CS 378H Lab 1 Report

Slava Andrianov

Introduction

In this report, I provide an overview of my findings from implementing and benchmarking two parallel prefix scan algorithms, the Blelloch work efficient algorithm and the two level algorithm against the standard serial prefix scan algorithm. The approach section details the logic behind my implementation, from setting up everything that the algorithms need to run to the logic behind the algorithms themselves. It also details the design of my barrier implementation and compares it against the implementation of the pthreads barrier. The performance section explains my system configuration and the results of my benchmarking. This contains the results of running three sets of benchmark suites against a set up of the Blelloch work efficient algorithm with pthreads barriers, the Blelloch work efficient algorithm with my barrier implementation, and a comparison of the two level algorithm with pthreads barriers and the Blelloch work efficient algorithm with pthreads barriers.

Approach

Before going into the full description of my approach, it is important to note that while the description here often refers to the summing of elements, the code itself is templated and can support any relevant prefix scan operation. Sum is used in the description because the operations performed during this lab were sums of either integers or vectors of floating point numbers.

The initial part of my code parses the command line arguments, parses the input into an array of the relevant data type, and creates a function from the `scan_op` template by providing the relevant data type and a templated functor as template arguments in order to create a function pointer which will perform the operation inside of the functor on the two passed in data values that are passed into the function pointer. In the case of this lab, I have provided three templated functors: one which sums two elements of the given data type using the plus operator, and two which sum vectors of the given data type. Out of the two vector addition functors, the one that I prefer to use is one which makes a copy of one of the first vector, performs the vector addition while saving the results into that copy, and then returns the copy. The other one does almost the same thing, except that it takes in two const references and creates an output vector explicitly which is then used to save the results of the vector addition. Afterwards, my code enters a setup function called `launch_threads` which performs all of the relevant set up for running the prefix scan depending on the passed in flags. In the case of the two level scan flag being present, the input array is resized upwards to be a multiple of the number of threads if it is not already, and then an intermediate array is created the size of which is the number of threads rounded up the the nearest power of two. If the two level scan flag is not present, then the input array is resized up to a power of two. In all of these cases, the additional elements have the value of the default constructor for the input data type, so correctness should not be impacted by this padding. The reason for this padding is because the parallel prefix scan is modeled after a complete binary tree, so the additional padding up to a power of two prevents any issues in the algorithm that may be caused by binary tree properties not being upheld. Afterwards, the presence or absence of the my barrier flag is used to determine the contents of a bar class object, which is essentially a wrapper around a union which allows the prefix scan algorithm to correctly wait on the barrier regardless of if its using a pthreads barrier or my barrier without needing to modify the code of the algorithms themselves. Once these decisions are made, the code then sets up some structs which hold the relevant arguments for each of the threads to use during the prefix scan, and then all of the threads are spawned in and begin execution. After all the threads have joined, all relevant allocated resources are released and the input array is resized back to its initial size. Another important decision to note is that in the specified number of threads is greater than the size of the input array, threads with an id greater than or equal to the size of the data array are not counted towards the capacity passed in to the barriers, and the function pointer that they are given to execute is to an empty function which immediately returns.

Serial Prefix Scan

My implementation of the serial prefix scan can be found in the function `prefixScanSeq` and it follows the generic and intuitive approach for building a inclusive prefix sum array, where the first index in the array is left unmodified and then starting from the second index each value in the array is updated to the its own value plus the value of the updated value of the previous element. The first element does not need to be modified because the sum of all of the values in the array up to the first index is just the value at the first index. For the rest of the indices in the array, since this algorithm starts at the second element and moves forward towards greater indices, it is guaranteed that before the update to index x where x is not the first index in the array, index x-1 will contain the sum of all of the values in the array up to and including the value at index x-1, so when the value for index x is updated to be the current value at index x plus the value at index x-1, index x will now contain the sum of all of the values in the array up to and including index x. As such, this ensures the correctness of the serial prefix scan implementation.

Parallel Prefix Scan Implementation - Work Efficient

My work efficient parallel prefix scan can be found in the function called `prefixScanPar` and it is heavily inspired by the algorithm described by Blelloch (1990). It is composed of an up sweep and a down sweep which together construct the full prefix scan array. The up sweep is implemented in the exact manner that is specified by Blelloch: the algorithm keeps track of an offset value which initially starts at one to indicate that adjacent elements should be summed, and this offset is used in conjunction with the thread index to determine which elements should be summed, with this sum being stored in the greater index. This offset doubles with each iteration until the offset is the size of the array. However, one way that my algorithm differs from the approach described by Blelloch (1990) is its handling of the case where the input size is greater than the number of threads. The algorithm also calculates the span over the array that the current number of threads can reach with the current offset, which is just double the offset. The algorithm uses the offset and its thread id to calculate the two indices to sum the values of. After each thread determines its indices and sums the values at those indices, it then adds this span to both of the indices. If both indices are still within the bounds of the array, then it will sum the values at those indices and keep going. This mechanism ensures at the end of the summing at each level of the tree, the array's values are equivalent to what they would have been if there was a thread to sum each relevant pair of values as described in the Blelloch (1990) algorithm.

On the down sweep, I also follow the algorithm described by Blelloch but with a slight modification to make my prefix sum inclusive instead of exclusive. I make the observation that at any non leaf node in the full binary tree, that node can be thought of as spanning and continuing the sum of some portion of the array. For example, for the root that would be the entire array. The left and right children of this node each contain the sums of the earlier and later half of this span, and this is true for any non leaf node in the tree. As such, for any node, each of its grandchildren contain exclusive sums of half of the span of its parent, or one quarter of the span of its grandparent. As such since each node's child contains half of the span of itself, and its grandchild contains a quarter of the span of itself, this means that for each left child can add its value to the left child of its sibling and not cause any value in the array to be counted twice, thereby maintaining the correctness of the prefix sum as the left child contains the sum of all of the elements up to the left child of its sibling without any overlap. As such, my algorithm starts at the top of the tree in the level after the root node since the root node does not have a sibling, and then adds the value in the left child to the value in its sibling's left child, which will be half of the offset away since the sibling is the full offset away. The offset is carried over from the up sweep stage, so its value at the start of the down sweep is guaranteed to contain the correct value. This change does not affect the total number of operations performed, so the time complexity of the algorithm is still overall O(n).

Parallel Prefix Scan - Two Level Algorithm

In the case that the two level algorithm is selected, a chunk size is determined by dividing the input array size by the number of threads. Each thread then performs a serial prefix scan on a chunk of the input array, starting at the thread id times the chunk size and ending at the minimum of its start plus the chunk size and the size of the array. Each thread then saves the value at the last index of its chunk into the intermediate array discussed earlier, and then the work efficient prefix scan is executed by all of the threads on the intermediate array. Afterwards, each thread takes the value in the intermediate array at the index one less than the one that they previously populated, and then the threads add that value to all of the threads in their chunk from earlier. The code for this algorithm can be found in the function called `twoLevel`.

My Barrier vs Pthreads Barrier

The main difference between my barrier implementation and the pthreads barrier implementation is that the pthread barrier blocks waiting threads while my barrier has waiting threads spin. Internally, the pthreads barrier uses a futex and a round counter to accomplish its task of having waiting threads sleep and knowing when to wake and release the threads waiting on the barrier. My barrier instead uses two pthreads spinlocks and an atomic counter. Upon initialization, the second spinlock is locked and the atomic counter is initialized to the specified number of threads. As threads arrive, they will attempt to take ownership of the first spinlock, decrement the atomic counter, release the first spinlock, and then try to take ownership of the second spinlock. The exception to this is the thread which decrements the atomic counter down to zero, as it will instead just unlock the second spinlock, while keeping the first spinlock locked. All of the threads inside of the barrier will now begin atomically incrementing the counter back up to the specified value, where the thread that increments the value for the last time locks the second spinlock and releases the first spinlock, resetting the barrier to its original state. This two spinlock design combined with an atomic counter prevents any race conditions from threads entering the barrier while the other threads are exiting while also ensuring the correct behavior for the barrier.

