Prefix Scan and Barriers

Fukun Lai

EID: fl6944

1 Background

Prefix scan is trivial if performing it sequentially. Conceptually, prefix scan of a list of data $[x]$ it could be done by:

$$y_0 = x_0$$
$$y_i = x_i \oplus y_0$$

Sequential scan takes time complexity of $O(n)$, where n is the number of element of in the data set.

For large data set, sequential scan might not be efficient and practical. Parallel scan using multiple processors could improve efficiency.

2 Parallel Algorithm

2.1 Hillis and Steele algorithm

There are a few parallel algorithms that could perform prefix scan. A relative simple approach is to add adjacent element of different strides parallelly at each step. The pseudocode for each processor/thread is:

```
int stride = 1;
// each processor handles a block of elements by n_threads
while (stride <= n) {
        for (int j = thread_id; j < n; j += n_threads) {
                if ( j >= stride ) {
                        out[j] = in[ j – stride ] + in[ j ];
                } else {
                        out[j] = in[j];
                }
        }
        stride *= 2;
}
```

If there are total n_p processors, and n data elements, the time complexity of this algorithm is $O(\frac{n}{n_p}\log(n))$. However, the total work complexity is $O(nlog(n))$ which is the sum of work that

all processors need to perform. This means the above algorithm is not work efficiency, since a sequential scan only need $O(n)$.

2.2 Blelloch algorithm

There are a few alternatives of the above algorithm to make it work efficient, one of them is Blelloch algorithm which is invented by Blelloch in 1990. This algorithm consists of two phases, called up sweep (or reduce) phase and down sweep phase. The up sweep is to compute sum of two adjacent elements (difference by stride) in-place parallelly, and the down sweep is to build the remaining sum by reset the root element to zero. An C implementation of Blelloch algorithm is illustrated below:

Up-sweep algorithm:

```
int stride = 1;
while (stride <= n) {
    if (thread_id < n/2/stride) {
        //data that each processor needs to handle
        for (int j = (thread_id+1)*2*stride-1; j < n; j += 2*stride*n_threads) {
            output_vals[j] += output_vals[j-stride];
        }
    }
    stride *= 2;
}
```

Down-sweep algorithm:

```
int stride = n/2;
if (thread_id == 0)
    output_vals[n-1] = 0;
while (stride > 0) {
    if (thread_id < n/2/stride) {
        //data that each processor needs to handle
        for (int j = (thread_id+1)*2*stride-1; j < n; j += 2*stride*p) {
            int tmp = output_vals[j];
            output_vals[j] += output_vals[j-stride];
            output_vals[j-stride] = tmp;
        }
    }
    stride /= 2;
}
```

Blelloch algorithm is work efficient, because for every step, it does $n/\log(n)$ operations, and it has totally $O(\log(n))$ steps. However, the disadvantage of Blelloch algorithm is the length of input array has to be power of 2, to make it a balanced tree. The input array could be extended to power of 2 and padding with 0 if it is not already so. Another disadvantage is that it need to use barrier for every loop, and this could be used $2\log(n)$ times.

2.3 Two Level algorithm

Another work-efficient algorithm is to divide the data into a few blocks and use two levels to calculate the prefix sum as following steps:

1. Divide the data into $n_{threads}+1$ blocks;
2. Compute prefix sum parallelly for the first $n_{threads}$ (i.e., from 1 to $n_{threads}$) blocks;
3. Compute the prefix sum of the last element of the first $n_{threads}$ blocks, this could be done parallelly using the sequential algorithm or work inefficient Hillis and Steele algorithm;
4. Compute the prefix sum for the last $n_{threads}$ (i.e., from 2 to $n_{threads}+1$) blocks.

Step 2 and 3 is to calculate the start value of each of the blocks, and then the actual prefix sum of each element could be calculated. Step 3 needs $n_{threads}$ operations, if the number of processes is not big, a sequential computing is a good option.

Step 2 pseudocode:

```
int dn = n/(p+1); // block size
int last = (tid+1)*dn – 1; // last id of current block
output_vals[tid * dn] = input_vals[tid * dn];
//computer prefix sum at tid blocks
for (int j = tid * dn + 1; j < (tid + 1)*dn; j++) {
      output_vals[j] = output_vals[j-1]+ input_vals[j];
}
input_vals[last] = output_vals[last];
```

Step 3 pseudocode:

```
// if total number of process is large, compute prefix sum parallelly
if (n_threads > 100) {
      int stride = dn;
      while (stride <= n) {
         if (last >= stride) {
            output_vals[last] = input_vals[last] + output_vals[last - stride];
         } else {
            output_vals[last] = input_vals[last];
         }
      }
   } else if (tid == 0) {    // computer prefix sum sequencially
      int x = 0;
      for (int i = 1; i <= p; i++) {
         x += output_vals[i*dn- 1];
         output_vals[i*dn - 1] = x;
      }
   }
```

Step 4 pseudocode

```
int i = tid+1;
for (int j = i * dn; j < (i == p ? n : (i + 1)*dn - 1); j++) {
      output_vals[j] = output_vals[j-1] + input_vals[j];
   }
```

The advantage of this algorithm is that it could be used for any size of input array without padding, and it only need to use two times of barrier if sequential prefix sum is calculated in step 3.

