Laboratory Exercise 2

Shane House (749524), Benjamin Rosen (858324)

ELEN4020, 16 March 2018

I. Algorithm Description

$$\begin{pmatrix} d & x & x & x & x & x \\ \mathbf{1} & d & x & x & x & x \\ \mathbf{2} & \mathbf{3} & d & x & x & x \\ \mathbf{4} & \mathbf{5} & \mathbf{6} & d & x & x \\ \mathbf{7} & \mathbf{8} & \mathbf{9} & \mathbf{10} & d & x \\ \mathbf{11} & \mathbf{12} & \mathbf{13} & \mathbf{14} & \mathbf{15} & d \end{pmatrix}$$

$$1;\ 3;\ 6;\ 10;\ 15;\ ... \tag{1}$$

$$T_n = 0.5n^2 + 0.5n \tag{2}$$

$$n = 0.5 + \sqrt{0.25 + 2T_n} \tag{3}$$

$$x = floor(n) \tag{4}$$

$$y = trunc((n - x) * (x + 1)) \tag{5}$$

A. Mathematical Explanation

An NxN matrix, with N set to 6, is shown above. d represents the diagonal values that do not need to be swapped. The numbers in bold indicate the swap iteration, while the x's are the values to be swapped with correspondingly.

The goal of this algorithm is to evenly divide the number of swaps among the available threads. To this end, a method needed to be developed that mapped the swap iterations to an *(x, y)* position in the matrix. Looking at the matrix above, the last swap iteration value in every row forms a quadratic number pattern, shown in Equation 1. The equation that relates the value of the pattern (T_n) to the iteration n is given by Equation 2. This is a quadratic equation which can be solved using the quadratic formula. Since the iteration value cannot be negative, only one of the values is used. The simplification of the quadratic formula is depicted in Equation 3. From this value, x and y values can be obtained using Equations 4 and 5. For values in between the ones given in the number series, Equation 3 will give a number in between the iteration value. For example, swap iteration 5 will give an n value of 3.7, which indicates it is between the 3rd and 4th iterations. Therefore, we know it is in the 3rd row of the matrix. It was found that the value of n increases for each iteration depending on the number of values in that row. Thus, the y value is determined by taking the decimal place, of 0.7 in the above example, and multiplying it by the number of values in that row (3 for our case). Taking only the integer value gives us the column, 2 for the example.

B. *Program Implementation*

1) Pthread: A main thread delegates the swaps to child threads by first determining the number of swaps each thread will perform. It then iterates through the start positions (increasing by the number of swaps) and determines the child thread's start position (using the algorithm outlined in Section I-A). This value, along with a reference to the matrix and the number of swaps to perform, is passed in a struct to the thread. Each thread performs the swaps by cycling through the swap iterations (as in the matrix above) and then joins the main thread. The main thread, after delegating the work, performs the last batch of swaps.

2) OpenMP: A parallel region is defined, in which the threads, including the main thread, perform the swaps as in the above section. However, each thread determines its own start position, by making use of its thread ID.

3) Timing: For accurate timing of parallel threads, *clock_gettime()* is used, which uses the actual time (instead of CPU cycles). This delivered more consistent timing when running the program multiple times.

II. Results

Table I summarises the times for the different dimensions, threads and implementations used. The general trend was that for small dimensions, the timings varied significantly but the smaller number of threads tended to have the better times. For large N, having more threads caused the timings to improve. OpenMP, except for some outliers, completed the operations quicker than Pthread.

Of course, only a certain number of threads could be executed in parallel on our machines, meaning that the time benefit of having more threads was not significant.

Table I
Results for Pthread and OpenMP Programs for Varying Matrix Dimensions and Threads

Implementation	No. of Threads	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
OpenMP	4	0.0213945	0.0146729	0.5200562
	8	0.0004815	0.0143658	0.5099768
	16	0.0011964	0.0074405	0.5125988
	32	0.0018648	0.0100943	0.5074549
	64	0.0037445	0.0084525	0.5076541
	128	0.0179112	0.0137622	0.5088798
Pthread	4	0.0006303	0.0078572	0.7830937
	8	0.0008100	0.0099369	0.8439945
	16	0.0017638	0.0110775	0.8351318
	32	0.0034362	0.0115999	0.8432556
	64	0.0072569	0.0142217	0.8368004
	128	0.0139730	0.0193327	0.8371888