The tradeoffs between the two implementations are similar to the classic tradeoffs between spinning and blocking. In cases where the threads need to wait on the barrier for very short periods of time, such as when the work that the threads are doing is fast, the spinning barrier may be more beneficial as it avoids the overhead of having the threads be added to and taken from the thread scheduling system. However, in cases where the time that threads spend waiting on the barrier is longer or if there are lots of threads waiting on the barrier, the spinning implementation is much more wasteful in that it wastes cycles spinning and all of the threads will fight for the spinlocks at once, causing a lot of contention for the lock. Another downside of my spinning implementation is that if there are more threads than cores on the machine, then the performance can be significantly hurt if the remaining threads that have yet to reach the barrier get preempted as then the machine will need to run multiple tasks which do nothing but spin before cycling back to the thread with remaining work. The effects of these tradeoffs are demonstrated and discussed in more detail in the performance results section.

Performance Results

Testing Environment And Procedure

All data presented in this section was gathered on a system running Ubuntu 20.04.6 with the GNU/Linux 5.15.0-127-generic x86-84 kernel. The processor used was a Intel i5-14600 with 20 cores, a 336 Kib L1 data cache, a 224 KiB instruction cache, and a 15 MiB L2 cache. The system also had 31 gigabytes of memory. All code was compiled with g++ 9.4.0 using the C++ 17 standard and the O2 optimization flag. The Boost 1.71 version of the program options library was used for command line argument parsing.

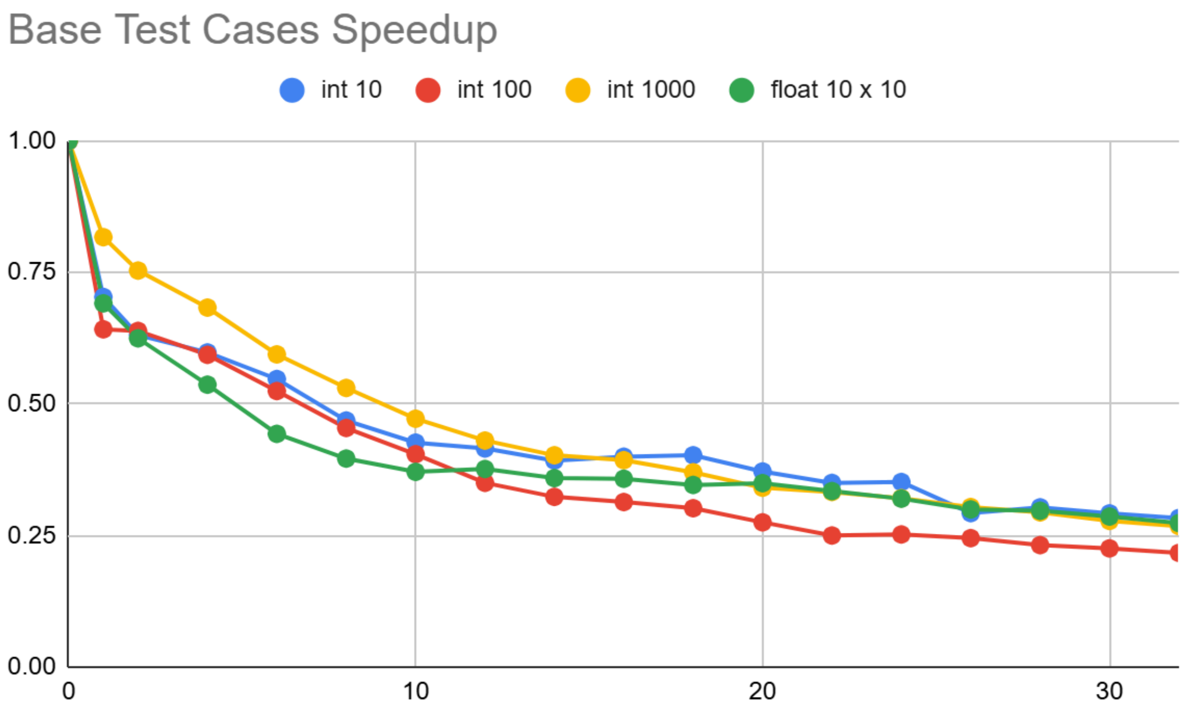
All timing for this program was done using the C++ chrono library using the high resolution clock. The starting time of the program was taken immediately upon entry into the main function, and the ending time of the program was taken right before the last print which prints the difference between the starting and ending point to standard output, encompassing the full end to end execution time of the program. This means that argument parsing, reading in input, and writing to the output file are all included in the timing measurements.

Testing was composed of three benchmark suites. The first suite was the set of provided test cases consisting of a list of 10 integers, a list of 100 integers, a list of 1000 integers, and a list of 10 floating point vectors with 10 floating point numbers in each. The second suite was composed of lists of hundreds of thousands of integers, and the third suite was composed of lists of tens of thousands of 200 dimensional floating point vectors. For the last two suites, the input sizes were given as multiples of 10 and powers of two in order to determine if the vector resizing performed by the algorithm was having a significant effect on the runtime. All three of these benchmark suites were run under four configurations: the work efficient algorithm with the pthreads barrier, the work efficient algorithm with my spinning barrier, the two level algorithm with the pthreads barrier, and the two level algorithm with my spinning barrier. Each of these configurations was run using a bash script which first warmed up the relevant input and output files by running the program on the given input and output files with the relevant flags and arguments a serial run of the specified configuration ten times. After that, the program would run each test case in each suite, first with the serial flags, then with one thread, and then going from two threads up to thirty two threads in increments of two, with each of these being run ten times. Afterwards the times of the ten runs were averaged to determine an average runtime for each combination of test case and thread count.