3 Prefix scan results

The above three algorithms have been implemented and the results are plotted below for different data sets (1k, 8k, and 16k data). The scan operation has 100k loops, and the number of threads is from 0 to 32 with 2 increment. The running time of all curves are in log scale.

It is obvious that HS algorithm is not work efficient, as multi processors result in much higher time consuming than signal processor as shown in Figure 1.

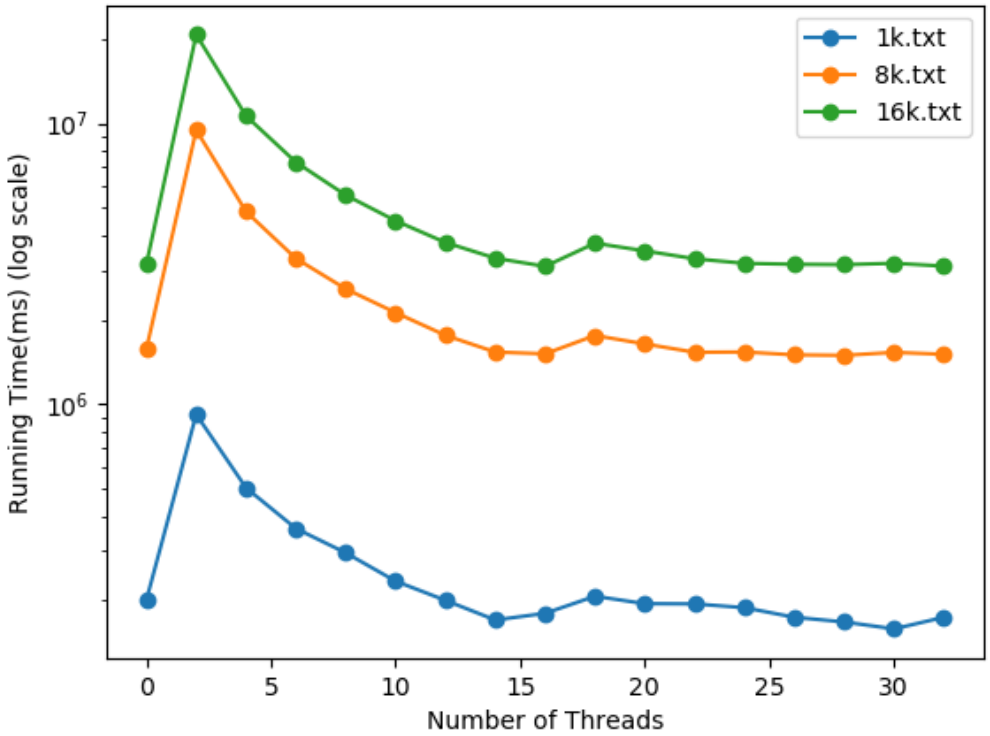

Figure 1: Performance of HS algorithm

The Blelloch algorithm is work efficient as shown in Figure 2, though time of 2 processor is similar to single process, but it scales down when more processes are used.

As expected, the two level algorithm has the best performance, it is scaled as more processes is added as shown in Figure 3.

Figure 4 shows the comparison of all three algorithms with 16k data. It is seen that two level algorithm has the best performance, and as expected, HS algorithm performs the worst. One reason might be two level algorithm uses less barriers which saved a little time comparing to two level algorithm.

One other observation for these algorithm is that, the running time decreasing as the threads increase until the thread number reaches 16. These code is running in a 8 core CPU with hyper threading, which means each CPU core could handle two virtual processes efficiently. Therefore, it could scale upto the maximum CPU virtual cores that the current computer has. Further increase of threads results in more overhead and actually interleaved between thread, and thus suppress the benefit with more threads.

