclusive prefix sum section.

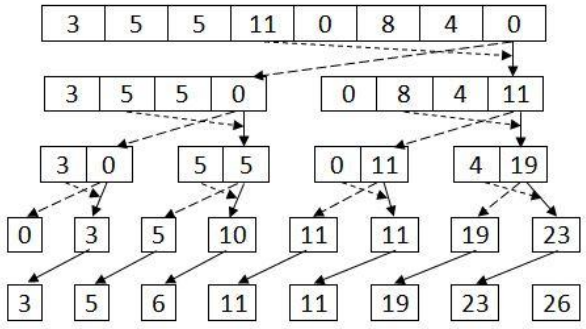

Figure 2.5 Inclusive Down-pass

2.2.5 Inclusive Prefix Sum: Summary

Some minor changes to the prefix sum are also required. The altered Java code looks like this:

```
1 inclusivePrefixSum(array) {
2       upPass(array, 0, array.length-1);
3       int temp = array[array.length-1];
4       array[array.length-1] = 0;
5       downPass(array, 0, array.length-1);
6       array[array.length-1] = temp;
7 }
```

2.3 Parallel Algorithm

This section demonstrates how to transform a recursive algorithm into a parallel algorithm.

First there will be a brief explanation of the difference between the Java Thread class and Fork/Join framework. Then the process used to convert a recursive algorithm into a parallel algorithm, using the Fork/Join framework, will be presented. This process is used on the inclusive prefix sum algorithm. A new algorithm is then developed for the inclusive prefix sum algorithm.

Before the creation of the Fork/Join framework, Java programmers wrote multi-threaded programs with the Thread class (or closely-related Runnable interface). The problem with the Thread class is it has a lot of overhead associated with the creation of each thread. This made it so that parallel algorithms requiring large numbers of threads were unlikely to achieve a speedup. The Fork/Join framework changes that. The Fork/Join framework is from the new JDK 7 release on July 7, 2011. It does not eliminate the overhead completely, but significantly lessens it. The three classes I use in the algorithms below are: ForkJoinPool, RecursiveAction, and RecursiveTask.

To create the parallel algorithm is fairly simple for the exclusive prefix sum. It is just a matter of rewriting the recursive algorithm above using the Java Fork/Join framework. An outline of this process is provided below:

1. Change the recursive methods into classes that implement either RecursiveTask<E> or RecursiveAction
    a. The parameters now become parameters in the constructor
    b. The body of the recursive method is put into the compute method
    c. Change the recursive calls so that they create a new instance of the class
    d. Add in the fork, compute and join methods so that the method works properly
2. Create a instance of the ForkJoinPool class called fjPool
3. Change the method that calls the recursive methods so that it now creates a new instance of the class and runs it using fjPool.invoke(*with the method as a class inserted here*)

Before further considering this process in the prefix sum problem, we will consider a much simpler problem as an example: summing all numbers in an array. This example is discussed in detail in Dan Grossman's paper [1]. Starting with the recursive algorithm for summing an array:

```
1  sumArray(array) {
2        return sumArrayHelper(array, 0,
         array.length-1);
3  }
4
5  sumArrayHelper(array, lo, hi) {
6        if (hi-lo == 0) {
7              return array[hi];
8        } else {
9              left = sumArrayHelper(array,
                     lo, (hi+lo)/2);
10             right = sumArrayHelper(array,
                     (hi+lo)/2+1, hi);
11
12             return (left + right);
13       }
14 }
```

Using the steps above the code is rewritten to look like the code below:

```
1  fjPool = new ForkJoinPool();
2
3  sumArray(array) {
4        return fjPool.invoke(new
               SumArrayHelper(array, 0,
               array.length-1));
5  }
6
7  class SumArrayHelper(array, lo, hi)
       extends RecursiveTask<Integer>{
8        int[] array;
9        int lo, hi;
```

```
10
11      SumArrayHelper(array, lo, hi) {
12              this.array = array;
13              this.lo = lo;
14              this.hi = hi;
15      }
16
17      compute() {
18              if (hi-lo == 0) {
19                      return array[hi];
20              } else {
21                left = new
                        SumArrayHelper(array,
                        lo, (hi+lo)/2);
22                right = new
                        SumArrayHelper(array,
                        (hi+lo)/2+1, hi);
23
24                left.fork();
25                rightAns = right.compute();
26                leftAns = left.join();
27
28                return (leftAns + rightAns);
29              }
30      }
31  }
```

2.3.1 Concurrency Problem: Inclusive Prefix Sum

The inclusive algorithm has a few concurrency issues when changed into a parallel algorithm using the steps above. The part that is particularly interesting is: array[hi-1] = array[hi]; This line's actual execution can be represented in the following manner:

```
temp = array[hi];
array[hi-1] = temp;
```