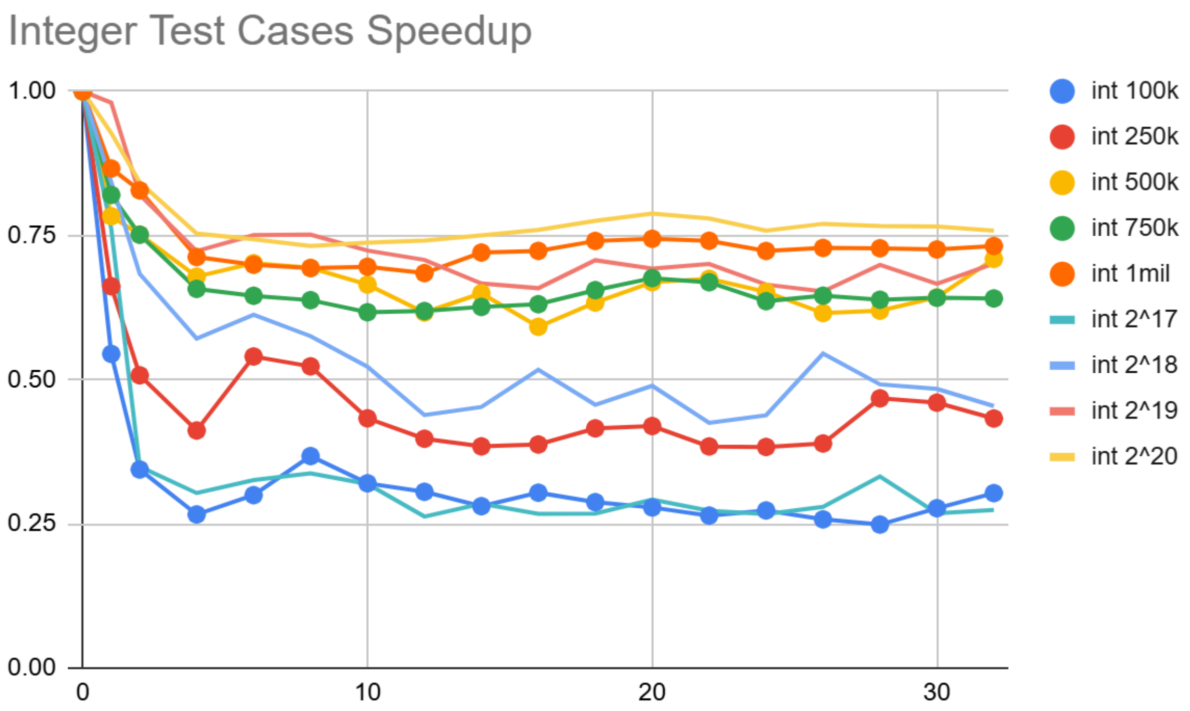
All of the graphs presented in this section have the speedup of the relevant configuration compared to the serial algorithm on the Y axis and the number of active threads on the X axis. Naturally, this means that there will always be a value at (0,1) as 0 threads spawned indicates that the sequential algorithm was used, and the result of dividing the average sequential runtime by itself will just be one.

An important note to make here is that the following analysis compares speed ups for various benchmarks and configurations. Unfortunately, no configuration was able to achieve a meaningful speedup across the different benchmark suites, so some result being described has having a greater speedup over another can also be thought of as having the performance of that configuration be closer to that of the serial algorithm, indicating better performance in comparison to the other discussed benchmark or configuration.

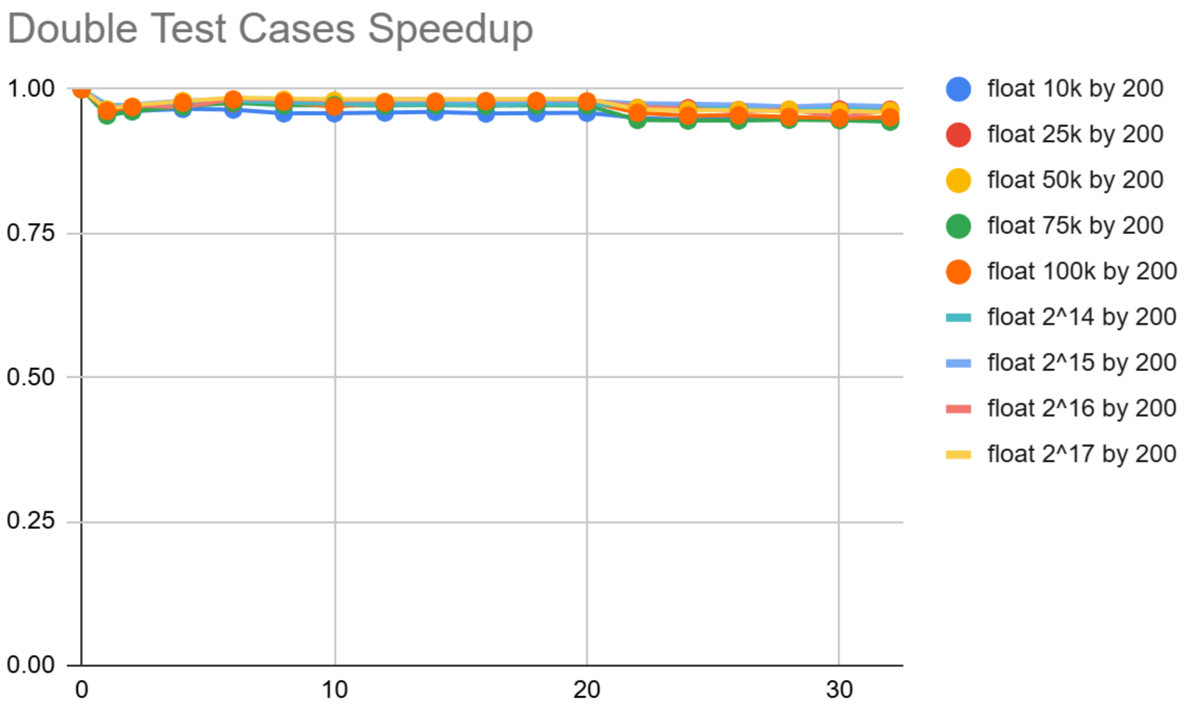
Work Efficient Algorithm With Pthreads Barrier



Above are the results for the provided test case suite. The general trend is that increasing the number of threads spawned is negatively correlated with a program speedup, with all thread numbers leading to runs with longer runtimes than the serial implementation. This is true for both the integer test cases and the floating point test case, where the increased work of vector addition not being significant enough to see a speedup with an increased number of threads. This is somewhat expected given that all of the test cases in this suite are very small, so the overhead of communicating with the operating system for tasks such as spawning threads and blocking on the pthread barrier is quite significant compared to the actual work of summing integers or small floating point vectors.



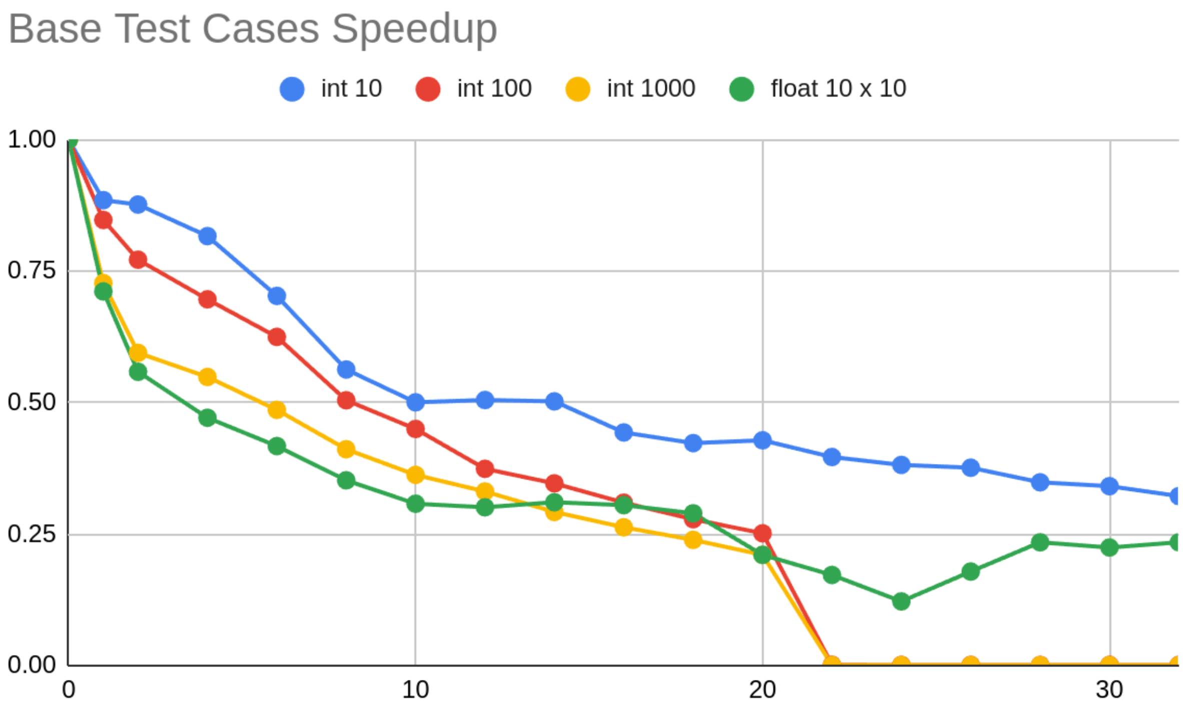
To address the issue of the small input sizes and the fact that some of the smaller test cases in the first suite were creating threads which were not doing any work, the second test suite created some significantly larger test cases where this would not occur, and there would be greater opportunity for each of the threads to do more work. Here the trend continues in that the serial prefix scan algorithm performs better than using any number of threads. Very small thread counts also see better performance compared to larger thread counts, although with larger numbers of threads the speedup remains roughly equivalent as the number of threads increases. However, a notable trend in this data is that larger input sizes see a greater speedup across almost all thread counts, likely due to the fact that there is more work that can be done in parallel between each of the barrier synchronizations at each level in the binary tree structure that the algorithm relies on. Despite this, no number of threads yields a speedup over the serial algorithm across any of the benchmarks in the test suite. One of the reasons for this carries over from the previous benchmark suite, in that the serial algorithm entirely avoids the overhead of creating threads and synchronizing on barriers. However, a second reason for this could have to do with the differences in data prefetching between the serial and parallel algorithms. In the serial algorithm, the code is always working with adjacent pairs of elements and always just using the next element in the array in each new interaction of its loop. Prefetching the next element in the array is significantly easier to predict and execute compared to the varying distance jumps of the parallel algorithm, especially since the jumps first increase in size and then decrease in size. As for the inputs which are lists with a size that is a power of two, they do see a slightly greater speedup compared to the non power of two inputs, which could be correlated both to the fact that they don't need to resize the input vector and due to them having slightly more elements than their closest non power of two neighbor inputs.