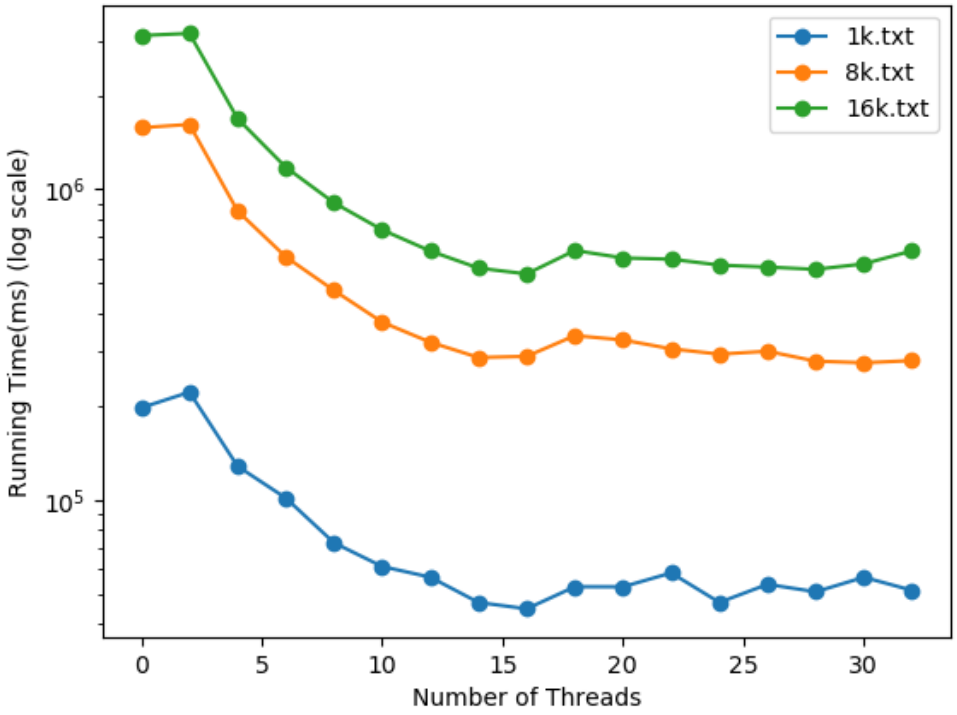

Figure 2: Performance of Blelloch algorithm

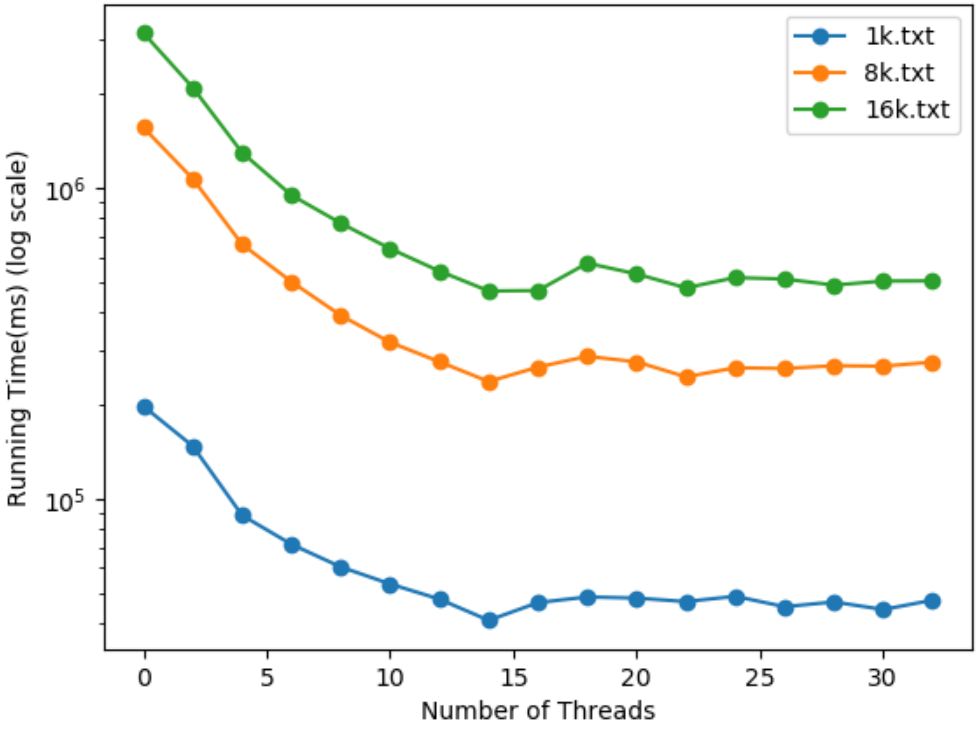

Figure 3: Performance of two level algorithm

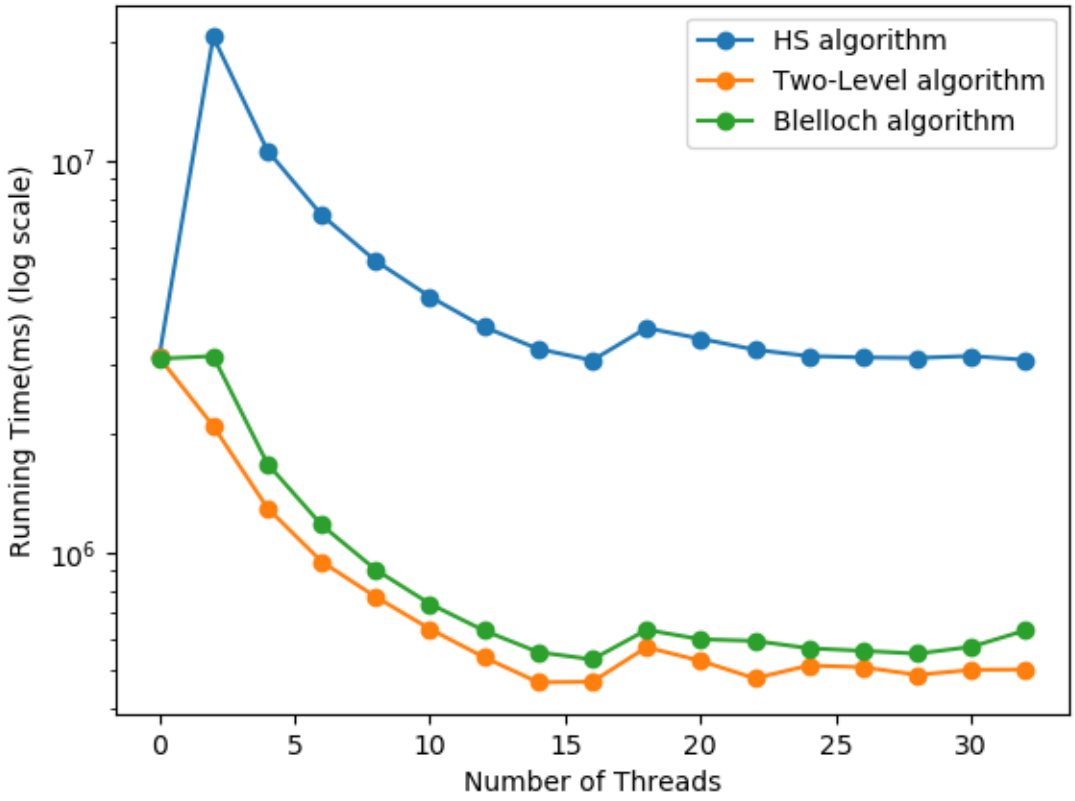

Figure 4: Comparison of HS algorithm, Blelloch algorithm and two-level algorithm

4 Barrier Implementation

There are a number of different barrier implementation, including linear counter based, tree based and butterfly barrier. Linear counter has a centralized counter that counts the number of threads that has reached to the barrier. Tree based barrier is a more distributed barrier, and it has a number of counters on the tree node. Once all of the children of a node arrived, it notifies its parent. Once the root node has all of its children arrived, the tree recursively notifies the children of a node to release. Tree based barrier doesn't have a centralized counter, and it is faster if a large number of processors are visiting the barrier.

In this lab, a spin lock based linear counter barrier is implemented. This barrier maintains a global flag, a global counter, and a local flag. When a process get into the barrier, reverse the local flag. If the counter reaches the total number of processes, reset the global counter, and assign the local flag to global flag. Otherwise, wait until the global flag is the same as the local flag. Note that, all process should have the same initial local flag. This implementation is straightforward and has better performance for small number of processes. Pseudo code as follows:

```
*local = 1 - *local;

    pthread_spin_lock(&barrier->lock);
