ELEN4020 LAB 2 - 2D Matrix Transposition

School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050

Ntladi Mohajane-1599953, **Phetolo Malele**-1412331, **Zwavhudi Mulelu**-1110574

I. Introduction

This lab exercise illustrates the use of shared memory programming libraries, namely PThread and OpenMP. Three matrix transposition algorithms (naive, diagonal and block) are implemented on 2D matrices of various dimensions - 128, 1024, 2048 and 4096, using these shared memory programming libraries. This exercise is constrained to having only one square matrix and no second matrix should be created to store the transposed matrix, but a small temporary storage . This exercise is deemed successful if the matrix is optimally transposed by the three algorithms. It is assumed that matrix sizes and block sizes will all be of powers of two.

II. Naive OpenMP Matrix Transposition

The naive algorithm is a minor upgrade of the basic matrix transposition where rows are simply swapped with columns. The nested for-loops are parallelized using the OpenMP *#pragma* directive. The *#pragma omp parallel* is an OpenMP 'worksharing' construct which takes an amount of work(swapping the rows with columns) and distributes it over the available threads in the parallel region. The pseudo-code in Algorithm 1 shows the use of OpenMP threads in the naive algorithm.

Algorithm 1: `naive_openmp`

input: 2D matrix of size $N_o \times N_1$
input: mat_length, num_threads
`#pragma omp parallel`
for $i \leftarrow 0$ **to** mat_length **do**
  for $j \leftarrow i$ **to** mat_length **do**
    temp $\leftarrow mat[i,j]$ $mat[i,j] \leftarrow mat[j,i]$
    $mat[j,i] \leftarrow$ temp

During testing, it is observed that this parallelization method runs worse with an increase in threads. However, the resultant matrix is only guaranteed to be accurate if only one thread performs the computation. This is due to race conditions, thus resulting in multiple threads performing the same transposition twice.

III. Diagonal

This algorithm traverses through the matrix diagonal and swaps each column element to the right of the diagonal with its corresponding row element below the diagonal. It is similar to the naive implementation described above however, race conditions are avoided by explicitly assigning tasks to each thread.

A. PThread Diagonal Algorithm

A struct of type *data* is used to store the information each thread will need. It stores a pointer to the matrix, the number of threads assigned, the length of the matrix and the thread ID. Each thread is assigned a diagonal index by round robbin distribution to avoid race conditions and transposes along the row to the right of the assigned diagonal. In a addition, a simple counter with a mutex lock ensures each thread has a unique integer ID. Algorithm 2 shows the pseudo-code for this algorithm.

Algorithm 2: `diagonal_pthreads`

2D matrix pointer mat_length, num_threads
struct values [**mat,num_thread,mat_length,thread_id]
foreach *thread* **do**
  Mutex Lock
  thread_id++
  Mutex Unlock
  for $row \leftarrow thread\_id$ **to** $values.mat\_length$ **do**
    for $col \leftarrow row + 1$ **to** $values.mat\_length$ **do**
      temp$\leftarrow$values.$mat[row, col]$;
      values.$mat[row, col] \leftarrow$values.$mat[col, row]$;
      values.$mat[col, row] \leftarrow$temp;
foreach *thread* **do**
  Join thread and wait for other threads to exit

Although the diagonal implementation allows the threads to work concurrently with no race conditions, the work is not distributed evenly. This is because threads that work on diagonals which lie on the upper rows will do more work. Therefore, as the algorithm progresses, more threads will be idle. This causes the algorithm to behave in a more serial nature overtime.

B. OpenMP Diagonal Algorithm

Rather than manually distributing the diagonals to each thread, the OpenMP diagonal algorithm uses a *#pragma omp for* directive to automatically parallelize the outer for loop instead. Algorithm 3 shows the pseudo-code for this algorithm.

IV. Block

For this algorithm, The matrix is divided into blocks. Threads are then assigned to each block and the elements within each block are transposed. A barrier is then applied to synchronize the threads and the threads then transpose the blocks themselves.