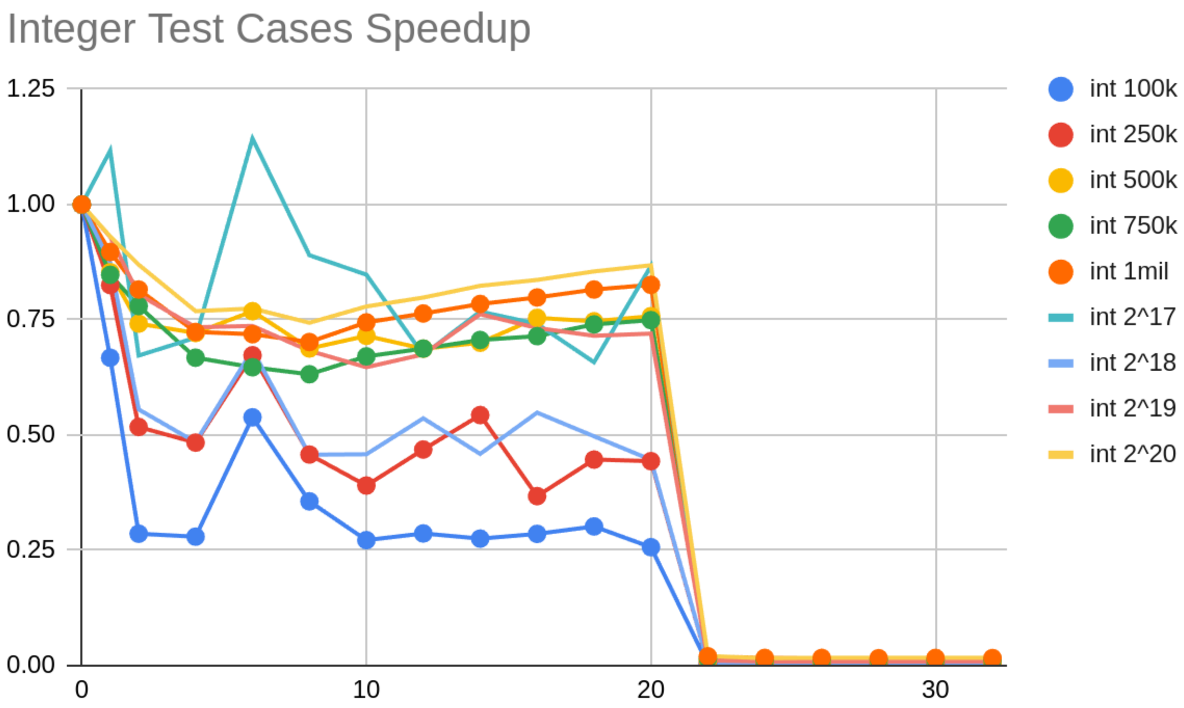
Just as is the case with the previous two benchmark suites, the floating point benchmarks still see no speedup compared to the serial implementation regardless of the number of threads or the input size. However, all of the floating point benchmarks see their performance be significantly more similar to the serial implementation. This is a departure from either of the previous benchmark suites, as all of the previous benchmarks generally saw a large difference between larger numbers of threads and the serial implementation and small numbers of threads and large number of threads. Here, the speedup remains roughly the same regardless of the thread count. One potential reason for this is that that these benchmarks use relatively large dimensionality floating point vectors, so the amount of time to complete the scan operation is much greater and the difference between the scan operation work and the synchronization work is much smaller compared to the integer addition and small float vector addition of the previous benchmark suites.

In conclusion, while the work efficient algorithm with the pthreads barrier does allow for the processing of different parts of the input vector in parallel, this does come at the cost of greater overhead and worse cache performance which leads the parallel implementation to not offer a speedup over the serial algorithm.

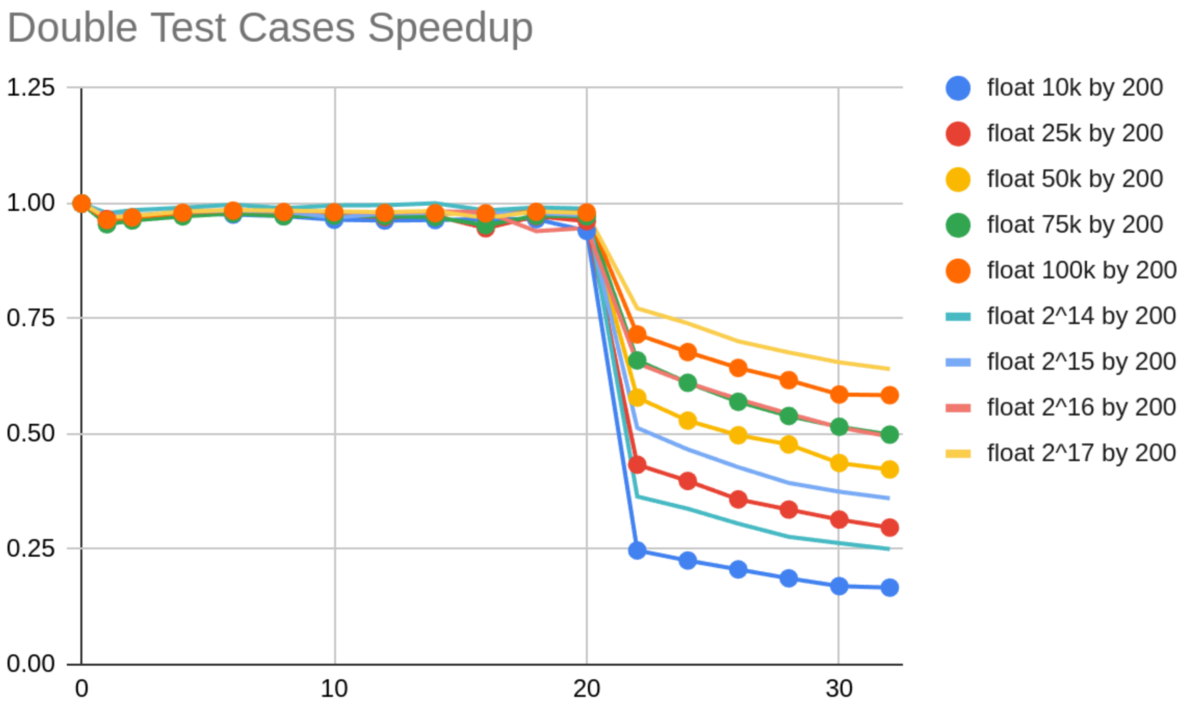
Work Efficient Algorithm With Spinning Barrier