    barrier->counter++;
    int arrived = barrier->counter;
    if (arrived == barrier->total) {
        pthread_spin_unlock(&barrier->lock);
        barrier->counter = 0;
        barrier->flag = *local;
    } else {
        pthread_spin_unlock(&barrier->lock);
        while(barrier->flag != *local);
    }
```

5 Results and Discussion

5.1 Parallel Implementation

As shown in Figures 1-4 above, work efficient algorithm such as Blelloch and two level shows a trend of decreasing running time with more threads up to a certain number, which in this case 16. The reason is because this lab is running in a 8 core computer with hyper-threading on. This effectively could has 16 threads running parallelly, and thus could be scaled up to 16 threads. Further increment of threads would force the processor to run concurrently, which does not benefit in the current test.

5.2 Playing with Operators

With 10 operation loops only, more threads increase the running time as shown in Figure 5. This is because of the overhead on thread creation and running synchronization. The benefit of multi-process/threads is to be able to perform the operation parallely, (i.e., at the same time). But there are some additional cost including overhead of threads and synchronization. The time saving on the parallel operation must be over the additional cost to be effectively. With simple operation (e.g., 10 loop of addition), each scan operation is so trivial that the time saving on multi-process is not able to overcome the additional cost of thread overhead and synchronization. That is why the increasing trend appears with 10 loop.

