Sparse Matrix Multiplication Utilising Hybrid Parallelisation Techniques

Assignment 3

Dylan S. Carpenter [21982288]

$$\begin{bmatrix} 3 & 1 & 4 & 1 \\ 2 & 7 & 1 & 8 \\ 1 & 6 & 1 & 8 \\ 1 & 4 & 1 & 1 \end{bmatrix}$$

CITS3402: High Performance Computing
School of Computer Science and Software Engineering
The University of Western Australia

2nd November, 2018
2041 words

1 Introduction

Parallel computing allows large repetitive computation tasks to be calculated significantly faster than sequential execution methods and comes in two distinct flavours: shared memory and distributed memory. The combination of these two methods allows for high performance processing of large amounts of data in scientific applications.

One such application is the multiplication of sparse matrices; the foundation of modern computation applications such as deep learning and neural networks. Matrix multiplication algorithms take two matrices as input and outputs a matrix composed of dot products of corresponding rows and columns of the input matrices.

An example of matrix multiplication is seen below below in both the traditional notation and the space-saving Matrix Market notation used in the context of this report and code.

Typical Notation

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 10 & 12 \end{bmatrix} \times \begin{bmatrix} 2 & 5 \\ 0 & 1 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} 1\times 2+0\times 0+2\times 8 & 1\times 5+0\times 1+2\times 0 \\ 0\times 2+10\times 0+12\times 8 & 0\times 5+10\times 1+12\times 0 \end{bmatrix} = \begin{bmatrix} 18 & 5 \\ 96 & 10 \end{bmatrix}$$

Matrix Market Format

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 3 & 2 \\ 2 & 2 & 10 \\ 2 & 3 & 12 \end{vmatrix} \times \begin{vmatrix} 1 & 1 & 2 \\ 1 & 2 & 5 \\ 2 & 2 & 1 \\ 3 & 1 & 8 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 18 \\ 1 & 2 & 5 \\ 2 & 1 & 96 \\ 2 & 2 & 10 \end{vmatrix}$$

In the context of this report and the attached code, the Matrix Market format is used to reduce the memory and bandwidth requirements of handling large sparse matrices. This format expresses only non-zero array elements as a tuple of information in the format `<row, column, value>`.

This report describes the attached implementation of a hybrid sparse matrix multiplication algorithm utilising both distributed memory computation across a cluster of machines as well as shared memory parallelism on a node-by-node basis. Part two of this report describes the algorithms used to implement sparse matrix multiplication, part three evaluates the performance of the algorithms and part four concludes this report.

The attached code can be compiled using the `mpicc` compiler with the following command:

```
$ mpicc -std=gnu99 -fopenmp dmm.c -o dmm
```

2 Algorithms

The algorithms used in the code utilise two programming APIs, the Message Passing Interface (MPI) and OpenMP (OMP). Work is distributed across a cluster of worker computers though the use of MPI, with a singular node acting as a master controlling the cluster. On a node-by-node basis, OMP is used to parallelise the computation of matrix products using multi-threaded shared-memory techniques. The specific algorithms used by the Master and Worker nodes are described below.

2.1 Master Node

The main roles of the master node are to read the two matrices which are being multiplied, distribute the data between the node in the rest of the cluster and collect the results from completed worker nodes. Descriptions of these steps are as follows:

Input data

Data is read from two specified files, each containing a matrix in the matrix market format, sorted by either the row or the column numbers. Additionally, the lengths of the two files should be specified. In the current implementation of the program, a bash command is used to retrieve the length of said files as well as sort them. As the size of the input increases this becomes increasingly impractical and would ideally be completed outside the program or be computed in a parallel way.