Algorithm 3: `diagonalOpenMP`

input: 2D matrix pointer
input: mat_length, num_threads
`#pragma omp parallel`
`#pragma omp for`
for $row \leftarrow 0$ **to** mat_length **do**
  for $col \leftarrow row$ **to** mat_length **do**
    temp $\leftarrow mat[row, col]$;
    $mat[row, col] \leftarrow mat[col, row]$;
    $mat[col, row] \leftarrow$ temp;

A. *Pthread block algorithm*

Algorithm 4 below illustrates how the Block-Oriented-Threading algorithm is implemented. Threads are allocated similarly to the PThread diagonal implementation in that work is distributed in a round robbin fashion. The difference is that the *data* struct also holds the block length. Moreover, the functionality for transposing the elements within the blocks as well as the blocks themselves are encapsulated within separate functions.

Algorithm 4: `block_pthread`

input: 2D_matrix_pointer
input: mat_length, num_threads, block_length
struct data values[**mat,mat_length,num_thread,block_length,thread_id]
$values.mat \leftarrow 2D\_matrix\_pointer$
$values.mat\_length \leftarrow mat\_length$
$values.numth\_reads \leftarrow num\_threads$
$values.block\_length \leftarrow block\_length$
thread barrier initialised
foreach *thread* **do**
  Mutex Lock
  values.threas_id = global_thread_id
   global_thread_id++
  Mutex Unlock
  function to swap elements is called
   pthread_barrier_wait
  function to swap the blocks is called
foreach *thread* **do**
  Join thread and wait for other threads to exit

During testing it is observed that the performance of the block algorithm is not only dependent on the number of threads, but also the block size itself. Larger block sizes tend to increase performance especially for larger matrices. However, there comes a point where increasing the block size no longer has an effect and the optimum size is the smallest value where this is true.

Small block sizes imply that the threads are spending more time transposing the blocks themselves. This workload is not evenly distributed amongst the threads which hinders performance.

B. *OpenMP block algorithm*

The OpenMP block algorithm is a higher level PThread implementation similar to the diagonal algorithm. Moreover, it is observed to be faster than PThread block algorithm because the *#pragma* directives are more optimized. Algorithm 5 shows an illustration of the pseudo-code.

Algorithm 5: `block_openmp`

input: 2D_matrix_pointer
input: mat, mat_length, num_threads, block_length
$tile\_d \leftarrow block\_length$
$max\_tiles\_length \leftarrow mat\_lengthtile\_d)$
$max\_tiles \leftarrow max\_tiles\_length * max\_tiles\_length$
setting number of threads
`#pragma omp parallel` **foreach** *thread* **do**
  function to swap elements is called
  `#pragma omp barrier`
  function to swap blocks is called

V. Results

Table 1 shows that for smaller matrices (128) parallelism is of no benefit because few operations occur. However, for larger matrices (2048 and 4096), the block algorithm performs better, especially if an optimum block size is chosen, because the workload is more evenly distributed amongst the threads.

Table I
TABLE OF PERFORMANCE RESULTS

$N_o = N_1$	Serial	Pthreads		OpenMP		
		Diagonal	Block	Naive	Diagonal	Block
128	0.003	0.001	0.001	0.001	0.001	0.001
1024	0.038	0.015	0.014	0.024	0.014	0.013
2048	0.058	0.055	0.052	0.090	0.051	0.049
4096	0.232	0.211	0.198	0.342	0.202	0.188

VI. Conclusion

In conclusion the block algorithm is deemed most suitable for large matrix sizes especially if an optimal block size is chosen. The OpenMP library in particular performs better . Moreover, a brute force parallelism approach may result in an inaccurate result.

VII. References

[1] Stefan Amberger.(2018). A Parallel, In-Place, Rectangular Matrix Transpose Algorithm. JOHANNES KEPLER UNIVERSITY LINZ. [online] Available at: https://www3.risc.jku.at/publications/download/risc_5916/main%20v1.0.2.pdf