Home assignment 2

Emanuel Paraschiv & Mohammad Asfour
Group 45

February 2025

1 Introduction

The scope of this assignment is to present our implementation of different programs making use of Open MP library for parallel execution of code.

2 Matrix

The program should be responsible for computing the sum of matrix elements along with finding a minimum and a maximum with their corresponding position. In order to attain the last 2 features, when a candidate is found, it has to be updated atomically, by only one thread at a time, reason for which the pragma *critical* is used.

```
//example of finding minimum
if(matrix[i][j]<min)
    #pragma omp critical
    min_row=i;
    min_col=j;
    min=matrix[i][j];
```

The execution time on different number of processors (threads) with a varying input size is presented in the table below:

size / threads	1	2	3	4	5
1000	0.0013	0.0010	0.0009	0.0007	0.0009
10.000	0.148	0.088	0.066	0.079	0.088
1.000.000	0.152	0.088	0.072	0.067	0.055

Table 1: Execution time overview

As it can be seen, the speedup is consistent, but for small matrices the overhead becomes considerable after about 4-5 threads and the speedup decreases, simulating a gauss curve with the peak at 4 threads. However, for a bigger matrix the result is better displayed and we can see an increase in performance for even more threads.

nb of threads/ size of matrix	**1000**	**10.000**	**1.000.000**
2	1.3	1.68	1.72
3	1.44	2.24	2.11
4	1.85	1.87	2.26
5	1.44	1.68	2.76

Table 2: Speedup of the program for different number of processors and matrices

3 Quicksort

Quicksort works by recursively splitting an array of unordered elements around a pivot until it becomes sorted.

A problem however arises when the given array becomes too small and the benefits of parallelization are not observable. In order to cope with that inconveninece, one solution that we were able to come up with was to set a lover threshold, a baseline at which, given a small set of numbers, the program starts executing in a sequential manner:

```
if ((max - min) < THRESHOLD) {
        quicksort(arr, min, pivot - 1);
        quicksort(arr, pivot + 1, max);
```

Otherwise, given a reasonable input Open MP library is used in recursive calls on the shared given array. There however comes another problem: the calls are not synchronized, and in order to avoid anomalies, we need to wait until both are complete using *taskwait*.

```
#pragma omp task shared(arr)
quicksort(arr, min, pivot - 1);

#pragma omp task shared(arr)
quicksort(arr, pivot + 1, max);

#pragma omp taskwait
```

The execution time on different number of processors (threads) with a varying input size is presented in the table below:

size / threads	1	2	3	4	5
100.000	0.0086	0.0052	0.0049	0.0041	0.0035
1.000.000	0.156	0.092	0.077	0.065	0.054
100.000.000	0.162	0.086	0.079	0.067	0.057

Table 3: Execution time overview

nb of threads/ size of matrix	**100.000**	**1.000.000**	**100.000.000**
2	1.65	1.69	1.88
3	1.75	2.02	2.05
4	2.09	2.4	2.41
5	2.45	2.88	2.84

Table 4: Speedup of the program for different number of processors and arrays

As we can see again, a bigger input size of array yields a better speedup by allowing the threads to work at their full potential.

4 Palindromes

For this task we are required to find palindromes and semordnilaps given a dictionary file.

In the main function, the dictionary is read and all words are stored in a 'words[]' array. When the threads are joined, the caught palindromes and semordnilaps will be written to an output file. These steps are done sequentially.

For the concurrent part, 2 helper functions (return boolean) are created `isPalindrome` and `isSemor`, that ensure that a given word is either a palindrome (by reversing it and comparing to the original) or a semordnilap (by reversing it and checking its existence in the dictionary). A binary seach algorithm was also used (since the file was ordered) to find the reversed word in the file for Semordnilaps.

In the `Palindrome` function, a for loop, which is run with the #`pragma omp parallel for shared(words)` directive, iterates over the words[] array and uses the helper functions to catch all palindromes and semordnilaps. Incrementation of the shared variable are done with the #`pragma omp atomic` to ensure mututal exclusion.

Performance evaluation

To evaluate the performance, we measure the speedup for different number of processors on different sizes of dictionary sizes. The results are shown in the following table:

nb of threads/ size of dictionary	5000	15000	25000
2	1.2	1.25	2
4	1.1	1.4	2.6
6	1.2	1.6	4
10	1.2	2	4

Table 5: Speedup of the program for different number of processors and dictionaries

For small dictionary sizes, we can see that we don´t get a lot of speedup because the `omp_get_wtime()` function cannot record smaller values of time, or because the dictionary size is too small. But for larger sets of data, we can see that multi-threading actually pays off as we can get up to 4 times speedup when we use the full dictionary.