The inflexion point is dependent on the data set, for 8k and 16k data set, it is at about 55 loop as shown in Figure 6.

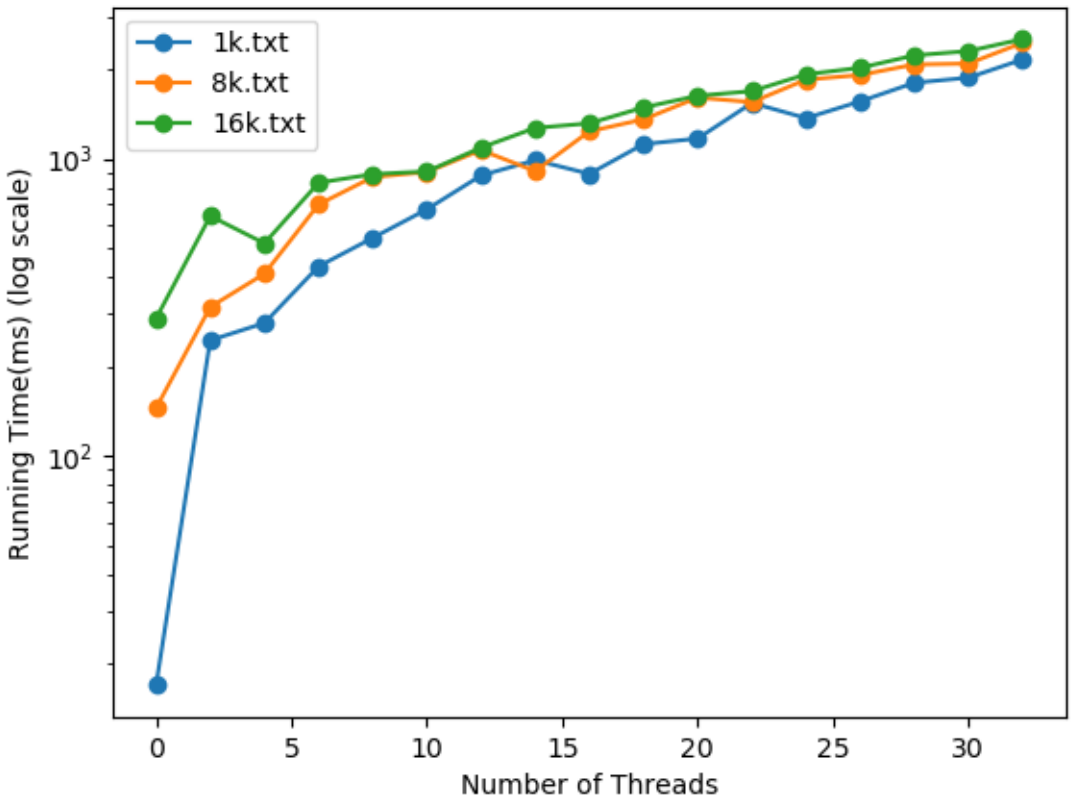

Figure 5: Performance trend with 10 operation loops

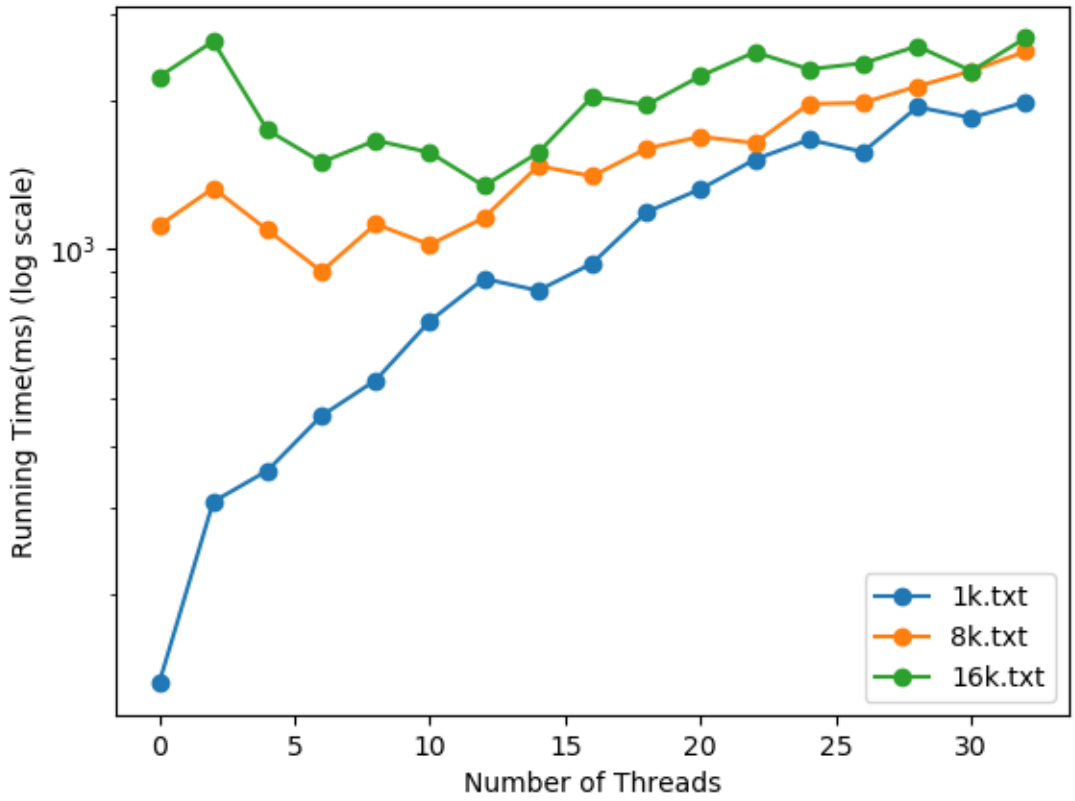

Figure 6: Performance trend of 55 operation loops

5.3 Barrier Implementation and Results

Barrier implementation is presented in Section 4. The current implementation uses a linear counter with spin lock, where pthread barrier use similar idea.

Figure 7 shows the performance trend of current implementation of spin barrier for different data set. It has a big performance down when the threads increase to 16. This is again related to the computer that the code is running. With 16 virtual core, up to 16 threads could access the spin barrier parallelly, and this will not cause big synchronize time delay. If more than 16 threads are created, at least one core is running more than two threads concurrently, which requires to access the barrier. This causes synchronization delay.

Figures 8 and 9 shows the performance trend for the current spin barrier, the inflexion point is dependent more on the number of threads. Figure 10 shows the comparison of the pthread barrier and spin barrier, it is observed that pthread barrier is better at large number of threads, and spin barrier perform a little better at small number of threads, the overturn point is the number of processor of the computers.