Matrix Segmentation

The principal responsibility of the master node is to distribute the input data between worker node in an even manner. There are many ways to do this with varying implications on algorithm performance. Since the summation and merging of output products is calculated at the worker node level prior to returning, it is vital that a singular node receives all matrix elements residing in the same row/column. To ensure this, the master node determines the largest index for rows in the input matrix and uses this split up the input data. This is analogous to assigning rows of the fully expanded matrix to each node and giving that node all elements, which exist in those rows:

Matrix Market Format

$$A = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 3 & 2 \\ 2 & 2 & 10 \\ 2 & 3 & 12 \end{vmatrix} \quad B = \begin{vmatrix} 1 & 1 & 2 \\ 1 & 2 & 5 \\ 2 & 2 & 1 \\ 3 & 1 & 8 \end{vmatrix}$$

Expanded Matrix

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 10 & 12 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 5 \\ 0 & 1 \\ 8 & 0 \end{bmatrix}$$

With two worker nodes, w_1, w_2; red rows are sent w_1, blue to w_2 and black to both.

This method does not provide exactly even distribution of input lines, however will approach equality in matrices where the distribution of elements is approximately uniform. The primary advantage of this method of distribution is that the computational cost of merging together and summing cross product elements, an $O(n^2)$ operation, is distributed across multiple nodes rather than concentrated in the master node.

It should be noted that only the largest of the two matrices is divided into pieces, with the smaller of the two input matrices being distributed in its entirety to all worker nodes. This ensures that all product entries can be calculated such that the worker with the n^{th} row/column of the input matrix will evaluate the n^{th} row/column of the output matrix

Interaction with nodes

The master node interacts with workers throughout the cluster by utilising the `MPI_Send()` and `MPI_Recv()` function calls. These blocking methods send a pre-specified amount of data to a worker node who has corresponding `MPI_Recv()` and `MPI_Send()` function calls. The master node's actions can be described as below.

```
calcInputPartitions(matrix1, num_workers)

//send all data
for worker w
    send(size_of_Partition(matrix1, w))
    send(Partition(matrix1, w))
    send(matrix2)

//receive all products
for worker w
    receive(size_of_partial_product)
    receive(partial_product)
    write(partial_product)
```

2.2 Worker Nodes

Worker nodes within the cluster receive input data from the master node and perform individual matrix multiplication operations on the given data. The products from this minor matrix multiplication are then sent back to the master to be output to some filestream.

The operation performed by each node is an independent and distinct product operation such that the output from each node is not only the product of the input data, but a distinct entry in the overall product matrix. This is achieved though the careful segmentation of data described in Section 2.1.

To accelerate this product operation at a node-by-node level, shared memory parallelism is utilised through multi-threading. OMP library calls create a parallel region in which threads compare every entry in the first matrix to every entry in the second and emit their product if their column/row numbers match. Following this, threads take turns to merge their emitted products into a shared pool and sum those corresponding to identical entries in the result matrix. The shared pool of entries is then sent back to the master node. This method is described below in pseudocode below. Note that step (1) occurs in parallel whereas step (2) is executed one thread at a time.

```
//variables
(List) Pairs
(List) Output
//receive data
input (Matrix) A
input (Matrix) B

//(1) find and emit products of matching rows/columns
for a in A
    for b in B
        if a.column = b.row
            Pairs.add(a*b)

//(2) merge and sum matching pairs
for p in Pairs
    if Output.contains(p.coordinates)
        Output.get(p.coordinates) += p
    else
        Output.add(p)

//output matrix product
output Output
```

3 Performance Analysis

The performance of this algorithm can be considered in two ways: theoretically through complexities and through rigorous testing.

3.1 Time Complexity

Assuming that data is input to the master node re-sorted, the complexity of the master nodes execution is based on both the sizes of the input, the number of workers and the time complexity of the workers' tasks. Given the input is two lists of length n, the total worst-case complexity for reading and partitioning the data is only $O(n)$. Additionally, if there are w worker nodes who each operate in $O(t)$ time, then the overall complexity of the master node is $O(nwt)$.