As expected, when thread counts are small and the operations being performed are fast as is the case with the smaller of the two integer test cases, the speedup of the spinning barrier implementation exceeds that of the blocking pthreads barrier implementation. However, even this does not offer a speedup over the serial algorithm. However, when workloads are larger or the operations require more time, such as with the larger integer workload and the small floating point vector, the spinning barrier sees a lesser speedup than the pthreads barrier. Another important trend to notice is that once the number of threads exceeds the number of cores on the machine, the performance drops dramatically as all of the waiting threads are spinning instead of blocking, meaning that if the last threads which need to reach the barrier get preempted, then the scheduler must run several threads which do nothing but spin before those remaining threads can run again, significantly hurting performance. This effect is not seen for the benchmark with ten integers and ten floating point vectors because the extra threads past the input size just return immediately, so they do not contribute to the contention for the spinlocks.



The integer test cases for this configuration generally follows the trend of the integer test cases for the work efficient algorithm with pthreads barriers in that small numbers of threads show a better speedup than larger numbers of threads, and speedup doesn't really increase much as larger numbers of threads are added. Once again, test cases with larger numbers of elements perform more closely to the serial algorithm compared to the test cases with smaller numbers of elements. The data also follows the same trend as the first benchmark suite in that performance drops dramatically after the number of threads exceeds the number of cores on the machine. However, a somewhat interesting trend is that for one of the test cases, there is a speedup with one thread and six threads, and several threads see a large spike in performance at the six thread mark. While this trend is interesting, the speedup is generally pretty minimal, so it is most likely caused by variation rather than anything special about six threads. Similarly to the pthreads configuration, synchronizing on a barrier and spawning additional threads introduces overhead, and larger test cases give threads more work to do before they have to synchronize on a barrier which is why their speedup is generally better than that of the test cases with smaller numbers of elements. Additionally, test cases with larger numbers of elements with the spinning barrier see slightly better performance than the same test cases with the pthreads barrier, with this potentially being caused by the fact that acquiring and releasing a spinlock is much faster than blocking and waking up threads. However, this effect is relatively marginal and does not offset the massive decrease in performance once the number of threads exceeds the number of cores on the machine.

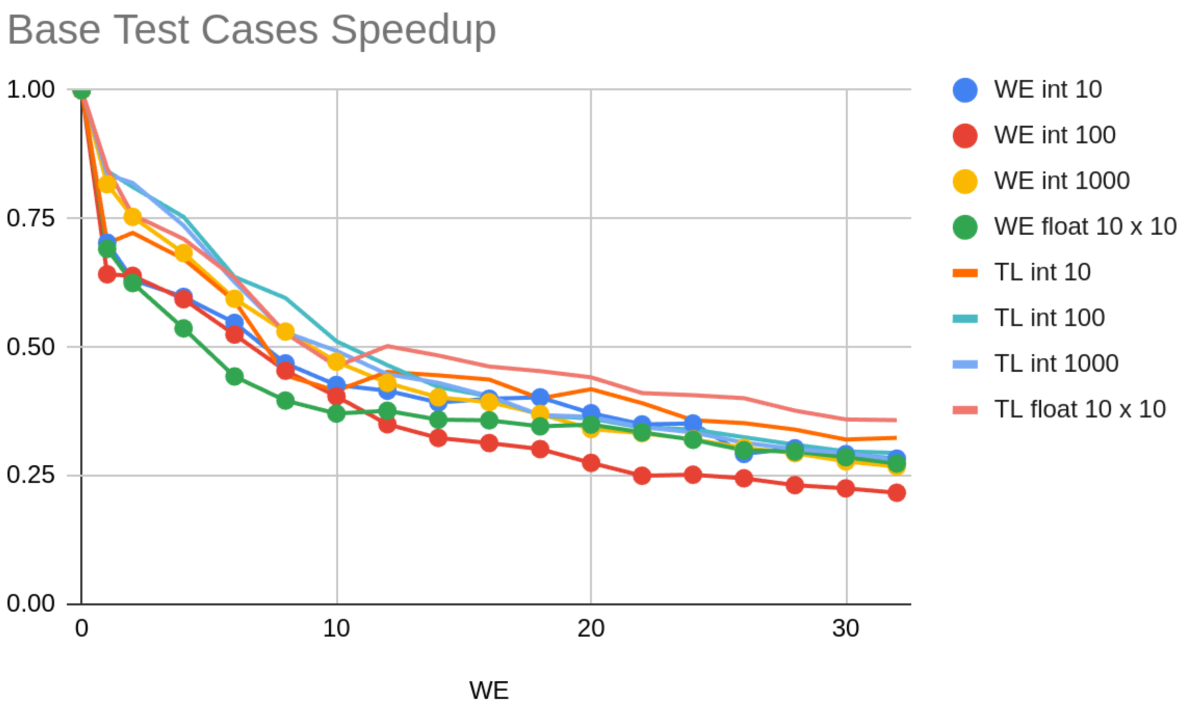


Unlike the integer benchmark suite, the floating point benchmark suite doesn't display any significantly different behavior from the work efficient algorithm with the pthreads barrier. While the serial algorithm still continues to be the fastest, the parallel algorithm is generally only a little slower for all thread counts, with adding additional threads not having any significant effect on the speed up until the number of threads exceeds the number of cores on the machine, at which point the program slows down significantly once again for the same reasons as with the prior two benchmark suites. An interesting thing to note is that while the performance decrease at this point is drastic, it is not nearly as significant as with the integer benchmarks, presumably because summing 200 dimensional float vectors does take a much larger amount of time than summing integers, so the threads will generally spend a bit more time doing useful work instead of spinning on the barrier and needing to be preempted by the operating system while spinning. Overall, if the number of threads is less than the number of cores on the machine, then the pthreads barrier and spinning barrier perform pretty similarly on this set of benchmarks, but after the number of threads exceeds the number of cores on the machine the spinning barrier begins to cause the program to slow down significantly across all test cases in this benchmark suite.

