SPM Assignment 1

Francesco Benocci

April 2024

1 Introduction

The goal of the assignment is to parallelize an algorithm that simulate an upper-triangular wavefront computation pattern on a NxN matrix. Each element of an upper diagonal (starting from the leading diagonal) can be computed independently. Instead, distinct upper diagonals have to be computed serially in order (first, all elements of the diagonal k, and once the computation has completed, all elements of the diagonal k+1).

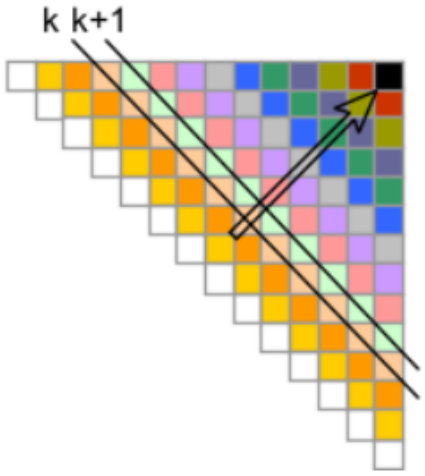

Figure 1: Graphical example of the computation.

To address this issue, I've implemented two versions of the code using C++ Threads: the initial one with a static assignment of the rows to the threads, while the alternative version utilizes a dynamic assignment using atomic variables. The solution is composed by a single file named UTW.cpp, and I've wrote a Makefile enabling compilation with simple 'make' command without additional dependencies.

Additionally, the Makefile allows running the code in two different ways:

- With a single execution using the command "make run_single".
- With 10 executions, averaging the time obtained from each, using the command "make run_multiple".

You can specify arguments for both types of executions using the format ARGS="N min max num_thread". If not specified, the default values are N=6, min=0, max=1000, and the number of threads=6.

2 Implementation

2.0.1 Static parallelization

The static version is based on a cyclic distribution. Each threads is created with a unique id (increasing integer) that identify itself. A barrier named "my_barrier" is necessary to ensure that all threads are synchronize and stop at the end of a diagonal. The initialization of the thread is the following:

```
void wavefront_element_cyclic(
    const std::vector<int> &M,
    const uint64_t &N,
    const uint64_t &num_threads) {

    std::barrier my_barrier(num_threads);

    std::vector<std::thread> threads;
    threads.reserve(num_threads);

    for (uint64_t i = 0; i < num_threads; i++){
        threads.emplace_back(static_parallelization, i);
    }

    for (auto& thread : threads)
        thread.join();
}
```

Each thread compute the following lambda function that takes only the thread id as argument:

```
auto static_parallelization = [&] (const uint64_t& id) -> void {
    const uint64_t off = id;
    const uint64_t str = num_threads;

    // for each upper diagonal
    for(uint64_t k = 0; k<N; ++k) {
        // for each elem. assigned to the thread of the diagonal
        for(u_int64_t i = off; i<N-k; i+=str){
```

```
            work(std::chrono::microseconds(M[i*N+(i+k)]));
        }

        // if the number of element of the diagonal are less then the number of threads
        if(id+1 > N-k){
            my_barrier.arrive_and_drop(); //drop the thread
            break;
        }
        else{
            my_barrier.arrive_and_wait();
        }
    }

};
```

To execute all the work along the diagonal, each thread is assigned the first element (the first row) based on its id. Then, it's incremented by the number of threads so that the work is evenly distributed among the threads. Considering an example with N=8 and 3 threads, we can illustrate the distribution of elements along the first diagonal as follows:

- Thread 0 will execute elements 0, 3, and 6.
- Thread 1 will execute elements 1, 4, and 7.
- Thread 2 will execute elements 2 and 5.

This distribution ensures that each thread is responsible for processing a portion of the diagonal, contributing to parallel execution and efficient utilization of available resources. Furthermore, at each iteration along the diagonal, it's checked whether the number of elements in the current diagonal is less than the total number of threads. If the condition is met, the thread is dropped to prevent unnecessary idle time and optimize resource usage.

An alternative to this type of execution is the use of a block distribution. However, after a simple preliminary analysis, it appears to be worse than the cyclic alternative on individual elements.

