Homework 2 Report

Yige Hu and Zhiting Zhu

1 P1

1: **function** REC_PSUM(a, x_0, b, n)

2:   **if** (n == 1) **then**
3:     $s(0) = x_0; return; end;$
4:   **end if**

5:   $x = zeros(n/2, 1);$
6:   $a\_new = zeros(n/2 - 1, 1);$
7:   $x(0) = x_0;$
8:   **parfor** $i = 1 : n$ **do**
9:     $x(i) = b(i);$
10: **end parfor**

11: **parfor** $i = 0 : n/2 - 1$ **do**
12:    $y(i) = x(2 * i) * a(2 * i + 1) + x(2 * i + 1);$
13:    **if** (i != 0) **then**
14:      $a\_new(i) = a(2 * i) * a(2 * i + 1);$
15:    **end if**
16: **end parfor**

17: $c = $ REC_PSUM$(a\_new, y(0), y[1 : n/2 - 1], n/2)$;

18: $s(0) = x_0;$
19: **parfor** $i = 1 : n - 1$ **do**
20:    **if** isOdd(i) **then**
21:      $s(i) = c(i/2);$
22:    **else**
23:      $s(i) = c((i - 1)/2) * a(i) + x(i);$
24:    **end if**
25: **end parfor**

26: **return** s;
27: **end function**

2 P2

2.1 Algorithm

1: **function** SCAN(x, n, l)

2:   $step = ceil(log_2(n))$
3:   $temp = n >> 1$

4:      $offset = 1$

5:      **parfor** $i = 0 : n/2 - 1$ **do**
6:          **for** $j = i; j < temp; j+ = nthreads$ **do**
7:              $indx2 = offset * (2 * i + 2) - 1$
8:              $indx1 = offset * (2 * i + 1) - 1$
9:              $x(indx2) = x(indx1) + x(indx2)$
10:         **end for**
11:         $offset* = 2$
12:         $temp = temp >> 1$
13:     **end parfor**

14:     $temp = 2$
15:     $offset >>= 1$

16:     **parfor** $i = 1 : n/2 - 1$ **do**
17:         $offset >>= 1$
18:         **for** $j = i; j < temp; j+ = nthreads$ **do**
19:             $indx2 = offset * (2 * i + 1) - 1$
20:             $indx1 = offset * 2 * i - 1$
21:             $x(indx2) = x(indx1) + x(indx2)$
22:         **end for**
23:         $temp* = 2$
24:     **end parfor**
25: **end function**

2.2   Result

Wall Clock Time(us)	Number of threads		
Length of Arrary	sequential	6 threads	12 threads
1M	15679	15500	38192
10M	156797.9	212012.5	160871
100M	730794.8	1513714	1262623.5
1B	7305516.5	14843186	12431315.5

Table 1: Wall clock execution time for different array size with different number of threads for 1D vectors

Wall Clock Time(us)	Number of threads		
Length of Arrary	sequential	6 threads	12 threads
1M	20525.5	79923.5	146187
10M	247284.5	539063	375131.5
100M	2046770	4615023.5	3381959