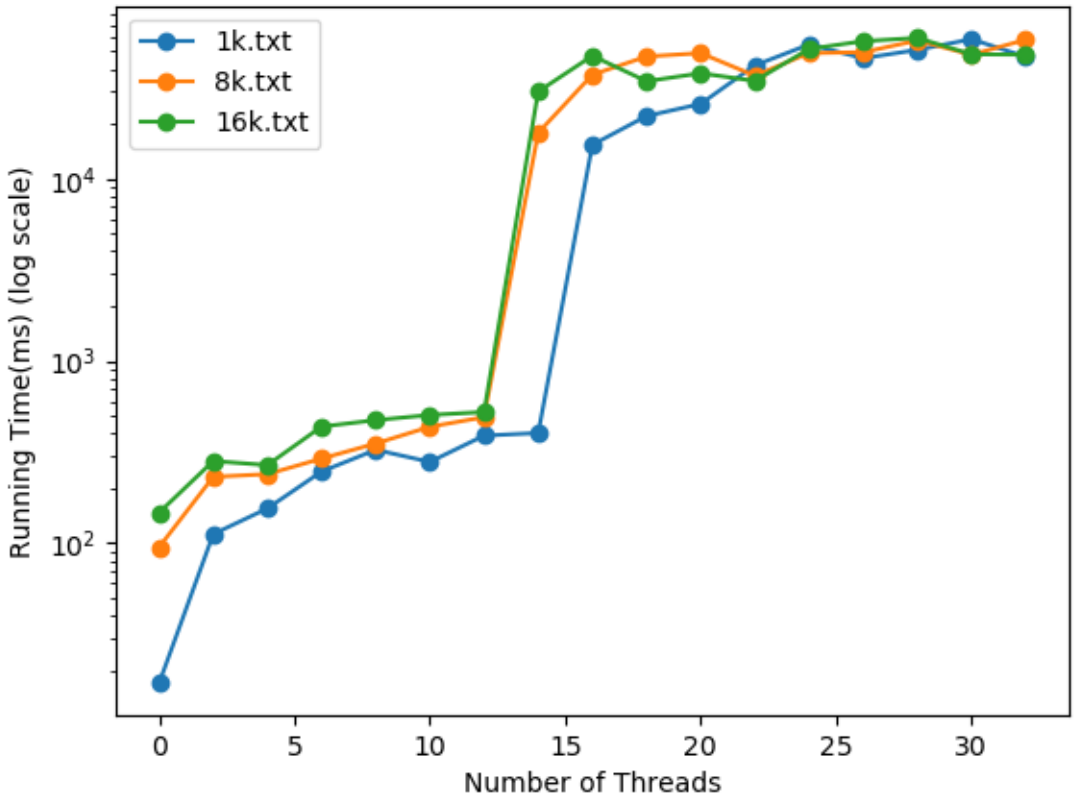

Figure 7: Performance trend with spin barrier and 10 loops

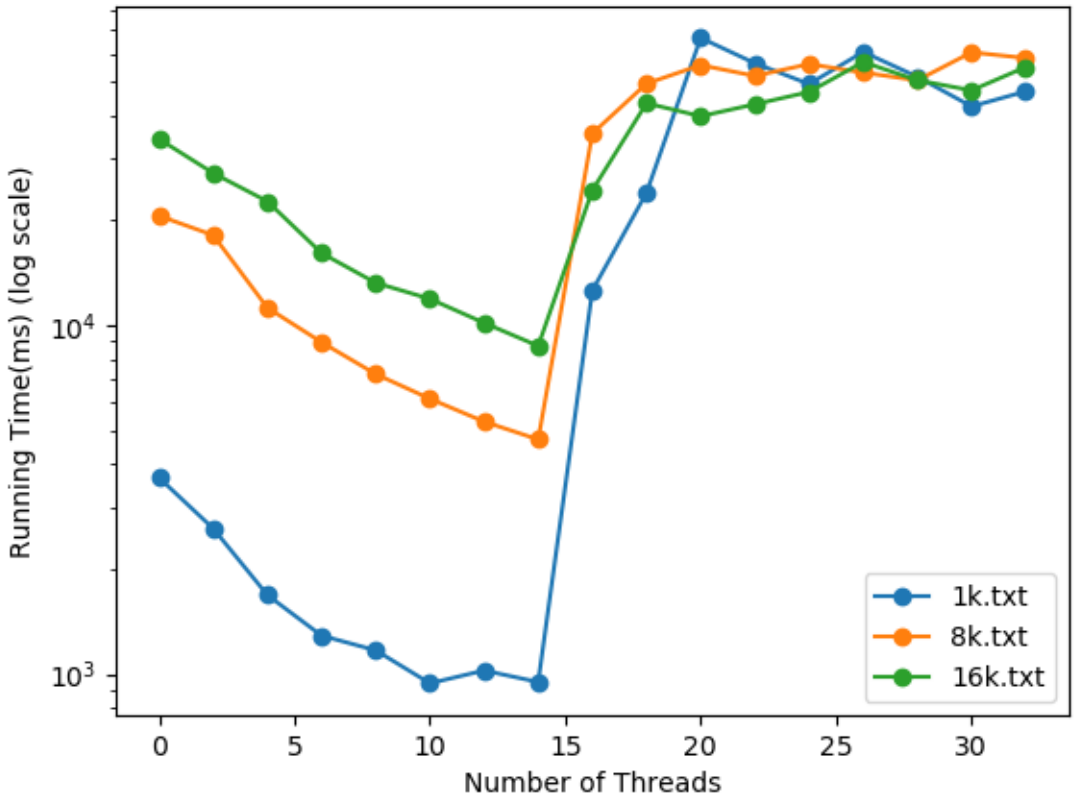

Figure 8: Performance trend for spin barrier with 1000 loop

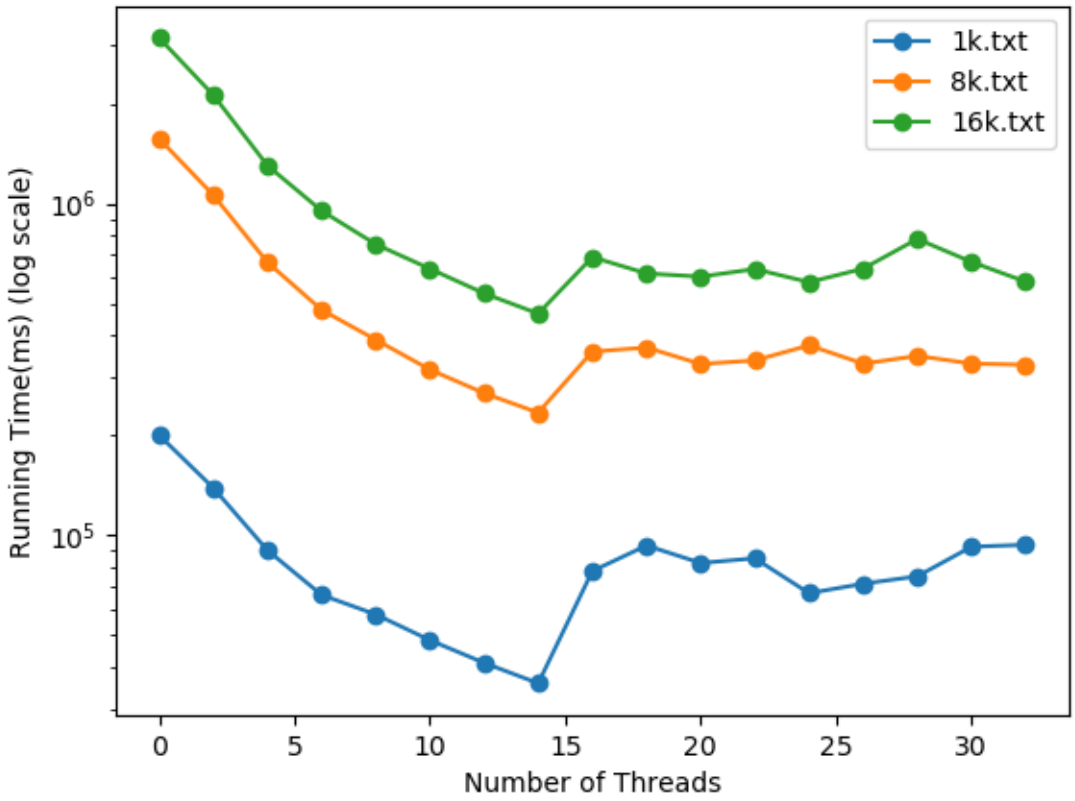

Figure 9: Performance trend for spin barrier with 100k loop

