Assignment 3 Report

Dylan Bell 21711951 & Ryan Hodgson 21969062

Introduction

A sparse matrix is defined as being a matrix where most elements are zero. It is very inefficient, in terms of both memory and processing power wise, to store a full sparse matrix in a two-dimensional array. One popular method to store a sparse matrix is in coordinate list format (COO) which stores a list of $\{row, column, value\}$ tuples. This makes matrix multiplication a lot more efficient when multiplying two sparse matrices. Large sparse matrices often appear in scientific, engineering and aerospace disciplines so it very useful to optimise the parallelisation of matrix multiplication.

Our program is designed to read in two files, containing two sparse matrices, and make use of a two-level parallelisation architecture (employing MPI and OpenMP) to efficiently compute the matrix multiplication of these matrices.

Our Program

Our program starts by reading in the two files and counting the number of lines in each. Matrix 1 is then sorted by row and matrix 2 by column.

The application then reads these files into an array of structures in the format $\{row, column, value\}$. The function splitMatricies() partitions the first matrix into (roughly) equivalent sized "chunks" to be sent to each worker node. Each chunk has to contain all of the same rows i.e. if the chunk ends at an entry with row 3 but there are still more entries with row 3, that chunk must continue to include it. It also splits matrix 2 by column corresponding to the largest row in each chunk.

The code below demonstrates how we send the different partitions to each worker node.

```
//send the size of the partition row and corresponding column to the worker so it knows how much memory to buffer for
MPI_Send(&sizeOfPartitionRow, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Send(&sizeOfPartitionCol, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);

//send a pointer to the index of the start of the partition for the row and corresponding column partition
MPI_Send(&matrix1[partitionRowIndex], sizeOfPartitionRow, mpi_struct, dest, mtype, MPI_COMM_WORLD);
MPI_Send(&matrix2[partitionColIndex], sizeOfPartitionCol, mpi_struct, dest, mtype, MPI_COMM_WORLD);

//similarly the worker node will receive each of these messages and store the relevant information in an appropriate data structure
MPI_Recv(&sizeOfPartitionRow, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
```

OpenMP

We came across a high level of difficulty in implementing a suitable algorithm in terms of OpenMP. The first solution we attempted to use involved using a shared result array (list of structs) and then accessing it in incremental order by using a variable called resultNonZeroEntries. However, this variable was required to be atomically incremented so it actually caused a slow down when parallelised compared to the sequential version.

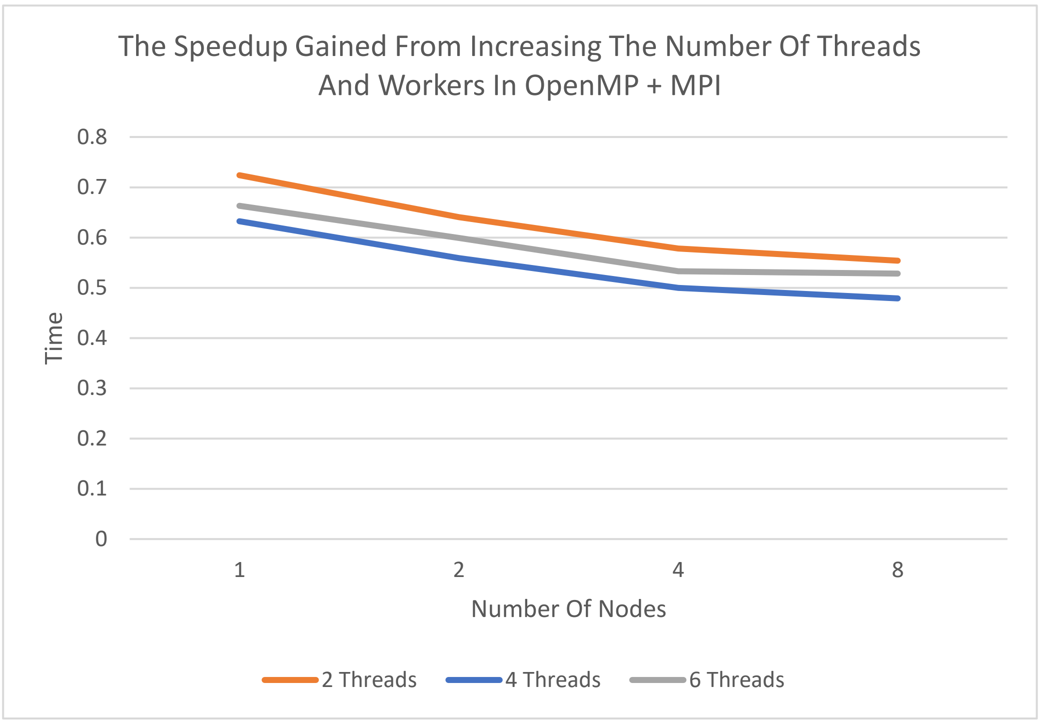
After this we tried to create a large non-sparse local result matrix stored as a 2-dimensional array. We then added each result into this matrix and them summed all the local matrices up at the end. When parallelised properly this led to about a 50% speed up.