Suppose we would like to find the inclusive prefix sum of the array [1, 2, 3, 4]. First, the up-pass will have changed this array to be [1, 3, 3, 10]. Then the down pass will recursively call itself until it has four threads running at once on the array [0, 1, 3, 6]. Each thread has a *hi* value for each position in the array. The thread with *hi* equal to zero will do nothing. The rest will attempt to execute the line of code above. This is where the problem occurs. If all of the threads execute in the order of the *hi* value (the order of 1, 2, 3), then there is no problem. However, since the threads are running in parallel this is not guaranteed. So if the threads execute in any other order we have a problem. For instance, if the order of hi values was 2, 3, 1 then the resulting array would be [3, 3, 6, 6]. The final output for the prefix sum will then be [3, 3, 6, 10]. This is not the correct answer; the inclusive prefix sum algorithm needs to be rewritten.

2.3.2 Exclusive Sum Algorithm

The exclusive prefix sum algorithm does not have the same issues as the inclusive prefix sum because it does not attempt to modify elements outside of its range of reference. The *hi* value in the range is the only value being changed in the exclusive prefix sum algorithm.

2.3.3 Resolving Inclusive Prefix Sum Concurrency Problem

While there is some published literature on many of the prefix sum issues discussed so far in this paper, there does not appear to be any discussion on resolving these concurrency issues in the down-pass of the inclusive prefix sum algorithm. Thus the following down-pass algorithm was developed. Again the same up-pass is used as in the exclusive prefix sum algorithm. Along with splitting the array in half, an extra value is passed recursively to calculate the correct prefix sum. The value that is passed in will be referred to as *num*, so that it is easier to keep track of what is going on. In the diagram the *num* is always in bold. To start the algorithm set *num* to the high value in the array. The down-pass uses the following Java code:

```
1  inclusiveDownPass2(num, array, lo, hi) {
2          //Base Case
3          If (hi-lo != 0) {
4                  temp = num – array[(hi+lo)/2];
```

```
5               mid = array[(hi+lo)/2];
6               array[(hi+lo)/2] = array[hi] –
                        temp;
7
8               downPass(mid, array, lo,
                        (hi+lo)/2);
9               downPass(temp, array,
                        (hi+lo)/2+1, hi);
10      }
11 }
```

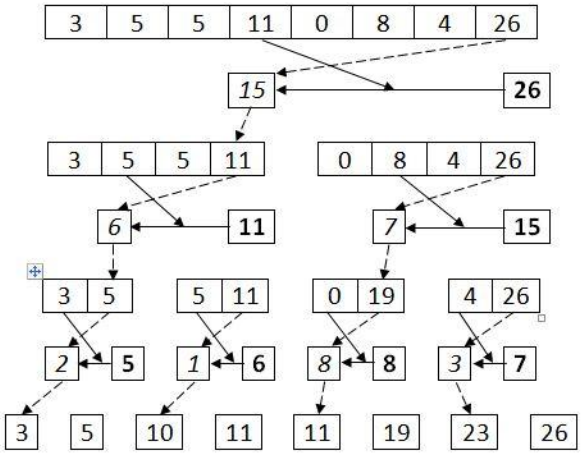

Figure 2.6 Inclusive Down-pass 2

In the above diagram the *temp* value is always in italics. The solid black lines represent how the temp value is calculated (inclusiveDownPass2 line 4). The dashed black lines represent how the mid position in the array is calculated (inclusiveDownPass2 line 6). The function will keep evaluating until it has one item in its range then it stops. At this point the array is an inclusive prefix sum of the original array.

3	5	10	11	11	19	23	26

3	2	5	1	0	8	4	3

Figure 2.7 Inclusive Down-pass 2 Results

The Java code for the inclusive prefix sum is as follows:

```
1 prefixSum(array) {
2       upPass(array, 0, array.length-1);
3       downPass(array[array.length-1],
        array, 0, array.length-1);
4 }
```

2.4 Cutoff Algorithm

Just making an algorithm run in parallel does not automatically provide a speedup. The overhead associated with creating so many new threads makes the parallel algorithm slower than the sequential one, even with the use of the lightweight threads of the Java Fork/Join framework. To remedy this, a *sequential cutoff* is inserted at 500 elements. A *sequential cutoff* is a base case that stops the creation of new parallel threads in an algorithm and continues the algorithm in serial.

The sequential cutoff for the exclusive algorithm is at 500 elements for both the up and down-pass. When the cutoff is reached the algorithm will simply call the recursive version of the algorithm. This will remove the overhead of creating new threads.

