Lab 1 – Prefix Scan and Barriers

Jerry Reinoehl

CS380P Fall 2023

1 Prefix Scan Implementation

To implement parallel prefix scan I used the Blelloch algorithm. First we divide the elements among the threads according to

$$B = \left\lceil \frac{N}{T} \right\rceil = \left\lfloor \frac{N+T-1}{T} \right\rfloor$$

where B is the block size, N the number of elements, and T the number of threads. To make the implementation a bit simpler and to avoid index-out-of-range errors, we allocate $B*T$ elements for both `input_vals` and `output_vals`. This allows all threads to operate on the same number of elements, and if the number of elements doesn't evenly divide the number of threads, extra zeroes are added only to the final thread without altering the final result.

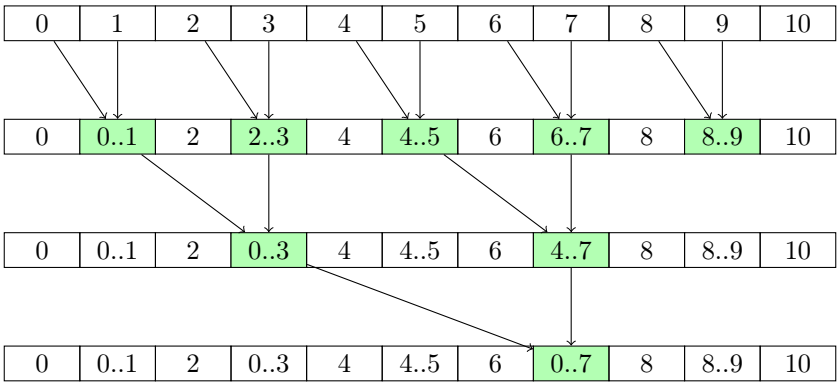
```
int block_size = (*n_vals + args->n_threads - 1) / args->n_threads; // ceil(N/T)
int output_size = block_size * args->n_threads;

// Allocate input and output arrays
*input_vals = (int*) calloc(output_size, sizeof(int));
*output_vals = (int*) calloc(output_size, sizeof(int));
```

Here, we initialized the extra elements with zeroes for the prefix sum, however, if we were to perform a different operation, such as multiplication, it is important to note that we must initialize with the identity value for our operation (e.g. 1's for multiplication).

The first step of the prefix scan is for each thread to perform a sequential scan on its own block of elements. This step also copies the input values to the output array so we may continue the scan in place. We allow all threads to synchronize as later steps are dependent on the values that are calculated here. This results in $O(N)$ operations in $O(N/T)$ time.

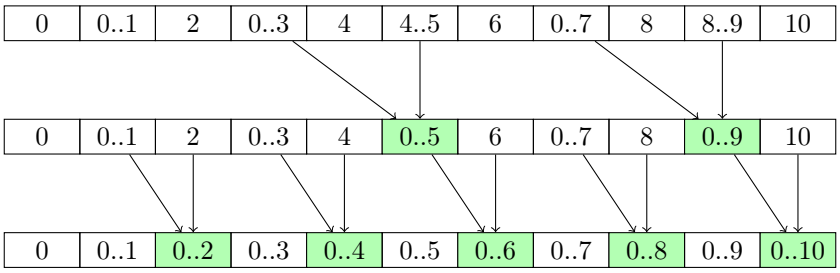
```
// Perform prefix scan on our block of data.
int x, y;
x = input[block_start];
output[block_start] = x;
for (int i = 1; i < block_size; i++) {
  y = input[block_start + i];
  x = op(x, y, n_loops);
  output[block_start + i] = x;
}

barrier->wait();
```

Next, we perform the up-sweep. We select every second thread and have it combine its sum with the previous thread's sum, and synchronize. We then select every fourth thread and combine it with the thread with an ID of two less than the current thread. We continue this operation, doubling the stride until the stride is greater than the number of threads. The up-sweep performs $O(T)$ operations in $O(\log T)$ time.




```
// Compute up-sweep.
for (stride = 1; stride < n_threads; stride <<= 1) {
  if ((t_id + 1) % (stride << 1) == 0) {
    x = output[block_end - stride * block_size];
    y = output[block_end];
    output[block_end] = op(x, y, n_loops);
  }
  barrier->wait();
}
```