Table 2: Wall clock execution time for different array size with different number of threads for 4D vectors
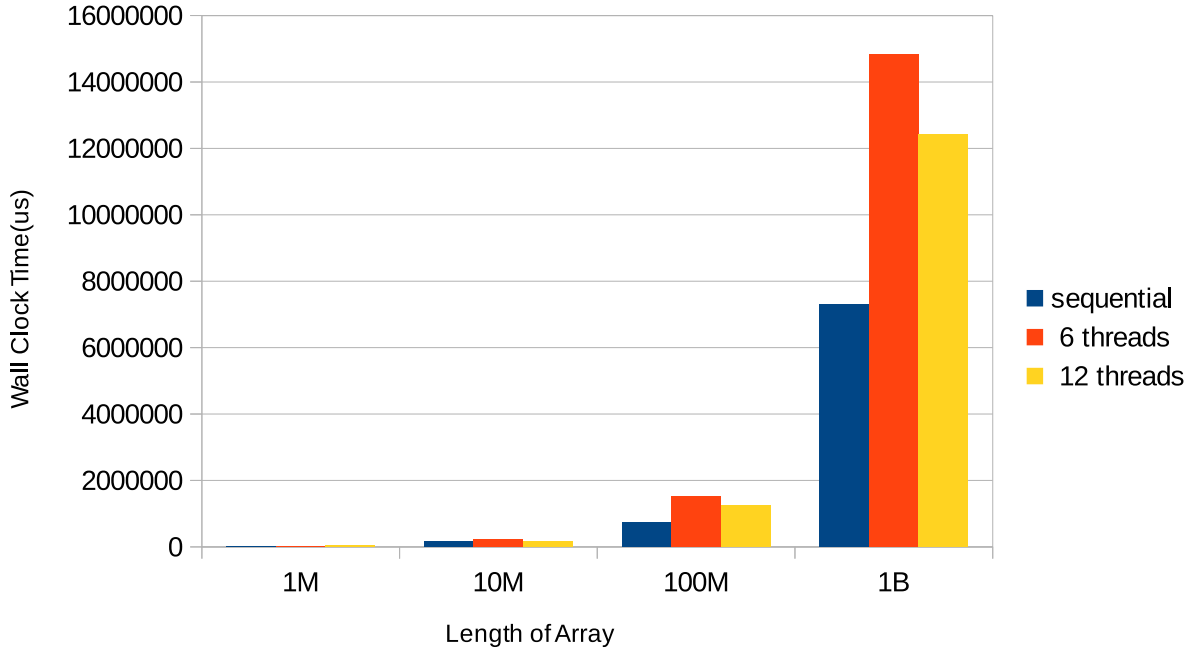
Figure 1: Wall clock execution time for different array size with different number of threads for 1D vectors
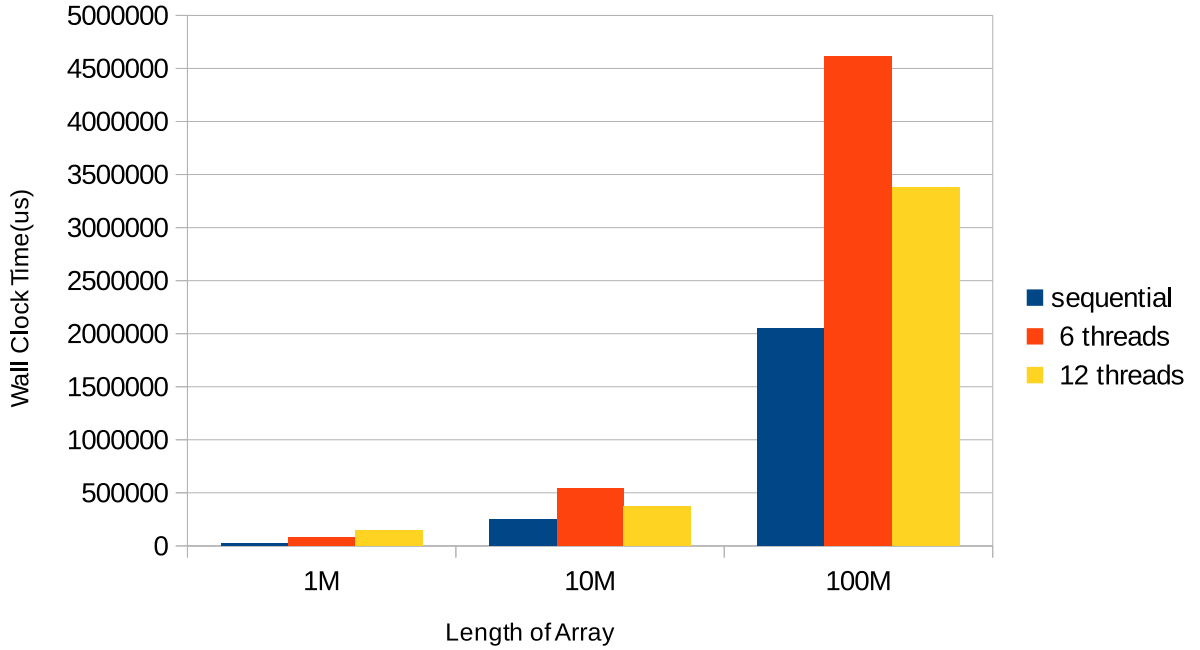

Figure 2: Timing measurements for different array size with different number of threads for 4D vectors

From Figure 1 and 2 and table 1 and 2, the parallel execution runs slower than the sequential execution. The speed up of doubling the number of threads is less than 2. The results are out of our expectation and we have consider two reasons that may be related to this result. First, the cost of administering threads

by OpenMP is high. Second, the input array is shared between multiple threads. During execution, several threads may try to write to this array at different locations which will incur frequent cache coherence for different processors as each thread is pinned to a different processor.

3 P3

3.1 Algorithm

1: **function** SEARCH($comm, keys, num\_keys, sub\_arr, arr\_size, num\_threads, rank$)
2:     $low\_pos = arr\_size * rank$
3:     $high\_pos = low\_pos + arr\_size - 1$
4:     $low = 0$
5:     $high = arr\_size - 1$
6:     **for** $i = 0; i < num\_keys; i + +$ **do**
7:         $low = 0$
8:         $high = arr\_size - 1$
9:         $position = low$
10:         $k = keys[i]$
11:         **if** $k < sub\_arr[high] \wedge k > sub\_arr[low]$ **then**
12:             **if** $arr\_size - 1 <= num\_threads$ **then**
13:                 **parfor** $i = 1 : num\_threads$ **do**
14:                     **if** $sub\_arr[low + i] <= k$ **then**
15:                         $position = low$
16:                     **end if**
17:                 **end parfor**
18:             **else**
19:                 $len = (arr\_size - 1)/(num\_threads + 1)$
20:                 $position = low$
21:                 **while** $len! = 1$ **do**
22:                     **parfor** $i = i : num\_threads$ **do**
23:                       $left = low + i * len$
24:                       $right = (left + len - 1) > high?high : (left + len - 1)$
25:                       **if** $sub\_arr[left] == k$ **then**
26:                         $position = left$
27:                         $len = 1$
28:                       **else if** $sub\_arr[right] == k$ **then**
29:                         $position = right$
30:                         $len = 1$
31:                       **else**
32:                         **if** $sub\_arr[left] < k \wedge sub\_arr[right] > k$ **then**
33:                           $len = (len - 1)/(num\_threads + 1)$
34:                           $low = left$
35:                           $high = right$
36:                           $position = low$
37:                       **end if**
38:                     **end if**
39:                 **end parfor**
40:                 **end while**
41:             **end if**
42:         $position = position + low\_pos$
43:         Output position
44:         **else if** $k == sub\_arr[high]$ **then**
45:             $position = high + low\_pos$
46:             Output position

```
47:        else
48:            if k == sub_arr[low] then
49:                position = low + low_pos
50:                Output position
51:            end if
52:        end if
53:    end for
54: end function
```

3.2   Result

Length	1 core	1 socket	1 node	2 node

Table 3: Wall clock execution time for different configurations to search array with different length

We have sussessfully run the program with 16 keys but for large keyspace(2^{20} keys) our program hangs. We could not figure out the reason for hanging but we suspect that we do not have enough memory to support the full execution.