Assignment 2 Report - Programming with OpenMP

Dean Tsankov, Kacper Lisik

February 9, 2025

Introduction

In this assignment, we learn how to write, build and execute a parallel program in OpenMP to show performance speedup on more than one processor as well as to find dependencies in loops and to write code such that it can correctly be executed in parallel.

Problem 1 - Compute the Sum, Min, and Max of Matrix Elements

To begin with we are given the matrixSum-openmp.c program which we have to manipulate in order to complete the given tasks as an introduction to OpenMP directives.

Task description

```
The purpose of this problem is to introduce us to basic OpenMP usage. The
↪ program matrixSum-openmp.c computes a sum of matrix elements in
↪ parallel using OpenMP. Develop and evaluate the following modified
↪ version of the program.
Extend the program so that in addition to the sum, it finds and prints a
↪ value and a position (indexes) of the maximum element of the matrix
↪ and a value and a position of the minimum element of the matrix. To
↪ check your solution, initialize elements of the matrix to random
↪ values. Use OpenMP constructs. Run the program on different numbers
↪ of processors and report the speedup (sequential execution time
↪ divided by parallel execution time) for different numbers of
↪ processors (up to at least 4) and different sizes of matrices (at
↪ least 3 different sizes). Run each program several (at least 5) times
↪ and use the median value for execution time. Try to provide
↪ reasonable explanations for your results. Measure only the parallel
↪ part of your program.
```

Key points of solution

OpenMp makes it very simple to turn a sequential implementation of a program into a concurrent one. The traversal loop over the matrix looks like so:

```
    (...)
    start_time = omp_get_wtime();
    #pragma omp parallel for reduction(+ : total) private(j) shared(min,max)
        for (i = 0; i < size; i++){
            for (j = 0; j < size; j++)
            {
                total += matrix[i][j];
                if (matrix[i][j] < min.value)
                {
                    min.value = matrix[i][j];
                    min.xCoordinate = j;
                    min.yCoordinate = i;
                }
                else if (matrix[i][j] > max.value)
                {
                    max.value = matrix[i][j];
                    max.xCoordinate = j;
                    max.yCoordinate = i;
                }
            }
```

```
        }
        // implicit barrier

        end_time = omp_get_wtime();
    (...)
```

Performance exploration

Example sequential time as a median from 100 executions for an 8x8 matrix:

$$8.73899 * 10^{-8} s$$

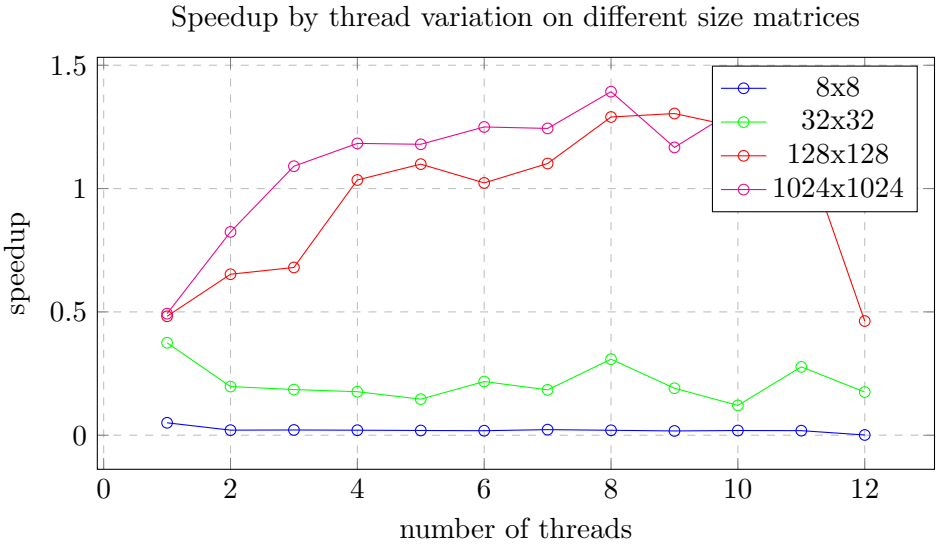

Figure 1: Tree add time complexity

Generally we would expect the speedup to keep increasing not decrease. Still these results could make sense considering that parallelizing helps for big workloads, or when the workers are used many times over for many moderate sized tasks (so the cost of launching the threads is a fraction of the work they do). But in this case performing an objectively small amount of additions and comparisons it is negligible to the CPU. And so they are just not doing enough to benefit from parallelizing this specific task.