We then compute the down-sweep. We select every `stride * 2` threads and have it combine its sum with the thread `stride` units greater, and divide the stride by 2. We continue while the stride is greater than 0. Similar to the up-sweep, the down-sweep does $O(T)$ operations in $O(\log T)$ time.




```
// Compute down-sweep.
for (stride = stride >> 1; stride > 0; stride >>= 1) {
  if ((t_id + 1) % (stride << 1) == 0 && (t_id + stride) < n_threads) {
    x = output[block_end + stride * block_size];
    y = output[block_end];
    output[block_end + stride * block_size] = op(x, y, n_loops);
  }
  barrier->wait();
}
```

Finally, each thread computes the final results for its block by adding the computed sum of the previous thread to each of its block's elements except for the last, which already contains the correct sum. This final

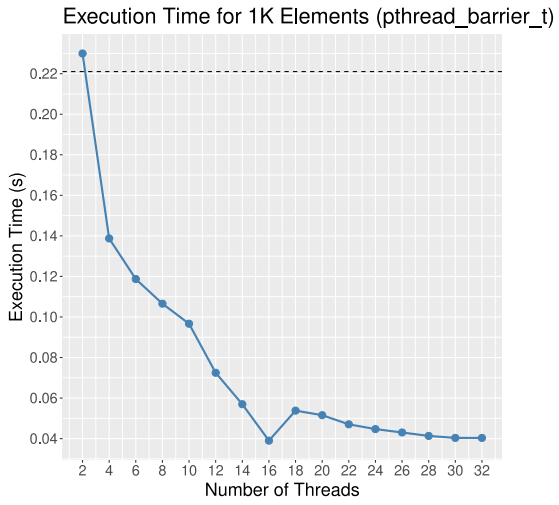
step performs $O(N)$ operations in $O(N/T)$ time.

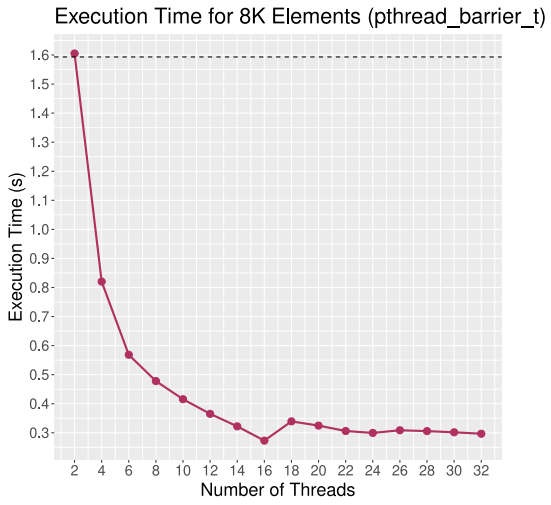
```
// Finalize prefix sum for our block. Add the sum from the previous block
// to all of our elements except for the last.
if (t_id > 0) {
  for (int i = 1; i < block_size; i++) {
    x = output[block_end - i];
    y = output[block_end - block_size];
    output[block_end - i] = op(x, y, n_loops);
  }
}
```

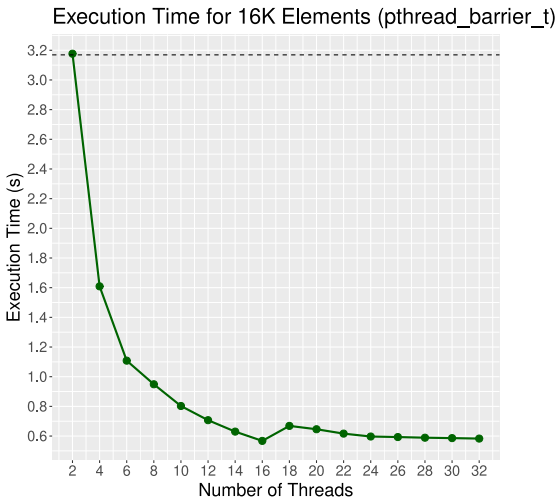
The total number of operations comes to $O(2N + 2T)$ in time $O(\frac{2N}{T} + 2\log T)$, or $O(N)$ and $O(N/T)$ for large N. The algorithm is work efficient because it does not do more operations asymptotically than the sequential implementation.

