OS Project 5 Write-up

Design of Solution

Our solution implements the Hillis Steele Algorithm, meaning we sum the numbers in the input in steps. Each time we will sum the current number with the number 2^{Step} spots to its left. All the numbers of the input are split up among the given number of threads and are summed concurrently. Between each step of the algorithm, we use a two-step barrier to sync up the threads so we don't move on to the next step of the algorithm until all threads have completed the previous step. When 2^{Step} is greater than or equal to the input size, we have completed the summation and the resulting sums of each thread combine to be the inclusive sum of the input.

Division of Data Among Threads

In our solution, each thread is given the full input array. From there we logically divide the data based on the following: Before we create the threads, we create an `offset` variable that is equal to the element count divided by the thread count, if the result of this is not a whole number, we cast to an int and add one. Then when the threads are created they are each assigned an ID ranging from 0 to the length of the input array minus one. Now each thread can calculate the data it should work. The start index of its data is equal to the thread ID multiplied by the offset, and it will end at the start index plus the offset, or the end of the array.

Concurrency Mechanisms

Our solution uses a two-phased barrier constituted of semaphores and one lock. The barrier works as follows:

When the "wait_barrier" function is called, the first phase of the barrier will be entered. In this first phase function, we first do `sem_wait(&lock)` to obtain a lock to enter the function. Once this is done we check if the current number of completed threads is equal to the total threads we want to wait on. If it is not, we increment the count of arrived threads, then release the lock with `sem_post(&lock)` and then wait on another semaphore called `stop_1` (`sem_wait(&stop_1)`). If we have reached the desired number of threads, we will `sem_post(&stop_1)` in a for loop from zero to total thread count minus one to allow all of the threads to move to phase two, then release the lock. In phase two, we again obtain the lock (`sem_wait(&lock)`) then decrease the thread arrived count and, if not equal to zero, wait on semaphore `stop_2`. If the arrived threads do equal zero, we do a for loop from zero to total thread count minus 1 and perform `sem_post(&stop_1)` to allow all the threads to exit the barrier. We then release the lock and exit the barrier.

In addition to the barrier, a small portion of our thread code is put in a `pthread_mutex_t`, meaning only one thread may execute the lines of code encompassed by the lock at a time. The

lines in this lock serve to copy the elements of the psums array to the elements array at the start of every new step and increase the count of total threads completed. The reason this section must be in a lock is we can not start the next step until the psums have been copied to the elements array, so we can not allow the thread to be preempted/interrupted until it is finished copying the array.

Maximize Concurrency

In order to maximize concurrency, we first ensure as little of our code as possible, reside in locks, meaning only one thread could execute it at a time. This, of course, means the summation of the numbers is done in a way that does not rely on any other threads or a certain order of completion of the threads for each given step. This allows all of the threads to run in any order and concurrently during each step of the algorithm improving efficiency.

In-Depth Code Explanation

Our implementation uses the Hillis Steele Algorithm to sum all of the numbers. Before we initialize the thread, we create two arrays called ‘elements’ and ‘psums’ that contain the values of the imputed vectors. These arrays will be shared between all threads. In addition to these arrays, our threads share the number of threads completed so far, the number of elements, the number of threads, and the offset which is the number of digits each thread should calculate the sum of. This offset is calculated by dividing the element count by the number of threads, then, if the number of elements divided by the thread count isn’t a whole number adding 1 to the offset `((int) (element_count / thread_count) + (element_count % thread_count == 0 ? 0 : 1))`. In addition to these shared variables, each thread is assigned an individual thread id from 0 to the total number of threads minus 1. We then execute the parallel scan function with the specified number of threads. Each thread will then do a for loop starting at their thread id multiplied by the offset until their thread id multiplied by the offset plus the offset. This ensures each thread only operates on its set of data and since the elements and psums arrays are shared these can be run concurrently. In this for loop, I check if the current position minus 2^current step is a valid index of the array. If it is, elements at that index plus elements at the current index and store it in the psums array. We then use a two-phase barrier to wait for all of the threads to finish the current step of the operation. Once all threads have reached the barrier, we increase the step by one then check if 2^local steps would be greater than the element count. If it would be, we exit the thread. If we don’t exit, in a lock we check if we are the first thread to get to this point and if so, copy the values of the psums array to the elements array for the next step. We then unlock and allow the process to repeat until all threads exit. To coordinate threads this design uses a two-stage barrier created using semaphores. This barrier ensures all the threads reach the necessary point before continuing while still allowing them to be run concurrently.

Members

Michael Emerson - memerson
Vishnu Priya Dendukuri - vdendukuri