Problem 2 - Quicksort

Task description

```
The quicksort algorithm sorts the list of numbers by first dividing the
↪ list into two sublists so that all the numbers if one sublist are
↪ smaller than all the numbers in the other sublist. This is done by
↪ selecting one number (called a pivot) against which all other numbers
↪ are compared: the numbers which are less than the pivot are placed in
↪ one sublist, and the numbers which more than the pivot are placed in
↪ another sublist. The pivot can be either placed in one sublist or
↪ withheld and placed in its final position. Develop a parallel
↪ multithreaded program (in C/C++ using OpenMP tasks) with recursive
↪ parallelism that implements the quicksort algorithm for sorting an
↪ array of n values. Perform speedup tests on the parallel part of the
↪ code and give explinations.
```

Key points of solution

Following the standard procedure of developing a quicksort algorithm we just have to introduce the proper opemmp directives in crucial spots in order to make the recursive calls of the algorithm execute on separate threads and as such parallelize the execution. The entire code can be found in the source code directory, here is the main openmp section

```
    (...)
    Array quickSort(Array *mainSet)
    {
        Array left;
        Array right;
        for (k = 0; k < currSize; k++)
        {
            if (mainSet->array[k] < pivot)
            {
                insertArray(&left, mainSet->array[k]);
            }
            else if (mainSet->array[k] > pivot)
            {
                insertArray(&right, mainSet->array[k]);
            }
            (...)
        }

        #pragma omp task shared(left)
```

```
        {
            left = quickSort(&left);
        }

        #pragma omp task shared(right)
        {
            right = quickSort(&right);
        }

        #pragma omp taskwait
        (...)
    }
    (...)
```

We have to use `omp taskwait` in order to ensure that all previously generated calls were executed before we can continue on joining the left and right parts of the set.

Another important note is that at the first `quicksort()` function call we need to initialize a parallel space as:

```
    (...)
    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        {
            sorted = quickSort(&toSortSet);
        }
    }
    end_time = omp_get_wtime();
    (...)
```

We then run the initial call as single so as to let the other threads be ready to begin working on the tasks spawned by the original single thread.

Performance exploration

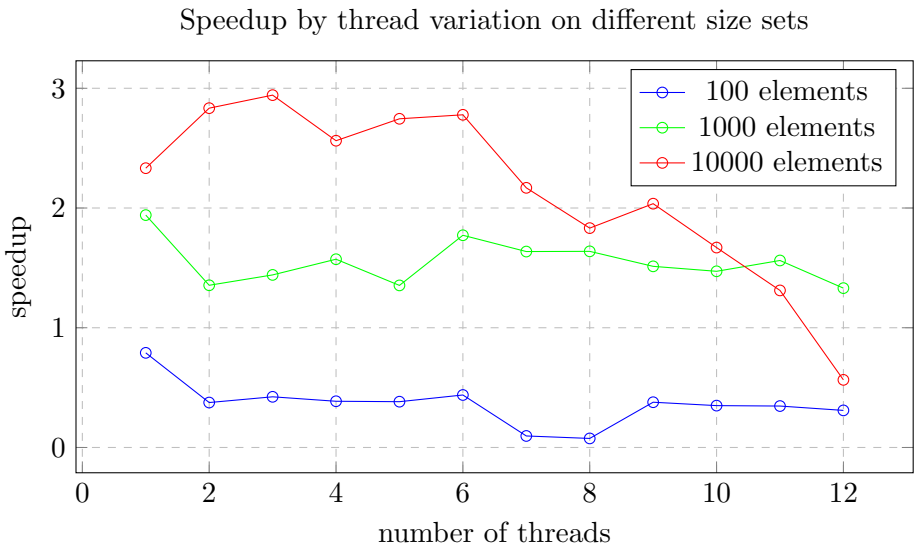

Figure 2: Tree add time complexity

Here it is unclear to us why the openmp single thread execution still has a noticeable speedup, but we could probably assume that this is just a standard offset due to the variation in the generated data. Otherwise the trends seem to follow the notion that there is a threshold at which creating more threads is more computationally intensive then the positive work done by them in total. Not to mention the fact that the computer on which these tests were ran has 6 physical cores and 12 logical processors so the downward trend after 6 threads could also have to do with the hardware implementation.

Problem 6 - Find Palindromes and Semordnilaps

```
There is an online dictionary in /usr/dict/words or /usr/share/dict/words
↪  under Linux. The file is used, in particular, by the Linux spell
↪  command and contains a few thousand words, e.g., 235886 words. You
↪  can download a smaller file called words containing 25143 words.
Recall that palindromes are words or phrases that read the same forward
↪  and backward, such as "noon" or "radar," i.e., if you reverse all the
↪  letters, you get the same word or phrase. Reverse pairs or
↪  semordnilaps are words whose reverse forms are different valid words,
↪  such as "draw" and "ward". Your task is to find all palindromes and
↪  all  semordnilaps in the dictionary. For example, "noon" is a
↪  palindrome; hence, its reverse is trivially in the dictionary. A word
↪  like "draw" is semordnilap and "ward" is also in the dictionary.
Your parallel program should use W worker processes, where W is a
↪  command-line argument. Use the workers just for the compute phase
↪  (i.e., to find the palindromic words); do the input and output phases
↪  sequentially. Each worker should count the number of words
↪  (palindromes and semordnilaps) that it finds. Your program should
↪  write the palindromes and semordnilaps to a results file and output
↪  the total number of palindromes and semordnilaps in the dictionary
↪  and the number of palindromes and semordnilaps found by each worker.
```

