Sparse Matrix-Vector Multiplication

using pThreads

Sarat Rallapalli & Zeeshan Siddiqui

1. Introduction

In this assignment we designed and implemented an algorithm for sparse matrix-vector multiplication with several optimizations. SMVM is used in many scientific applications and the matrix involved in this computation is mainly populated with zeros, which calls for a special matrix representation format to reduce useless zero based multiplication with the vector. The matrix is read from the input file in the Matrix Market Exchange Format (MMEF) and is converted to Compressed Sparse Row (CSR) format to do the computations. CSR format indicates exactly which elements of the matrix are non-zero along with their row and column indices represented in a separate array. We used the tool on the TA’s website to generate random matrices and verified our output with the Intel MLK library.

2. Algorithm(s) used


   i. Load Balancing: Here we devised our own algorithm that assigns equal number of non-zero values to each thread at the granularity of a row. It is possible for a thread to contain a little more load than the others, so we tried to approximate the load as much as possible.

b. Cache Blocking Algorithm: Here the goal was to keep a block in the cache and use it as much as possible for the calculation to get optimal performance time. We tried to convert the matrix from CSR format to the Blocked-CSR format using BeBop library but the library kept failing due to some unknown reason. So we decided to write our own conversion method, however, this method would take significant amount of time. Hence, we did not include this in the code that we submitted. Also we did not observe any significant performance improvement using this method. We referred to “Optimizing the Performance of SparseMatrix-VectorMultiplication” by Eun-Jin Im (http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/CSD-00-1104.pdf) for implementing this algorithm.

3. Optimizations

a. Optimization Flags: We used gcc compiler optimizations. The ones we found very useful for performance were -O4, -funroll-loops.

b. If the loop variables and other variables which the repeatedly used in the multiplication kernel are stored in registers, this gives better performance.

c. Use of a do-while loop instead of for loop gives a slightly better performance.

d. Load Balancing is very important when non-zero values in matrix are not evenly distributed.
4. **Performance**

To determine the optimal number of threads to be used so as to get fastest results for a given input matrix and vector, we run the algorithm using [1, 2, 4 … 32] threads and report the best time and number of threads which achieve the best time. We do not test with more than 32 threads because the performance of our code degrades when more than 32 threads are used and the input size of matrix is of dimension between 100 and 10000. Below are performance graphs for 2 different Matrix-Vector combinations comparing the number of threads and time taken by the program to return the result.
Time taken for SPMV on inputs uploaded on the TA webpage is:

1. Matrix 1 * Vector 1 took 10,788 microseconds.
3. Matrix 3 * Vector 3 took 3217 microseconds.

*Each of the timings reported above is median of 8 readings taken.

Load Balancing also plays an important role in speeding up the performance, especially if the non-zero values are not uniformly distributed across the matrix. For example, for multiplication of the test Matrix 4 and Vector 4 (on the TA page), time taken without Load Balancing was 7217 microseconds while after balancing the load among the various threads, time taken was 6089 microseconds.

5. **Difficulties encountered**
   We could not use Bebop library to convert a sparse matrix from CSR to BCSR format which can be used to implement blocking. So we had to code the conversion on our own which turned out to be quite tricky.

6. **Conclusion**
   We believe our algorithm produces correct results in an efficient manner. Over the course of the development we made several strides in improving the performance by taking into account various factors like cache blocking, avoid the use of locks for synchronization and the use of spatial and temporal locality to improve cache hits.