Topic : Parallelizing QuickSort using CilkPlus.
Students : Hao Hu (3576231), Abhijit Taware(3590004).
Supervisor : Eric Aubanel (Parallel Programming course CS 6025).

Abstract

QuickSort is efficient sorting algorithm and is mainly a divide and conquer algorithm. The algorithm divides the list of elements in two parts and recurse on each half. The partitioning is done by selecting a pivot element. All the elements smaller than or equal to the pivot goes to one partition and rest goes to the other. This division eventually ensures that the array is sorted. The pseudo code can be as follows -

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p - 1 )
      quicksort(A, p + 1, hi)
```

As a first deliverable we are focusing on implementing the partition efficiently. Parallelizing the recursive calls should be easy using cilk_spawn and cilk_sync.

E.g.
A = [7, 4, 5, 3 ,19, 10,11, 9, 1]
If we select first element i.e. 7 as pivot then the the two partitions will be as follows -
[7, 4, 5, 3, 1] and [19, 10, 11, 9]

Partitioning it using sequential algorithm has O(n) complexity and would take longer for a very huge value of n. We can use prefix sum to partition the array in parallel. The trick is to keep additional indicator array B which encodes information about whether particular element in array A is smaller or not. Array B will have value 1 for if A[i] <= pivot, 0 otherwise. Now doing prefix sum of B gives us way to partition array A. The array B would be as follows for above example -
B = [1,1,1,1,0,0,0,0,1] . It prefix sum would be → **B_sum** = [0,1,2,3,3,3,3,3,4]

The B_sum gives us indices for first half of the partition, i.e. the left half would be -
L = [7, 4, 5, 3, 1] , note we have to use indices from B_sum for the elements which have value 1 in array B.

As a part of first deliverable we have to implement exclusive prefix sum. We have done it using Blelloch scan algorithm mentioned in the book and at the following link - https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

Algorithm

This scan algorithm works in two phases viz. upsweep and downsweep. The upsweep is a regular reduction as given below.

```
Procedure up-sweep(a)
for d from 0 to (lg n) − 1
  in parallel for i from 0 to n − 1 by 2 d+1
    a[i + 2 d+1 − 1] ← a[i + 2 d − 1] + a[i + 2 d+1 − 1]
```

If we have array A = [1, 2, 3, 4, 5, 6, 7, 8], following steps show the complete up-sweep

```
1  2  3  4  5  6  7  8
   3     7     11    15
         10          26
                     36
```

Dow sweep starts by setting an identity element i.e. last element is set to zero. The pseudo code as follow.

```
procedure down-sweep(a)
  a[n − 1] ← 0
  for d from (lg n) − 1 downto 0                  % Set the identity
    in parallel for i from 0 to n − 1 by 2^d+1
          t ← a[i + 2 d − 1]                      % Save in temporary
          a[i + 2 d − 1] ← a[i + 2 d+1 − 1]       % Set left child
          a[i + 2 d+1 − 1] ← t + a[i + 2 d+1 − 1] % Set right child
```