2 Parallel Prefix Scan Results

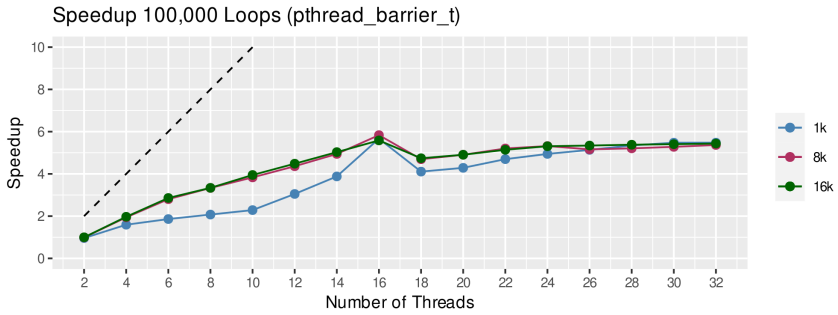
The following graphs depict the execution time of the parallel prefix scan, with 100,000 loops, for 1k, 8k, and 16k elements. The dashed line indicates the running time of the sequential prefix scan. These tests were run on a 16 core CPU, and you can see we have a steady improvement in performance until we run with more threads than the CPU actually has. At this point, running with more threads only adds more overhead as we can still only run 16 threads simultaneously, but must now incur the cost of creating and destroying more threads as well as the cost of contention at the barrier. This indicates that for best performance we should set the number of threads to the number of CPU core we actually have.



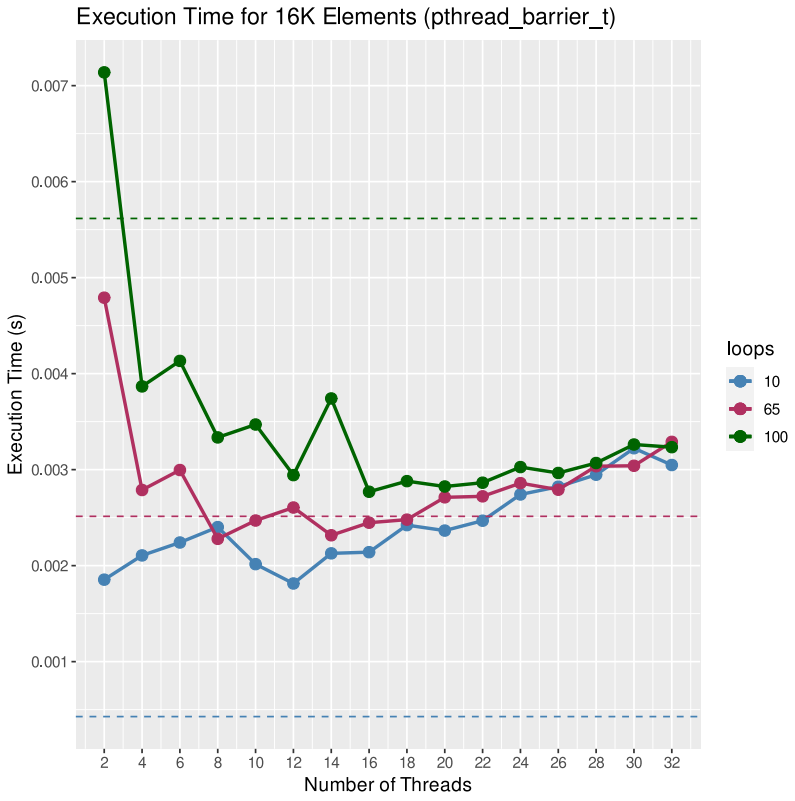




The graph below shows the speedup results, again for 100,000 loops. The dashed line indicates the ideal speedup. Again, we notice a peak at 16 threads, and also while we can attain a good speedup, we are not achieving the ideal speedup due to overhead and contention. We also note that we can get better speedup by increasing the number of elements.



By varying the number of loops we can find where the parallel implementation is no longer faster than the sequential scan. We see that at roughly 65 loops the parallel and sequential implementations run in about the same time. This suggests that the amount of work performed by the operator affects the speedup attained by incorporating parallelism. Parallelism is more effective when we want to parallelize longer running, more compute intensive operations, rather than simple ones, such as scalar addition. We want to amortize the overhead costs with a longer running computation to achieve a performance gain.