We also thought about storing the original matrix in CSR/CSC format which in retrospect would have been by far the quickest way to multiply these matrices and the easiest to parallelise however we realised this too late and could not change out whole program to store the matrices in a different format.

The solution we tried and chose in the end was to instead of atomically increasing resultNonZeroEntries. Like the first method we tried, this process involved saving and adding a list of structs locally. Using a barrier and #pragma omp single we saved all the non-zero entries of this result array and summed up the structs at the end. Although this would lead to some duplicate entries, they were easily removed afterwards. This method led to about a 250% speed up on our local computers.

In the end, we couldn't get this proper OpenMP code working on the cluster. When deployed and tested on our local computer as well as when compared to a sequential algorithm, we achieved a 250% speed up. This function was called matrixMultiplyBroken, we are still unsure why our code didn't work properly on the cluster after several hours of debugging. For the timing of our code we used a very simple parallelised version to get some results, this function was called matrixMultiply.

Results



A total of 12 experiments were recorded and are displayed in the above graph. The number of nodes, starting from 1 was doubled, to a value of 8. Therefore, the number of workers ranged from 12 to 96 (12 workers per node including the master). The number of threads, starting from 2, was incremented to a value of 6. The average times of these values (values were tested 10 times each to gain an average) were recorded and graphed. The matrices used had a maximum size of 10,000 x 10,0000.

As you can see from the above graph the general consensus is that as you increase the number of workers, the time taken to complete the program falls. This is to be expected – the greater the number of nodes, the greater the number of partitions for the given input matrices. Therefore, the less work a given worker node is required to do. This result is to be expected.

There is also a visible “dent” in the graph where the time seems to plateau (where the number of nodes is equal to four). This represents the decreasing returns to partitioning the matrix. A few theories were discussed about the program reaching some sort of timing asymptote – partitioning can only be so efficient and with the limited size of our matrix perhaps we were reaching it.

The results for the OpenMP component are interesting. Considering the method used for thread parallelisation wasn’t the most efficient – we still see a speedup occurring by increasing the number of threads from two to four. Although six threads is more efficient then two it doesn’t outperform four threads. Again, this could be due to a number of factors – the same “plateau” argument was discussed and perhaps the increasing overhead of splitting this matrix up efficiently played a role.

From this graph we can see we gained a slight speedup. It was desired to get the more efficient OpenMP function working which saw a larger, prominent speed up however due to time constraints we failed to implement it.

Conclusion

If we were to repeat this again, we would not store the matrices in a struct. Although it made the code readable, it is a complex datatype which caused us a lot of trouble when it came to memory allocation and MPI_Datatypes. We had to create a complex datatype for MPI using malloc and realloc which caused us quite a bit of trouble.

Furthermore, we would explore and implement more efficient openMP algorithms which employed CSC/CSR sparse matrices. This would have made the multiplication method very easy and given us a greater speedup.