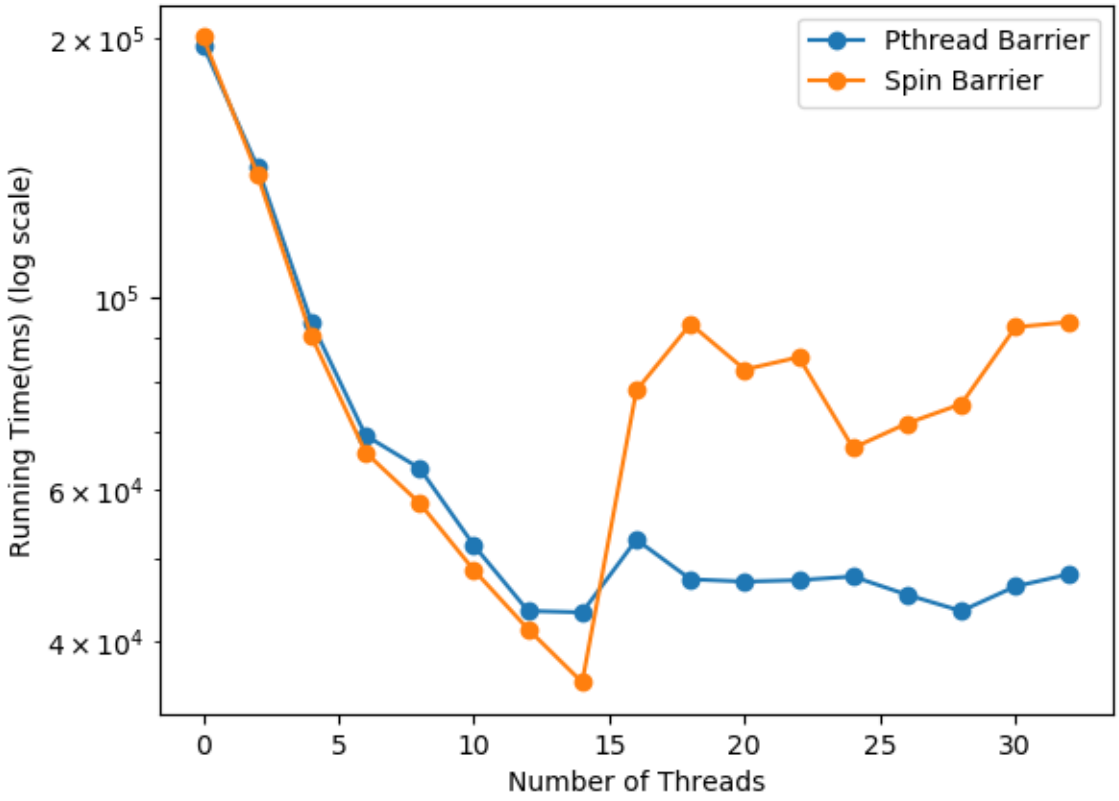

Figure 10: Comparison of Pthread Barrier and Spin Barrier

6 Conclusion

A few work-inefficient and work-efficient prefix scan algorithms have been implemented and compared in this report. It is found the two-level algorithm gives the best performance yet not difficult to implement. A spin barrier based on spin lock and linear counter has been implemented to compare with pthread barrier. It is observed the current implementation performs better if the number of threads is less than the number of processors, where pthread barrier performs better if the number of threads is more than the number of processors.