For the inclusive algorithm there is a slightly quicker algorithm that requires us to change the up and down-pass algorithm slightly. For the up-pass the cutoff will make it so that the bottom elements will hold the sum of 500 elements each. This will not change their position in the array, just the process used to sum them. The sum of the 500 elements will be done iteratively and will not modify any of the other elements other than the rightmost element (the 500th element). This element acts like the parent value for all of the 500 elements. Each of the parent values created like this will be put back into the recursive algorithm that was already developed and everything will function like normal.

Here is an example of what the process will be for the array of 12 and a cutoff of 3. The breakdown has not been illustrated since it is similar to the last one. The buildup starts with iteratively summing the bottom 3 elements in each of the ranges. This is illustrated in the very first step of the diagram. Every step after that is done recursively, by the popping of the stack.

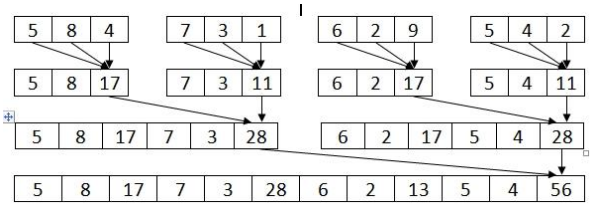

Figure 2.8 Cutoff Up-pass

The following is a modified version of the up-pass so that it uses a cutoff as the base case.

```
1  upPass(array, lo, hi) {
2        //Base Case
3        If (hi-lo <= 500) {
4                int sum = 0;
5                for (int i = lo; I <= hi; i++) {
6                        sum += array[i];
7        }
8        array[hi] = sum;
9        return array[hi];
10       } else {
11               upPass(array, lo, (hi+lo)/2);
12               right = upPass(array,
                         (hi+lo)/2+1, hi);
13
14               array[hi] = array[(hi+lo)/2] +
                         right;
15               return [hi];
16       }
17 }
```

In order for the cutoff to work properly the down-pass cutoff must be the same as the up-pass cutoff. The cutoff can either be used in both passes or not at all. All of the values under the cutoff will be created iteratively, using a for loop, that just adds the previous term to the current term. This is why we did not do this in the up-pass step earlier. The only case that has to be done differently is the first element in the array; this element does not change at all. Here is the Java code for the down-pass with the cutoff:

```
1  downPass(array, lo, hi) {
2        //Base Case
3        if (hi-lo <= 500) {
4                if ( lo != 0) {
5                        array[lo] += array[lo-1];
6                }
7                for (int i = lo+1; i< hi; i++) {
8                        array[i] += array[lo-1];
9                }
10       } else {
11               int temp = num –
                         array[(hi+lo)/2];
12               int mid = array[(hi+lo)/2];
13               array[(hi+lo)/2] = array[hi] –
                         temp;
14               downPass(mid, array, lo,
                         (hi+lo)/2);
15               downPass(mid, array,
                         (hi+lo)/2+1, hi);
16       }
17 }
```

3. EXPERIMENTAL SETUP

Here is some information about the computer I used to run the programs on. The computer is a Dell N5010. It has an Intel® Core™ i3 CPU, 2.53 GHz. 4 GB of RAM (3.8 GB usable). Operating System is a 64-bit.

4. RESULTS

Now it is time to analyze the final function against what we started with. The goal of this paper was to create an inclusive prefix sum algorithm that is more efficient than the sequential algorithm.

Note that the overhead involved in running the algorithm in parallel is logarithmic in the problem size, but constant in the number of processors. That is, for a given problem size, the number of parallel threads created remains constant regardless of the actual number of processors available. In addition, the number of parallel threads created is quite large, such that even hundreds of processors could be effectively applied to this algorithm with no increase in the cost of parallelization. Thus an increase in processors applied to this problem leads to an inevitable speedup.

Due to the limited number of processors available for use in these tests, I was unable to reach the point at which the speedup of parallelism overcomes the constant cost of parallelism over the sequential algorithm.

The cutoff of 500 elements was chosen to follow the process used by Dan Grossman. In testing I was unable to find a speed-up in any cases that I tried so I decided to leave the cutoff at 500 to mirror what has already been tested.

4.1 Applications

The prefix sum inclusive algorithm can be used in a few parallel algorithms. One algorithm is called pack. The pack solves the problem: “given an array input, produce an array output containing only those elements of input that satisfy some property, and in the same order they appear in input.” [1] For example one pack function would takes an array and a number, and then returns an array of all values greater than that number. Another use of the prefix sum is in the parallel quick sort algorithm.

5. REFERENCES

[1] Grossman, Dan. "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency." Diss. University of Washington, 2011. Web. <http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf>.