```
1  2  3  4   5   6  7   8
   3     7       11     15
         10             26
                        36
         10             0   ← Setting identity element and using values from upsweep.
          0             10  ← setting left and right children
   3      0       11     10
   0      3       10     21
1  0  3   3   5  10  7  21
0  1  3   6  10  15 21  28
```

We have used Cilk plus for this implementation. We are using array notations in cilk plus to vectorize the code. Detailed implementation can be found in the source code provided with the report.

The challenges and solutions:

Above implementation assumes the array size can be expressed as power of two. However in practice we encounter different array sizes and hence need to modify the algorithm to handle this case as well. We are proposing two different implementations for handling arrays of different sizes -

1. Zero filling the array to match next smallest value which is power of 2
2. Implement the Blelloch scan in the opposite direction.

1. Zero filling the array

Suppose we have an array with 5 elements as follows -
1 2 3 4 5 , we extend it to an array with 8 elements and zero fill the additional elements.
1 2 3 4 5 0 0 0
Now the blelloch scan can proceeds as usual. The downside is we have allocate space for extra elements.

2. Reverse Blelloch scan

In an attempt to avoid zero filling, we have tried a different approach to Blelloch scan which can be described as follows.

- Reverse the original array
- Start up sweeping by adding element on the right to element on the left
    - A[left] += A[right]
- We control the number of iterations, so don't need additional zero filling.
- Downsweep also happens in reverse direction.
- However we get the prefix sum array in reversed direction and we reverse the final result again.

Following example shows this simple trick

14	9	3	11	8	← original array
8	11	3	9	14	← reversed array
19		12		14	← A[left] += A[right]
31				14	
45					
0	11	12	9	14	← setting A[0]=0 & reusing results from upsweep
14				0	
14		12	9	0	
26	11	14	9	0	
37	26	23	14	0	← this is reversed prefix sum
0	14	23	26	37	← reverse again to get the prefix sum

The key thing here is to do the scan in the reverse direction. By changing the direction of Blelloch scan we avoid zero filling. The Figure 1 below illustrates the task graph for this algorithm.

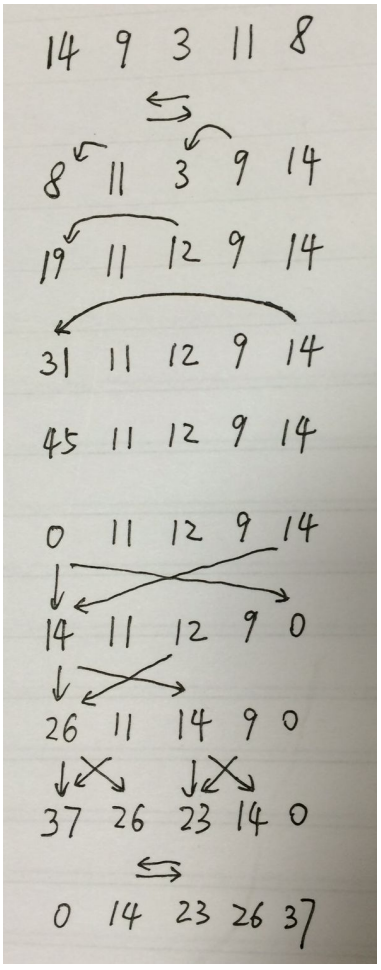

Figure 1

Data and Experiment Result

Following sections gives results for both the implementations. For reversed Blelloch scan, we have not included the timing of reverse in the result. The last reverse operation can avoided by manipulating index. E.g. index = (index + size -1) % size

Results :-
The code is tested on compute node 0, which 24 core 3.5Hz Intel Xeon machine.
$ uname -a
Linux compute-0-0.local 2.6.32-431.11.2.el6.x86_64 #1 SMP Tue Mar 25 19:59:55 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux

$ cat /proc/cpuinfo | grep processor | wc -l
24

Code can be compiled using standard make command on compute-0-0 node in lab.
Steps to compile and run.

a. make ← compiles prefix and r_prefix
b. make test ← runs test for different array values for both implementations.

prefix ⇐ zero filled array implementation.
r_prefix ⇐ reversed blelloch scan implementation.

-bash-4.1$ make
gcc -fcilkplus -O2 -ftree-vectorize -fopt-info-vec-optimized -lrt -o prefix prefix.c
gcc -fcilkplus -O2 -ftree-vectorize -fopt-info-vec-optimized -lrt -o r_prefix r_prefix.c
r_prefix.c:20:13: note: loop vectorized

-bash-4.1$ make test
bash test.sh

Testing standard Blelloch prefix sum implementation.

Array size : 32
sequential prefix sum time in seconds: 0.000008
parallel prefix sum time in seconds : 0.000004
correct result.

Array size : 1023
sequential prefix sum time in seconds: 0.000015
parallel prefix sum time in seconds : 0.000013
correct result.

Array size : 32767
sequential prefix sum time in seconds: 0.000248
parallel prefix sum time in seconds : 0.000444
correct result.

Array size : 33554431
sequential prefix sum time in seconds: 0.081702
parallel prefix sum time in seconds : 0.429251
correct result.

```
--------------------------------------------------
Testing reversed Blelloch prefix sum implementation.
--------------------------------------------------
Array size : 32
sequential prefix sum time in seconds: 0.000009
parallel  prefix sum  time in seconds   : 0.000003
correct result.

Array size : 1023
sequential prefix sum time in seconds: 0.000014
parallel  prefix sum  time in seconds   : 0.000013
correct result.

Array size : 32767
sequential prefix sum time in seconds: 0.000231
parallel  prefix sum  time in seconds   : 0.000555
correct result.

Array size : 33554431
sequential prefix sum time in seconds: 0.082191
parallel  prefix sum  time in seconds   : 0.436850
correct result.
```

Conclusion

We have been able to get the algorithm working correctly. For smaller values of n (< 5000), the parallel algorithm beats the sequential counterpart. However, when n is big enough, the sequential prescan has better performance. We are planning to profile the code to check which operations are time consuming and optimize accordingly.
In order to deal with the situation when n is not power of 2, we used zero filling at first, to make the array work with blelloch algorithm. But we thought it is not efficient, at least it is a waste of memory. So we figured out a new way to workaround this issue. Interestingly, this new way didn't performed better than zero filling. This is an open issue that we will investigate further.

References

Book : Elements of Parallel Computing by Prof. Eric Aubanel
Publisher : Chapman and Hall/CRC ISBN 9781498727891

Paper : Prefix Sums and Their Applications
Guy E. Blelloch Carnegie Mellon University

Cilk Plus Tutorial
https://www.cilkplus.org/cilk-plus-tutorial