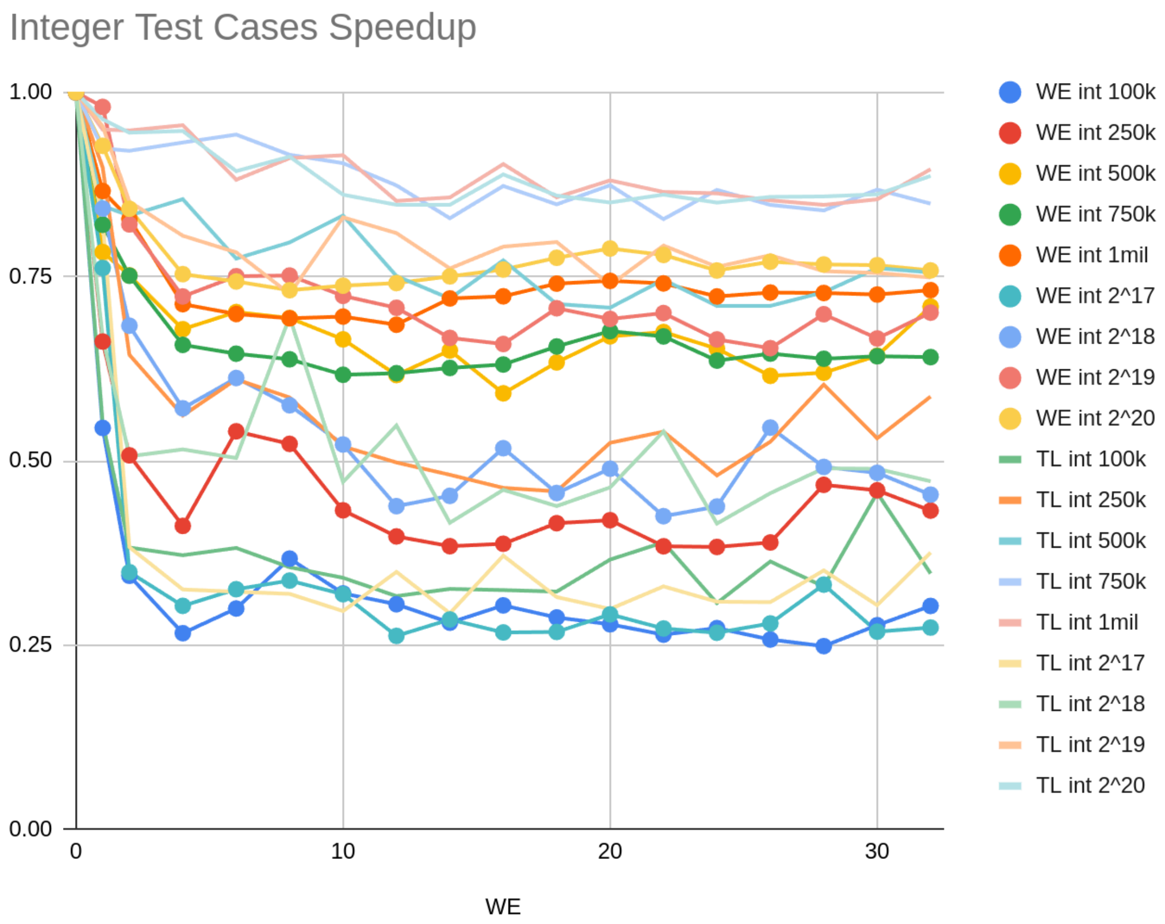
Once again, while the work efficient algorithm with the pthreads barrier does allow for the processing of different parts of the input vector in parallel, and the spinning barrier implemented here does allow for slightly better performance when a small number of threads is used when compared to the pthreads barrier. However, this comes at a significant cost when the number of threads exceeds the number of cores in the system, greatly hurting performance. Overall, even with a spinning barrier, the parallel algorithm still does not offer a speedup over the serial algorithm.

Two Level Algorithm Vs Work Efficient Algorithm

In all of the graphs in this section, WE is used to denote the work efficient algorithm discussed previously, and TL is used to refer to the two level algorithm. The barrier used during this comparison for the following three graphs was the pthreads barrier.

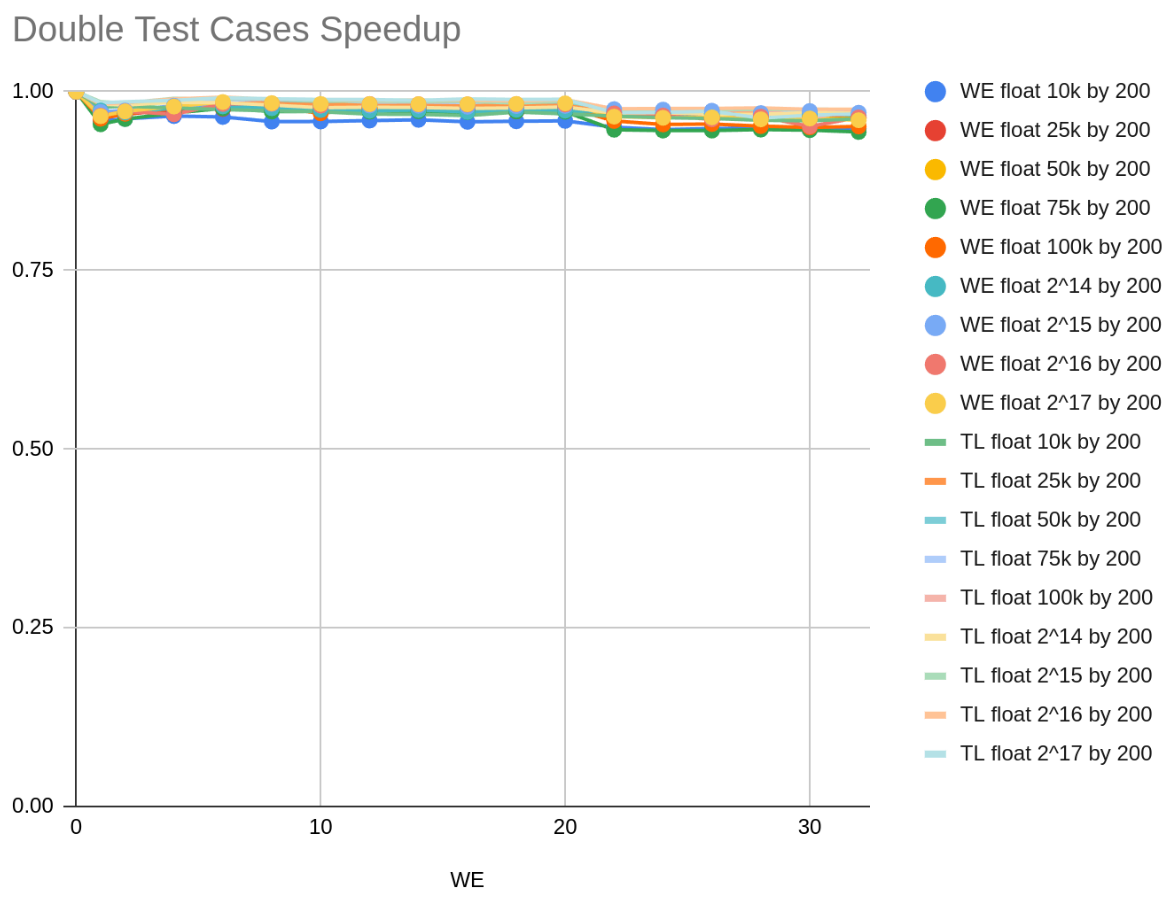


When compared to the work efficient algorithm on the base test cases, the two level algorithm still fails to outperform the serial algorithm regardless of the number of threads, and it also follows the trend of more threads generally seeing less of a speed up compared to smaller numbers of threads. This is likely due to the same reasons that impacted the performance of the standard work efficient algorithm, where the overhead of creating threads and waiting on barriers is quite significant compared to the quick work of adding integers or small floating point vectors. However, a visible trend here is that the two level algorithm does generally outperform the work efficient algorithm on the integer benchmarks. This is likely due to the fact that it is much easier for the data prefetcher to prefetch for the two level algorithm, as the initial prefix scan on each chunk and the fix up of the chunks at the end of the algorithm both make sequential access, which are much easier for the prefetcher to predict than the jumps of the work efficient algorithm.