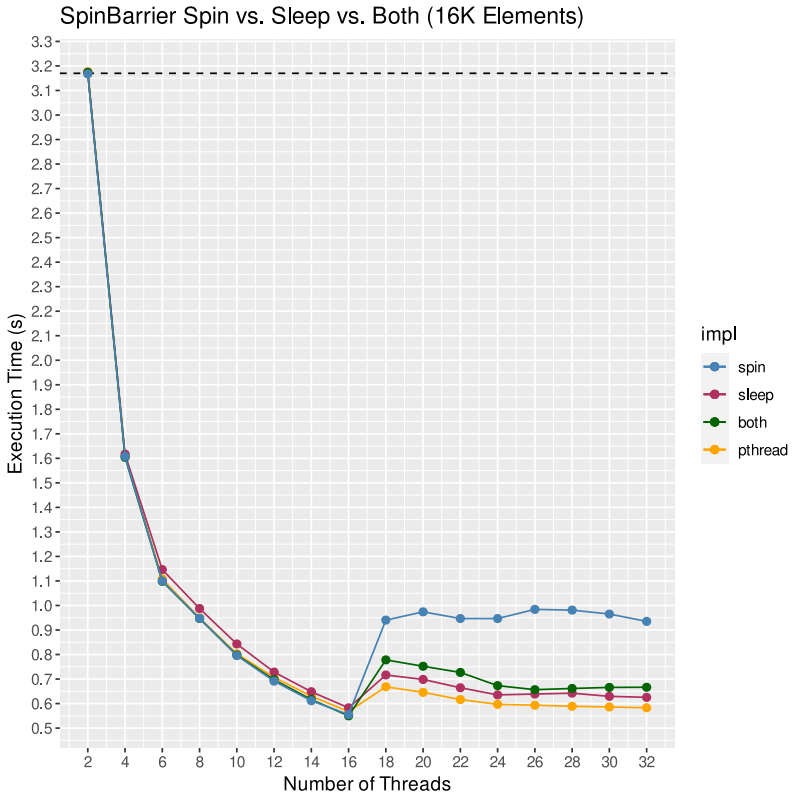
3 Barrier Implementation

My barrier implementation is based on a mutex and an array of go integers. Upon entering the barrier, a thread would grab the lock, decrement the count, grab its go bit, and check if the count became zero. If the count is now zero, it will reset the count, and flip all go bits, thus signaling to all awaiting threads that they may proceed. If the count is not yet zero, the thread would spin waiting for its go bit to flip.

I did find that simply spinning was good for performance up to the number of threads of the machine, however, when setting the number of threads greater, spinning often resulted in execution time much slower than the equivalent `pthread_barrier` scan. This was due to threads hogging the core and not yielding the CPU to the threads it was waiting on.

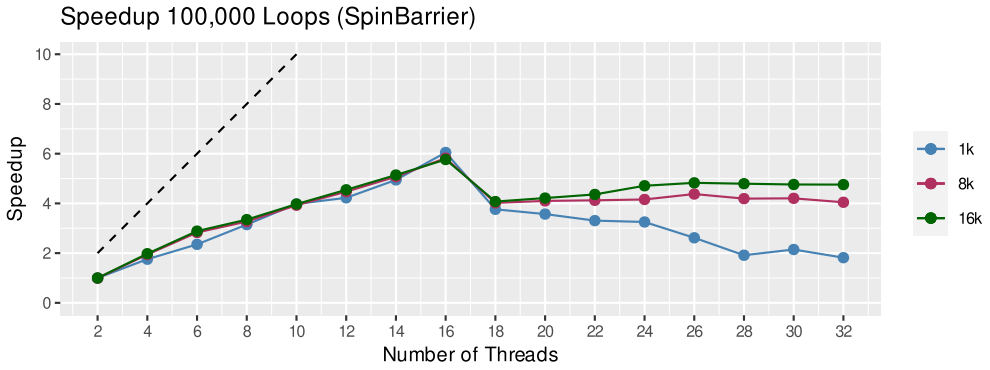
To fix this problem I added a `sleep(0)` to yield the CPU. While this greatly improved performance for threads greater than 16, it did hurt performance for threads 16 or less due to additional context switches. I was able to find a middle ground by spinning only the number of threads of the machine, and yielding the first threads that showed up. For example, if running with 20 threads, the first 4 threads would yield, while the remaining 16 threads would spin. This achieved similar performance for threads less than 16 because no threads would yield, and also achieved much better performance when setting the number of threads to something greater.

