Parallel Processing Homework 3

Toby Harvey

November 10, 2020

In this homework we wrote a function to convert COO to CSR sparse matrix storage formats, and then a parallel matrix-vector product for both COO and CSR format. The conversion from COO to CSR, can be done in different ways. I describe two ways it can be implemented. Assuming that the matrix is read in COO format sorted by rows (In the ans.mtx file it is sorted by columns, so I resort by rows), a histogram of the number of non-zero entries there are can be created, with a zero value at the beginning of the histogram. Then a prefix sum can be run over the histogram to generate the row pointer array. We could use the parallel implementation of the prefix sum from the previous homework to do a prefix sum on the histogram, but since that is only written for arrays that are a power of two it would need to modified. On way to modify it for a arbitrary length array is the following. Decompose the length n array into sub arrays of lengths that are powers of 2. The first array would be a single element then indices 1 and 2, then indices 3 through 6, then 7 through 14 (all inclusive) and so on. We can call parallel prefix sum on each of these sub arrays serially, and pass the last value of each to the first value of the next. This will decompose the array so that more parallel prefix sums can be called than if we just did a single maximum power of 2 sub array, and then did the rest serially. While I considered doing this, due to time constraints my COO to CSR function is serial. Alternatively the row pointer array can be constructed by writing the index at which rows indices change in the sorted row index array to a separate array. This implementation seems a bit more simple, but I wonder if it would be slower because the possibility of failed branch prediction on the if statement: if(row_ind[i+1] != row_ind[i]).

For the COO SpMV the col and rows and value of each non-zero elements can be obtained easily from the 3 arrays. The corresponding column of an element is the index to the x vector, and the row will index the result. The problem is that if parallelized a element in the result vector could be accessed at the same time leading to non-deterministic behavior, this can be avoided with a lock for each index in the result array. This is a non issue in the CSR SpMV, because each thread can deal with a single row of resulting vector, and therefore there will never be multiple threads attempting to access the same element.

I ran the code on my laptop which as 12 Intel(R) Core(TM) i7-9750H CPUs from 1 to 12 threads. Between 1 thread and 2 threads there was a 1.95x speed up with COO SpMV, and a 2.00x speed up with CSR SpMV. From 1 thread to

12 there was a 5.74x speed up with COO, and a 3.8x speed up with CSR.