22355071

Project One Report Submission

(CITS3402 – High Performance Computing)

2019 Semester Two

Summary: Operations on matrices such as addition, multiplication, transposing and calculating trace are carried out using OpenMP parallelisation.

Three types sparse matrix representations are used in this project.

1. Coordinate Format (COO)
   It is used for calculating the trace of a matrix, in scalar multiplication of a matrix and addition of two matrices.
2. Compressed Sparse Row Format (CSR)
   It is used in matrix - matrix multiplication along with CSC.
3. Compressed Sparse Column Format (CSC)
   It is used in getting the transpose of a matrix and also for matrix – matrix multiplication along with CSR.

Scalar Multiplication (COO)

The parallelism implemented for scalar multiplication are: (scalarMul.c)

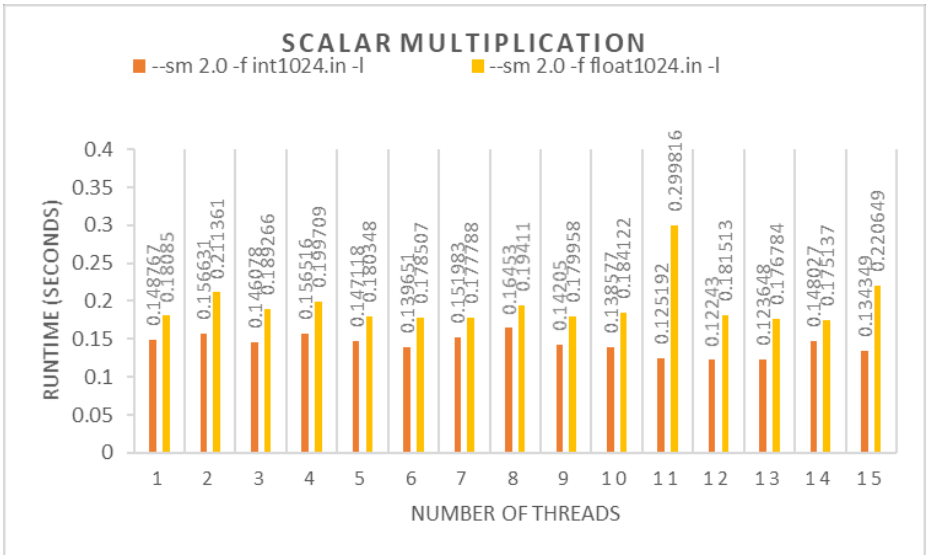
#pragma omp parallel { This directive forks additional threads to carry out the work enclosed in the block following below

#pragma omp single { This directive makes sure there is no data race as we are assessing the non-zeros values sequentially for multiplication. Whatever code present within this construct is executed by one thread only and all the other threads wait until current thread finishes executing this construct.

#pragma omp parallel for collapse (2) This directive is followed by two for loops and it collapses these loops which are used for printing and checking if the row and column coordinate values match the loops iterators. If there is a match the non-zero value corresponding to the coordinate is multiplied with the input scalar.

} }

The run-time observed for the sample int1024.in and float1024.in is shown below:



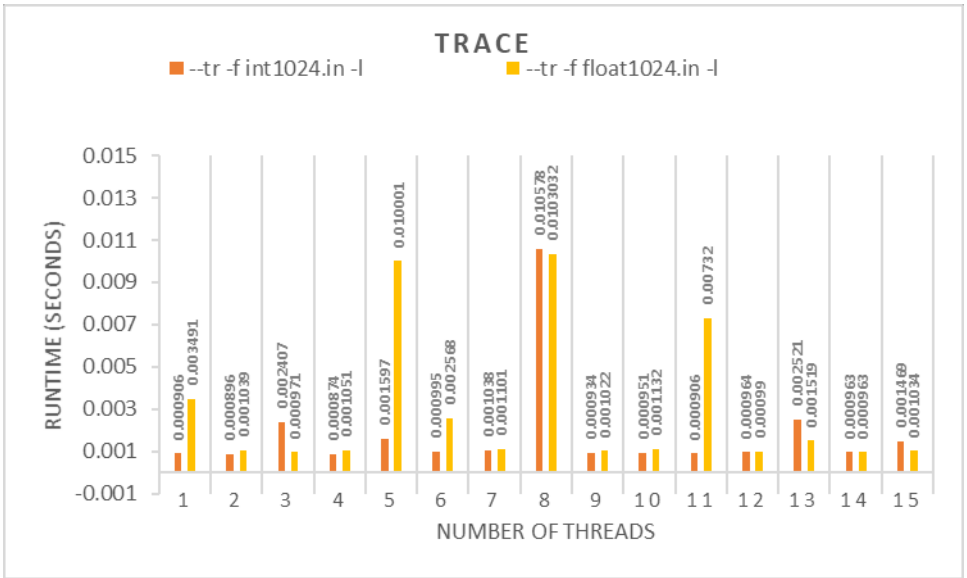
Trace (COO)

The parallelism implemented for calculating trace of matrix is: (trace.c)

#pragma omp parallel for shared (total_nonzeros) reduction (+ : trace_partialSum) { There is a shared variable "total_nonzeros" which will be used by all threads executing the for loop. The variable "trace_partialSum" is also a shared variable but we want to it to be updated only by one thread at a time so the reduction clause is used to hold the result each successive computation. In fact, the master thread spawns a group of threads and the iteration of the for loop is shared amongst these threads. Each thread holds its own successive computed value and combine the value from all threads when they all join together.