The worker nodes must iterate over all elements in both lists, and assuming the worst possible data partitioning will hence run in $O(n^2)$ time. Therefore, the overall complexity of the entire operation is $O(wn^3)$. This is not optimal and could be reduced through several other algorithms and partitioning schemes. Also note that the time spent transmitting data and the time processes spend blocked are considerable and surpass the computation time for small matrices.

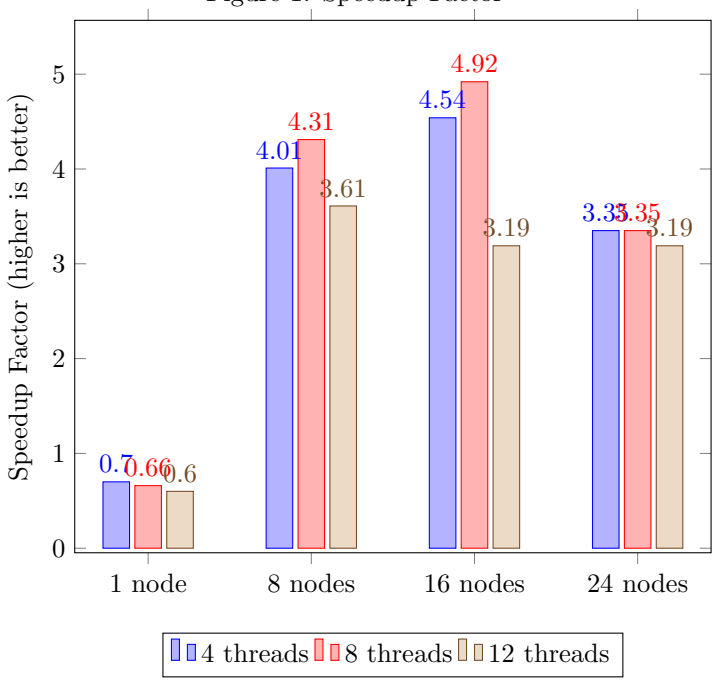
3.2 Experimental Results

The algorithm was tested extensively on a cluster of machines utilising several different numbers of worker nodes utilising several different numbers of threads. The test matrices were square matrices of size 1138, with a densities of less than 1%. The collected times, averaged over 100 trials, are compared to the a baseline time calculated from sequential execution and the speedup factor, $sf = \frac{sequentialtime}{parralleltime}$, is calculated.

Table 1: Speedup factor of differing worker and thread counts

Worker Nodes	Threads (per node)		
	4	8	12
1	0.70	0.66	0.60
8	4.01	4.31	3.61
16	4.54	**4.92**	4.61
24	3.35	3.35	3.19

Figure 1: Speedup Factor



It appears that the optimal choice of cluster size and thread count for matrices of this size are a cluster size of 16 nodes each running 8 threads. This is likely due to the high cost of sending data across the cluster and the relatively small size of partitioned data. As the cluster size grows, the data transfer time increases and the size of data partitions shrink. This amounts to an overall longer execution time. Additionally, as the thread count increases, the overhead involved in spawning, managing and synchronizing the threads increases exponentially, and as such, poor performance can be observed when either the number of threads in high or when the cluster has too many nodes.

4 Conclusion

The matrix multiplication algorithm presented operates efficiently for sparse matrices and allows for the parallel, distributed computation of matrix products. Compared to other existing algorithms such as the SUMMA or Fox algorithms it performs quite poorly, however this is also a product of the format of the matrices. This algorithm has particularly large room for improvement in the partitioning of data, and as such other methods should be explored. In addition to this, non-blocking send and receive operations could be employed for the return of data such that all nodes are not held up by a defective node, or alternatively a map-reduce-style task allocation system could also be employed.

The presented algorithm makes light work of small matrices, however its effectiveness does not scale efficiently with input size and has much room for implementation improvement including fault tolerance and dynamic memory allocation of output buffers. Further exploration of this topic is required and may provide useful speed benefits to modern computations.