TRANSPOSING MATRICES BY POSIX THREAD AND OPENMP AND EVALUATING THE PERFORMANCE

Massi Mapani (783767) Thabiso Magwaza (836403)

School of Electrical & Information Engineering, University of the Witwatersrand, ELEN4020A Electrical Engineering Data Intensive Computing

Abstract: This report presents the group's collective understanding on parallelism and the implementation of it to solve Laboratory two. The Laboratory required students to transpose a matrix in-place using parallel programming. This was done with Open Multi-Processing libraries and POSIX thread libraries; which are Linux shared-memory libraries. The two were compared by their performance in time execution and it was found that Pthreads are low-level with better performance than OpenMP.

Key words: Multi-processing, Shared Memory, Pthreads, OpenMP

1. Introduction

Parallel Computing allows for several processes to be executed simultaneously without much processing overhead. Threads allow the main program to execute different tasks within the same main program [3]. They do this independently, with less overhead, and with shared memory addresses; hence mutual access to the same data that the main program is working on, which facilitates the simultaneous execution action.

There are several ways to implement parallel programming, in this report focus is placed on Open Multi-Processing (OpenMP) and Portable Operating System interface for Unix (POSIX or Pthread). The implementation of matrix transposition in both Pthreads and OpenMP is aimed at illustrating the performance, advantages and disadvantages of each shared memory method.

2. Background

2.1 Posix Thread (Pthread)

Pthreads make use of the bare minimum of a unix process [2], which provides the low overhead and are most useful when independent tasks in a program can execute at the same time and their functionality is interchangeable and may overlap in relevance[2]. Pthread programming involves the creation of threads to handle tasks, allowing threads to complete the tasks and managing and keeping track of the interaction between the threads.

Threads can create and pass on information to other threads. They do not have a specific sequence of action and the specifics of which thread works on what is at the discretion of the operating system. This is illustrated in the cited image [2] in Figure 1. Threads can be programmed to handle information based on priority and preference, however, assigning specific threads to act on certain information is not a controllable aspect of threads.

This point was brought out when implementing **Algorithm 2** in a queue configuration. A counter may be incremented in the main program after two sets of threads have been created. Hence any of the threads may work on the information. This introduces problems such as data corruption and race conditions; which were corrected by applying synchronization principle to the code such as barriers. This is illustrated in **Algorithm 1**

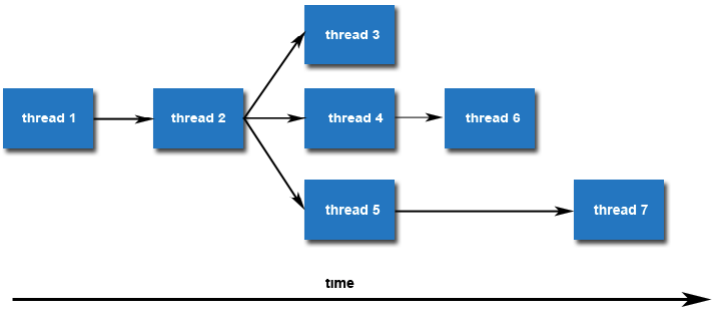

Figure 1 : Illustration of how threads work

Algorithm 1 Application of barrier in Pthread

For $i = 0 \leftarrow size$
  For $j = i \leftarrow size$
    thread[i]$\leftarrow pthread_create(transpose(matrix[j, i]))$
  End For
End For

2.2 OpenMP

OpenMP is an API that may be used for multi-threaded, shared memory parallelism. Although it is not guaranteed to make the best use of shared memory, it is a good way to transform serial code to parallel code [1]. The main purpose of openMP is to introduce standardization, ease of use and portability as it can run on both Unix/Lynx and windows. OpenMP programs begin as a single thread that executes until the first parallel construct is encountered. The master thread then creates a team of parallel threads which are then later joined into the main thread when they are finished executing.

3. Methodolgy

In the algorithm to transpose a square matrix in place, a number of swap operations are performed along the diagonals of the matrix. Algorithm 2 presents the algorithm used to transpose a square matrix of *size*. The work sharing *for* loop in openMP and pthreads are used to run this algorithm with a varying number of threads on matrices with different sizes.

Algorithm 2 In-place square matrix transpose

For $i = 0 \leftarrow size$
  For $j = i \leftarrow size$
    $swap(matrix[i, j], matrix[j, i])$
  End For
End For

In the code, the number of threads was increased from 4 to 128 with the time taken to execute the in-place transposition. Individual threads were tasked to transpose each row. However, this method does not result in each thread doing equal work, especially if the matrix dimension is not a perfect square. A better implementation of this problem would be to divide the matrix into four equal sections and then do the in-place transposition along the diagonal of each section with threads being assigned to a section to transpose.

4. Evaluation and Results

This section presents the performance tables for Algorithm 2 implemented using both Pthreads and OpenMP in Table 1-5. The sizes of the square matrices are varied from $N = 128$ to $N = 8192$. From the time performance it is evident that for Pthreads, as the number of threads increase, the performance of the transposition improves. While for OpenMP as the size of the matrix increases the performance slows down.

5. Conclusion

It is evident to see that Pthreads are a logical choice if the amount of tasks and data that a process needs to compute are large and of low-level as compared to OpenMP from the experimental data in Table 1-5. OpenMP has a higher processor overhead and thus takes more process time, however it is easier to implement the threading process with OpenMP, and though limited to C programming, is better for work-sharing projects.

REFERENCES

[1] Blaise Barney. Openmp programming, 2017.
[2] Blaise Barney. Posix threads programming, 2017.
[3] G. Wellein . G. Hager. *Introduction to High Performance*. CRC Press Taylor and Francis Group, 2001.

Table 1 : Performance table for 4 threads

	N = 128	N = 1024	N = 8192
Pthread	0.000061s	0.000056s	0.000082s
OpenMP	0.003415s	0.061691s	4.866210s

Table 2 : Performance table for 8 threads

	N = 128	N = 1024	N = 8192
Pthread	0.000059s	0.000069s	0.000081s
OpenMP	0.003898s	0.061553s	4.076873s

Table 3 : Performance table for 16 threads

	N = 128	N = 1024	N = 8192
Pthread	0.000043s	0.000045s	0.000053s
OpenMP	0.007929s	0.065755s	3.636055s

Table 4 : Performance table for 64 threads

	N = 128	N = 1024	N = 8192
Pthread	0.000043s	0.000050s	0.000051s
OpenMP	0.031432s	0.072457s	3.636055s

Table 5 : Performance table for 128 threads

	N = 128	N = 1024	N = 8192
Pthread	0.000043s	0.000046s	0.000052s
OpenMP	0.062685s	0.089909s	4.216756s