}

```
int trace_partialSum = 0;
for (int i = 0; i < total_nonzeros; i++) {
        if (matrix1_rowindices[i] == matrix1_colindices[i])
        trace_partialSum += matrix1_values[i];
        }
```



Addition (COO)

The parallelism implemented for matrix-matrix addition are: (addition.c)

#pragma omp task shared (variables…) { The shared variables in this construct are number of rows of matrix A and B, number of non-zeros in A and B and the index for iterating non-zeros of both the matrices.

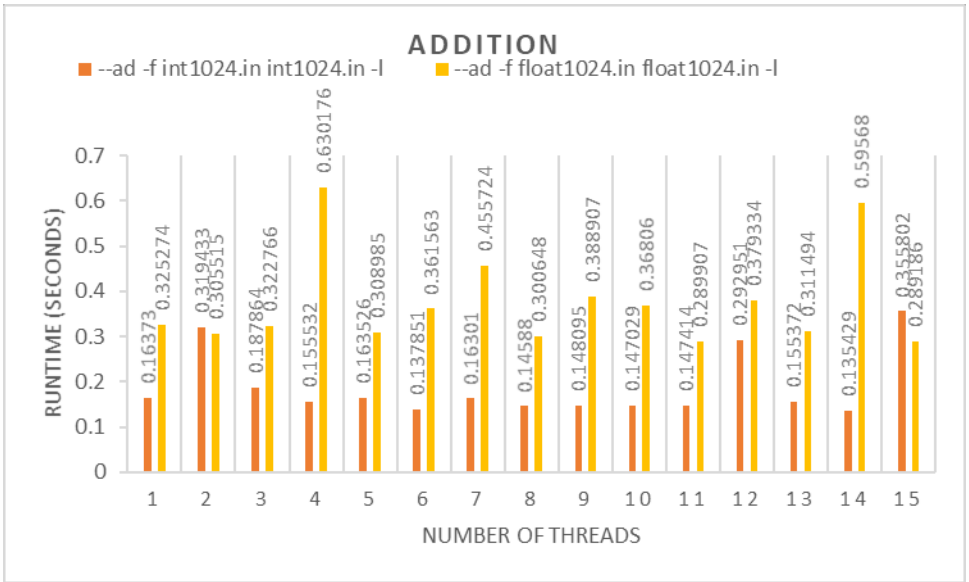
#pragma omp task shared (file1_numRows, file2_numRows, index1, index2,

file1_totalNonzeros, file2_totalNonzeros)

#pragma omp taskwait This construct specifies a wait on the completion of child tasks of the current task. Whenever a thread reaches this region it will wait for the thread that is executing the addition of non-zeros values of two matrices.

}

The run-time observed for the sample int1024.in and float1024.in is shown below:



Transpose (CSC)

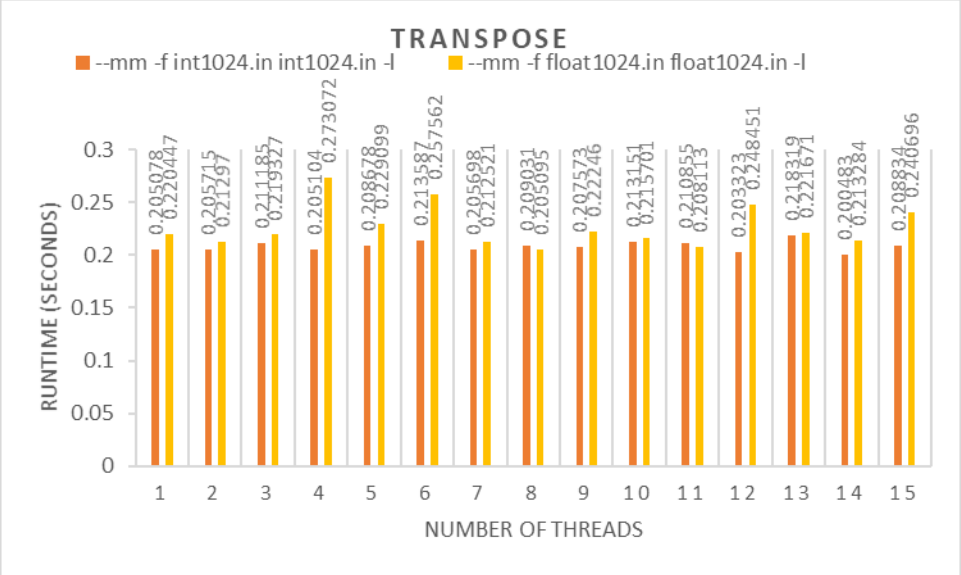
The parallelism implemented for transposing a matrix are: (transpose.c)

#pragma omp task shared (common_index) { The shared variable in this construct is the index of the CSC representation which is used for iterating all non-zero values. The CSC of a matrix is indeed the transpose of a matrix. The task construct creates a task for each thread to carry a comparison between the index of the non-zero values in CSC with the loop iterators for printing the result.