When it comes to the integer test case suite, the two level algorithm still fails to gain a speedup over the serial algorithm. This likely occurs because the adding integer values is generally very fast, so the overhead of synchronization and running the prefix scan algorithm on the intermediate array ends up having significantly more weight compared to the general work of adding the integers.

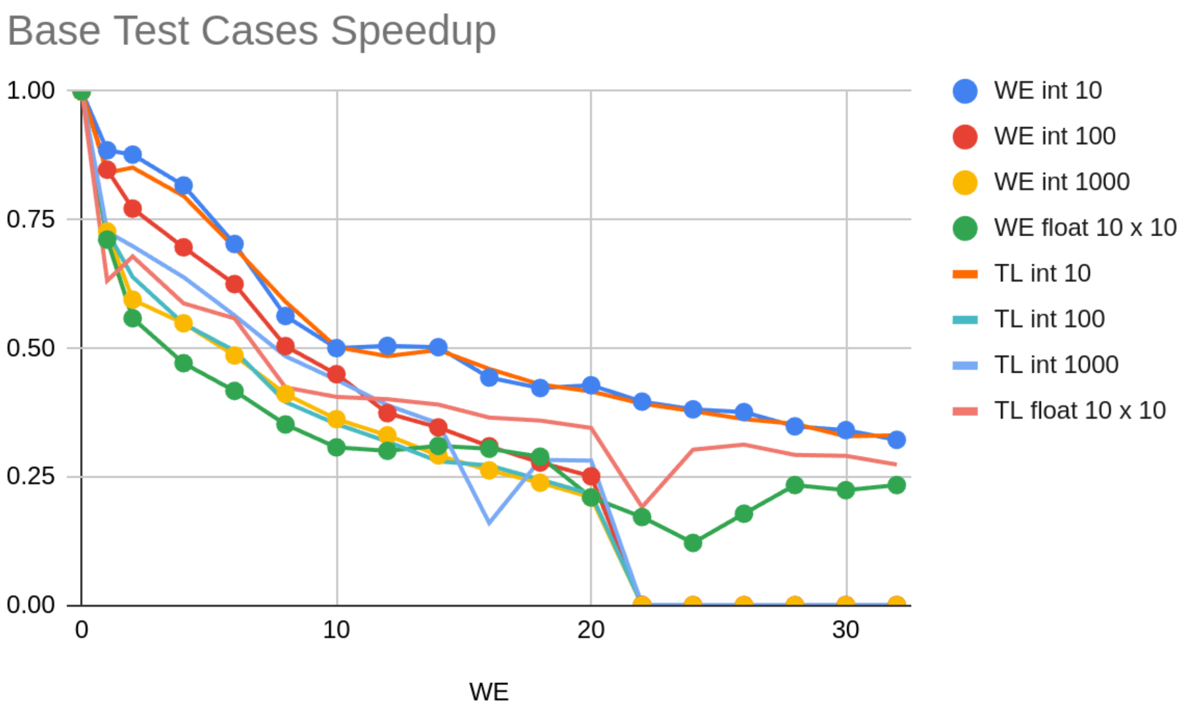
Similarly to the work efficient algorithm, test cases with larger numbers of elements have a greater speedup compared to test cases with smaller numbers of elements. However, the performance of the two level algorithm is generally much closer to that of the serial implementation across the different benchmarks. This is once again likely due to the cache and prefetching benefits of the two level algorithm compared to the work efficient algorithm. As the input size increases, the two level algorithm just keeps making sequential accesses during the initial prefix scan and the last fix up, and when it is performing a parallel prefix scan on the intermediate array the intermediate array is generally quite small. These factors lend themselves well to easy prefetching and great cache efficiency since the working set in the cache is sequential and quite small. This is in contrast to the work efficient algorithm, where larger input sizes mean that it must make more changes to the jumps that it is making to access data, further decreasing cache efficiency and making the data access pattern harder to predict. As the input size grows, this factor plays a larger and larger role, and it is likely the cause of the difference in speedup between the two level algorithm and the work efficient algorithm for these integer benchmarks.