Key points of solution

After sequentially reading the words file, we ensure that the words are sorted in alphabetical order. They appear to be sorted, but to be certain, we use the `std::sort()` command, which is computationally inexpensive if the words are already sorted or nearly sorted. We distribute the work and create N threads, with each thread checking whether any of num_of_words/num_of_threads words is a palindrome or a semordnilap:

```
(...)
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        thrd_data[thread_id].start = (words.size()/workers) * thread_id;
        thrd_data[thread_id].end = (thread_id == workers - 1)?
            words.size()-1 : (words.size()/workers) * (thread_id+1);
        thrd_data[thread_id].tid = thread_id;
        thread_fun(&thrd_data[thread_id]);
    }
(...)
```

Thread function:

```
(...)
void* thread_fun (thread_data* data){
    data->palindromes = 0;
    data->semordnilaps = 0;
    for (int i = data->start; i < data->end; i++){
        //calculate the reverse
        string reverse = words[i];
        for (int j = 0; j < reverse.size() / 2; j++)
            swap(reverse[j], reverse[reverse.size() - 1 - j]);

        //check if palindrome
        if(compare_strings(reverse, words[i]) == 0){
            data->palindromes_vector.pb(reverse);
            data->palindromes++;
            continue;
        }

        //look for semordnilaps - fast
        int result;
        if((result = binsearch(i+1,words.size()-1, reverse)) != -1){
            data->semordnilaps_vector.pb(words[result]);
            data->semordnilaps_vector.pb(words[i]);
            data->semordnilaps+=2;
        }
    }
    return NULL;
}
(...)
```

Each thread goes through its assigned list of words, computes the reverse of every word, and checks if the word is a palindrome. Then, using binary search, it looks through the word list to see if the reverse of the given word is also in the word list.

Parallel Performance Analysis

Experiment Setup

To analyze the speedup of the palindrome detection program, we ran the experiment on three different dictionary sizes:

- **Small Dictionary** (2,514 words)
- **Medium Dictionary** (8,381 words)
- **Large Dictionary** (25,143 words)

For each dictionary size, we ran the program with **1, 2, 4, and 8 threads**. Each configuration was executed **50 times**, and the median execution time was used to calculate the speedup:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

Results and Discussion

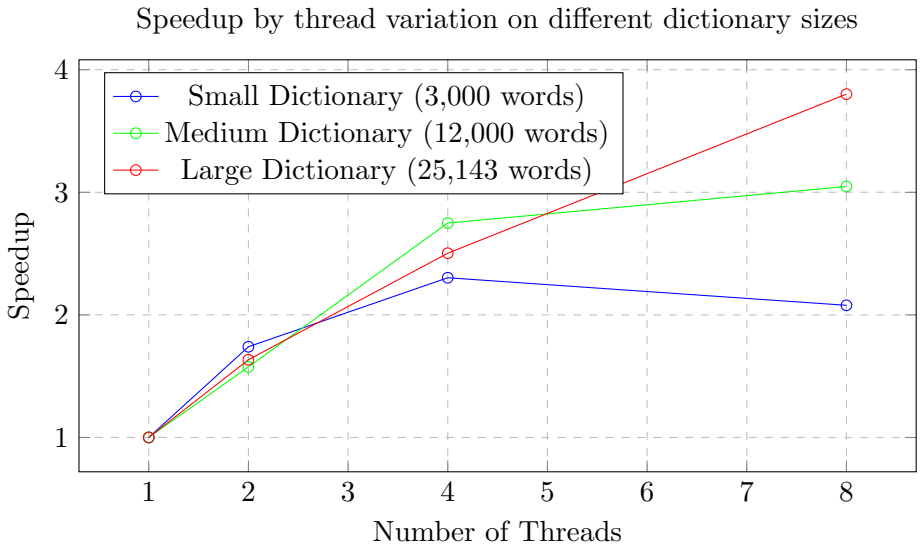

Figure 3: Speedup for different dictionary sizes

Analysis of Results

The speedup increases as we add more threads, but the improvement is not perfectly linear due to **parallel overhead** and **synchronization costs**.

- **For 2 threads**: Speedup is between 1.5x–1.7x, showing a good improvement.
- **For 4 threads**: Speedup increases to 2.3x–2.7x, demonstrating good parallel efficiency.
- **For 8 threads**: Speedup **flattens for the small and medium dictionaries**, meaning adding more threads does not improve performance much. This is likely because:
  - The extra threads **spend more time managing work** than actually processing words.
  - **Memory access slows down** when too many threads try to read and write at the same time.
  - The workload is **too small to be efficiently split** among all threads.

Conclusion

- **Smaller dictionaries** show **less speedup** because parallel overhead is relatively large compared to the computational cost.
- **Larger dictionaries** benefit more from parallelism, but the gains diminish as the number of threads increases.