#pragma omp taskwait/critical If-else clause is used in this region to compare index of a matrix and index of the CSC. Taskwait or critical directive is used to prevent data race so that one value is not printed multiple times.

}

The run-time observed for the sample int1024.in and float1024.in is shown below:



Matrix Multiplication (CSR + CSC)

The parallelism implemented for matrix-matrix multiplication are: (matrixMultiplier.c)

#pragma omp parallel shared (variables…) { This construct creates a team of threads and share the variables specified inside the bracket. The non-zeros values of matrix A (first matrix) are stored in CSC format and the non-zeros values of matrix B (second matrix) are stored in CSR format. The row indices in CSC are sorted in increasing order and column indices in CSR format are also sorted in increasing order.

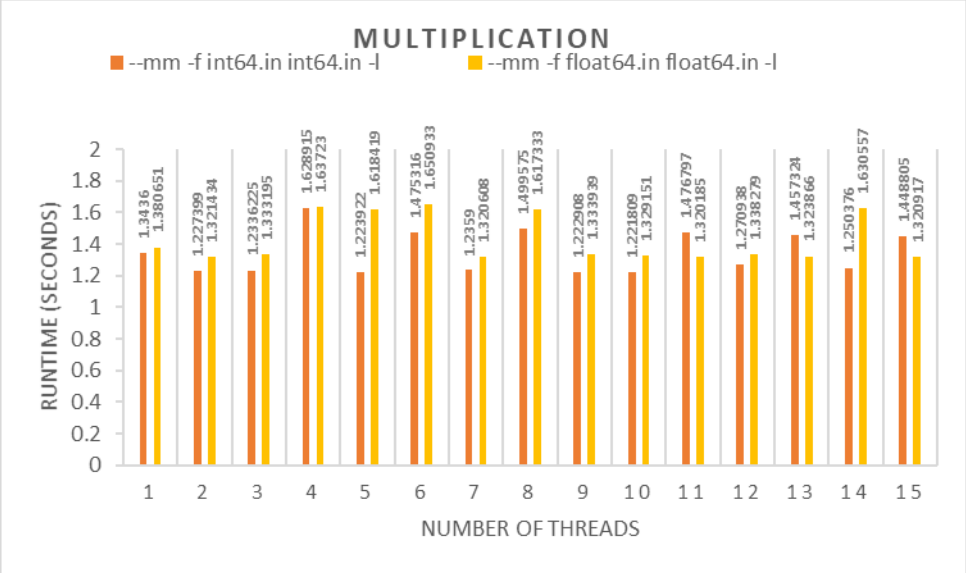
#pragma omp single This construct allows only one single thread to pass through this region at a time.

#pragma omp parallel for schedule(guided) reduction (+ : non-zero_values) This construct controls how the two for loops will operate depending on the shared variable above. The two iterators (i and j) in these two for loops are the row and column index of the matrix C (final/resultant matrix). If i is same with the row index of a non-zero value in CSC format and j is same with the column index of a non-zero value in CSR format they are multiplied together and if there is a match in the next iteration they are added together with the previous results which then finally become the non-zero value of the final matrix at Cij.

#pragma omp taskwait/critical This directive will wait for the threads to complete their task and make sure that no duplicate value is stored or printed to the output.

}

Calling another function to truncate and rounding the number for float values also might slow down the process as this can not be achieved with the standard format specifier. Printing to a file or console will also slow down the execution time.



Since the multiplication of matrix-matrix takes such a long time as the dimension increases the code were adjusted repeatedly but to speed up the performance. It difficult to achieve a faster speed as there are many factors involved in the process, such as computation overhead and idle threads. Matrix-matrix multiplication for int256.in*int256.in and float256.in*float256.in initially took about 5146 seconds (i.e. 1hour and 42minutes).

Observation of Performance

1. Everything works well on the sample provided except for multiplication of int1024.in*int1024.in and float1024.in*float1024.in as these two matrices contain 104249 and 105007 non-zeros values respectively.
2. Multiplication works better with the default core available on my system
3. Testing repeatedly within the same time(second) shows sudden increase in time.
4. It seems to appear that the fastest way to do matrix-matrix multiplication is using the OpenMP functions such as the for loop instead of tweaking around the code.
5. Using the reduction clause in calculating trace of a matrix significantly reduce the runtime.

Unit of time used in this project: ***second***

Calculation of time is done with the struct specified below:

struct timeval start, end; gettimeofday(&start, NULL); gettimeofday(&end, NULL);

float excecutionTime = ((end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1.e6);

Operating System

Windows edition

Windows 10 Home

© 2019 Microsoft Corporation. All rights reserved.

System

Manufacturer:	Acer
Model:	Aspire E5-523G
Processor:	AMD A9-9410 RADEON R5, 5 COMPUTE CORES 2C+3G 2.90 GHz
Installed memory (RAM):	8.00 GB (7.46 GB usable)
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display