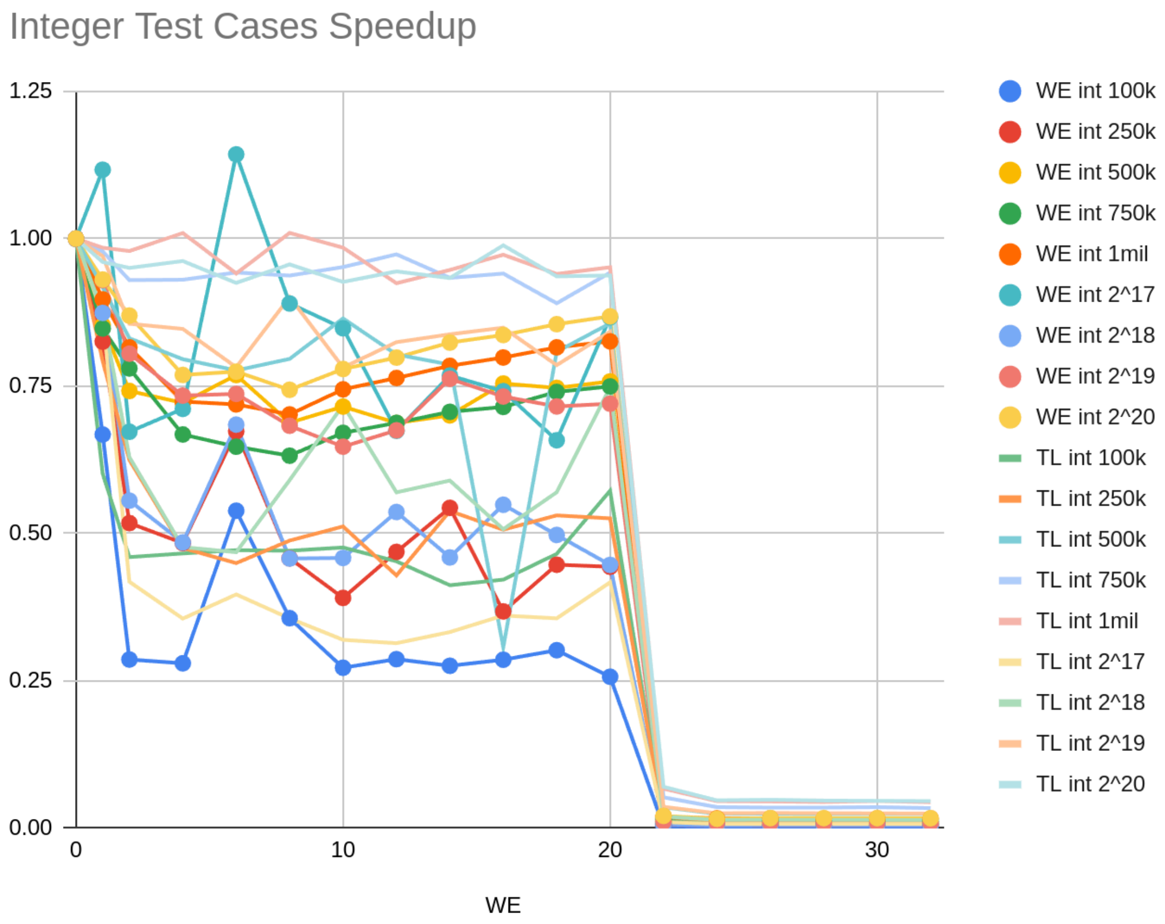


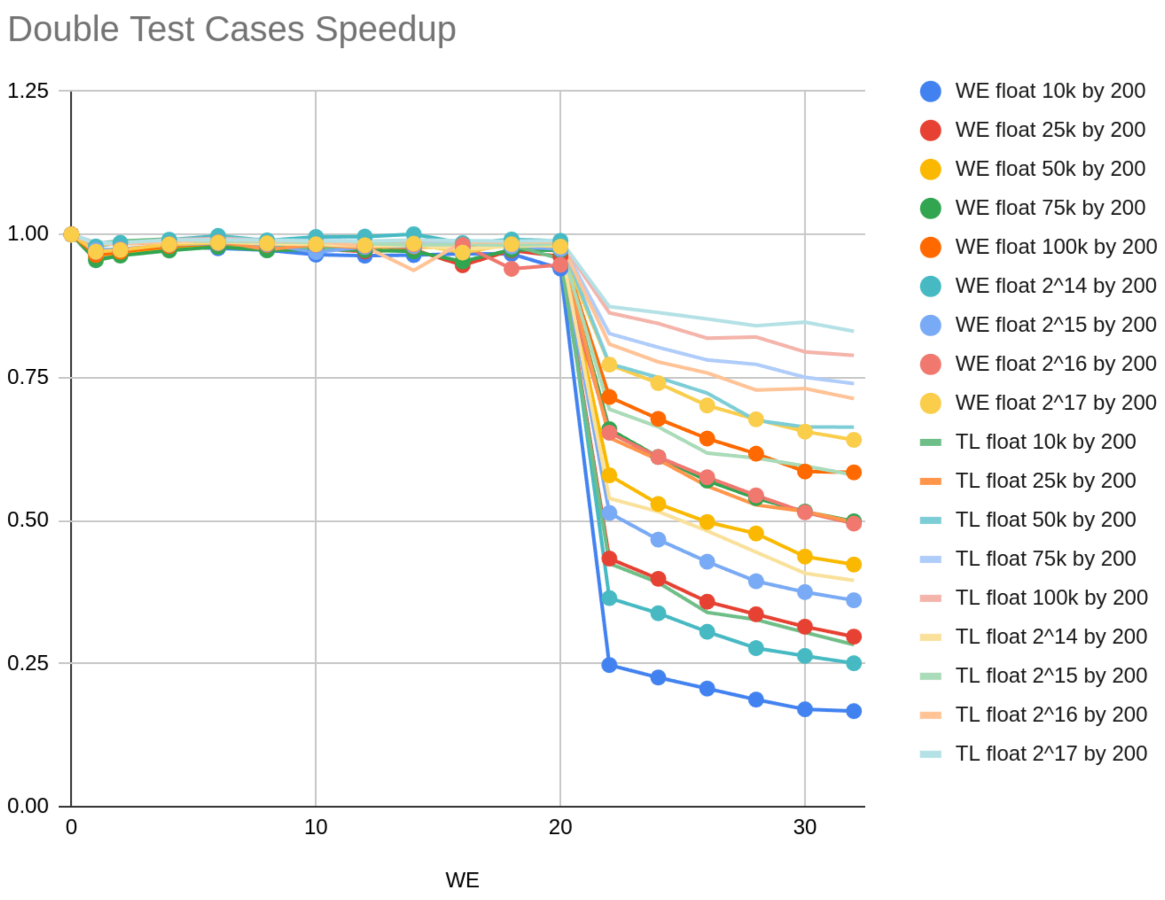
In contrast to the integer benchmarks, the two level algorithm has no significant advantage over the work efficient algorithm when it comes to summing large floating point vectors. Once again, the two level algorithm does not provide a speedup over the serial algorithm, although its performance is generally pretty close to that of the serial algorithm, which is likely due to the same reasons as the work efficient algorithm in that summing high dimensional vectors as is the case with these test cases is a larger amount of work than summing integers, so the comparative difference in work between summing the vectors and performing the synchronization operations is not as significant as the difference between adding integers and performing the synchronization operations. The marginal or nonexistent difference between the two level algorithm and the work efficient algorithm when it comes to summing high dimensional floating point vectors is likely due to the fact that the common work of summing the vectors is still quite significant compared to moving around the data to choose the vectors to sum, and because the two level algorithm is less able to exercise its advantages in cache efficiency when it comes to summing these vectors. Since the data of each individual vector is laid out sequentially in memory, both the two level algorithm and the work efficient algorithm benefit from the easy prefetching offered by this data locality. While the work efficient algorithm may be less cache friendly when it comes to accessing which vectors to sum, this cost is relatively small compared to the work of summing the vectors, hence why the two algorithms see very similar performance across all floating point benchmarks.

Despite the two level algorithm showing some advantages over the work efficient algorithm, it still fails to gain a speedup over the sequential algorithm. The overhead of parallelism ends up being significant enough such that the end to end performance of the program still ends up being worse even with the ability of the two level algorithm to allow different parts of the input array to be processed in parallel.

Testing was also performed with the spinning barrier, but the trends in these graph are not significantly different from the trends present with the pthreads barrier described below, and the spinning barrier also makes it harder to see trends with thread counts greater than the number of cores on the machine due to the two level algorithm and the work efficient algorithm both seeing significant slowdowns in this scenario. The tradeoffs between the spinning barrier and the pthreads blocking barrier still apply and are the same as those discussed in the earlier section on the work efficient algorithm with the spinning barrier. The graphs for this testing are attached here for completeness.


Base Test Cases Speedup
1.00
0.75
0.50
0.25
0.00
0
10
20
30
WE
WE int 10
WE int 100
WE int 1000
WE float 10 x 10
TL int 10
TL int 100
TL int 1000
TL float 10 x 10


Integer Test Cases Speedup
1.25
1.00
0.75
0.50
0.25
0.00
0
10
20
30
WE
WE int 100k
WE int 250k
WE int 500k
WE int 750k
WE int 1mil
WE int 2^17
WE int 2^18
WE int 2^19
WE int 2^20
TL int 100k
TL int 250k
TL int 500k
TL int 750k
TL int 1mil
TL int 2^17
TL int 2^18
TL int 2^19
TL int 2^20



Insights

In conclusion, my parallel algorithms failed to produce a speedup over the serial algorithms for the relevant data types. While these algorithms offered the ability to process different parts of the input data in parallel, the overhead of facilitating this parallelism and the potential effects upon the cache in the case of the work efficient algorithm made the trade off not as beneficial from an end to end program performance standpoint.

One of the main insights that I gained from this lab was getting to see the effect of Amdahl’s Law and the cost of synchronization overhead on a relatively simple concurrent program. Even with something like the two level algorithm which is fairly cache friendly and intuitive in how it can leverage parallelism still failed to see a speedup against the serial algorithm across all of my testing. This showed me that even in cases where the way to parallelize seems straightforward and relatively conflict free, the relevant overheads can still be significant enough to not allow for a speedup. The second insight that I gained was the impact of the length of the task on the benefit gained from parallelism. In my testing, the performance of the parallel algorithms when summing high dimensional floating point vectors was always relatively close to that of the sequential algorithm, and the speedup was not significantly impacted by changing the number of threads. I found this quite surprising as I expected longer operations to benefit more from parallelism, and while the floating point vectors did reach performance levels which were much closer to the serial algorithm, there was never an observed speedup when summing those vectors.

References:

1. https://graphics.stanford.edu/%7Eseander/bithacks.html#RoundUpPowerOf2
2. https://www.youtube.com/watch?v=1G8CZioSjnM
3. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda
4. https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf
5. https://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf
6. https://en.m.wikipedia.org/wiki/Prefix_sum