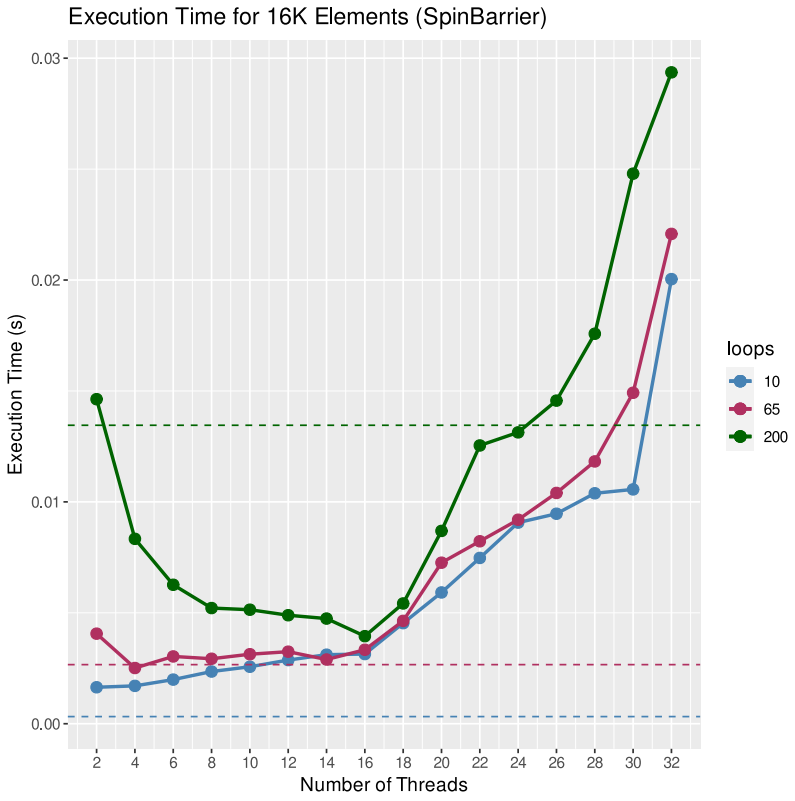
```
static const int HARDWARE_CONCURRENCY = std::thread::hardware_concurrency();

inline void SpinBarrier::wait() {
  int count;
  int go;

  pthread_mutex_lock(&count_lock_);
  count = --count_;
  go = !go_[count];

  if (count == 0) {
    pthread_mutex_unlock(&count_lock_);
    count_ = go_len_;
    for (int i = go_len_ - 1; i >= 0; i--)
      go_[i] = go; // flip go ints
  } else {
    pthread_mutex_unlock(&count_lock_);
    while(go_[count] != go) {
      if (count >= HARDWARE_CONCURRENCY) {
        sleep(0);  // yield cpu
      }
    }
  }
}
```



The `pthread_barrier_t` implementation is very similar to my own, with the difference being that, instead of an array of go bits to wait upon, it uses a condition variable. When a thread waits for the condition variable, it will be put to sleep until signaled by the corresponding broadcast. Whereas, with my implementation, we may yield the thread only to be rescheduled later and again re-yield. This hurts performance due to additional context switching. This is why we see faster execution times from the pthread barrier implementation above when the number of threads is greater than 16. However, because my implementation does not yield when the number of threads is less than the number of machine threads it has slightly better performance for threads less than or equal to 16.

The takeaway here is that if we are executing many threads, maybe over many programs, it is better to yield to give the CPU to the threads you are waiting on. However, for better performance we can just spin with a number of threads equal to the number of CPU cores.





Time Breakdown	
Prefix Scan Impl	15 hrs
Barrier Impl	8 hrs
Report	10 hrs
Total	**32 hrs**