2.1 Dynamic parallelization

In the dynamic version it's necessary to introduce an atomic variable called 'element', which indicates the current element that threads must process, an integer 'k' representing the current diagonal on which threads operate, and a lambda function that is called upon reaching the barrier. This lambda function increments 'k' and resets 'element' to 0, ensuring proper progression along the diagonal and effective synchronization among threads.

```
void wavefront_dynamic(
    const std::vector<int> &M,
```

```
    const uint64_t &N,
    const uint64_t &num_threads){

    volatile std::atomic<uint64_t> element {0};
    uint64_t k = 0; // current diagonal

    // lambda function called by the barrier when all the threads reach it
    auto on_completition = [&]() {
        k++; // increment the current diagonal
        element.store(0);
    };
    std::barrier my_barrier(num_threads, on_completition);

    std::vector<std::thread> threads;
    threads.reserve(num_threads);
    std::atomic<uint64_t> atomic_row {0};

    for (uint64_t id = 0; id < num_threads; id++)
        threads.emplace_back(dynamic_paralizzation, id);

    for (auto& thread : threads)
        thread.join();

}
```

So the threads execute the work on the element that indicated by the atomic variable and during that they increment that. The result is that each thread when is complete the work check the current element to execute and if the current diagonal is completed they arrive to the barrier.

Each thread iterates over the upper diagonal, reading and incrementing an atomic variable for its current element. If there's remaining work, the thread performs its task. Otherwise, it checks if it's necessary to drop the thread due to the imbalance between elements and threads. Finally, threads synchronize at a barrier before proceeding.

```
auto dynamic_paralizzation = [&] (const uint64_t& id) -> void {
    uint64_t e;

    while(k<N) { // for each upper diagonal

        // take and increment the current element
        e = element.fetch_add(1);

        if(e < N-k){ // if there is still work to do
            work(std::chrono::microseconds(M[e * N + (e + k)]));
        }
```

```
        else{
            // if the number of element of the diagonal are less
            // then the number of threads i drop the thread
            if(id+1 > N-k){
                my_barrier.arrive_and_drop();
                break;
            }
            else{
                my_barrier.arrive_and_wait();
            }
        }
    }

};
```

3 Performance evaluation and conclusions

All the results in this section are an average of 10 different execution with specific arguments (N, min, max and number of threads). We can see an analysis of the performance calculated by running the code on the remote machine provided for the course (spmcluster), which has 40 cores. This is the table of results, represented in seconds:

Thread	Static	Dynamic
1	65,584	65,583
2	33,613	32,868
4	17,552	16,547
8	9,3298	8,4029
16	5,09843	4,3446
32	2,9225	2,3293
40	2,4666	1,9288

Figure 2: Results for N=512 matrix and min=0 max=1000

The metric used is relative speedup and is computed as $\frac{T_{seq}}{T_{par}}$. The following figure (3) show the strong scalability, namely the speedup as the number of threads increases while keeping the matrix size constant. Specifically, in the case of an unbalanced workload distribution (i.e., min=0 and max=1000), there is a clear distinction in the increase of threads between static and dynamic approaches, with the latter being significantly higher. Conversely, in the case of a balanced workload (min=500 and max=500), the two approaches are practically identical.

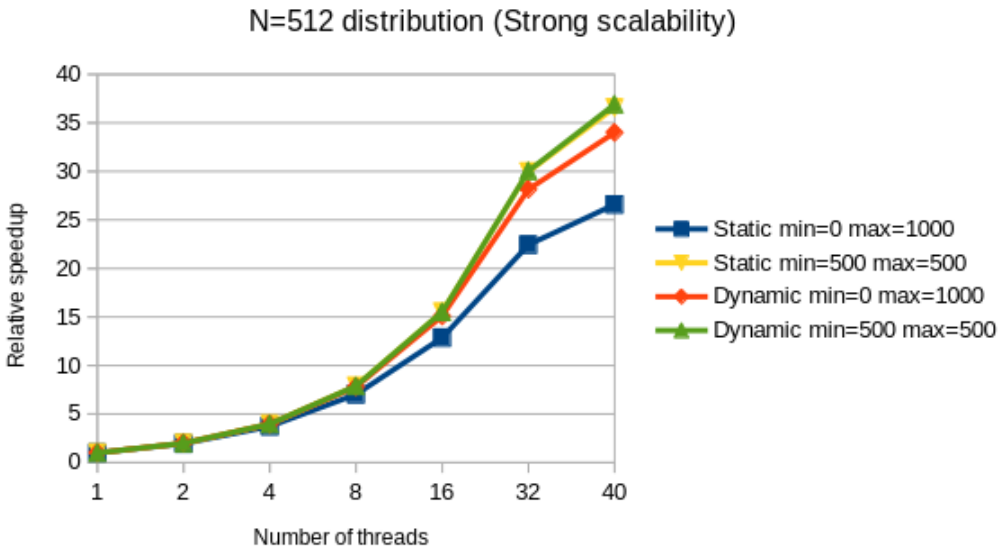

Figure 3: Strong scalability.

In the following figure (4) we analyze weak scalability, which examines the performance as the number of threads increases with respect to the size of the matrix (N). As previously observed, in the case of a balanced workload distribution, the static and dynamic approaches behave practically the same way, while substantial differences are noticeable in the case of an unbalanced distribution, where the dynamic approach consistently outperforms.

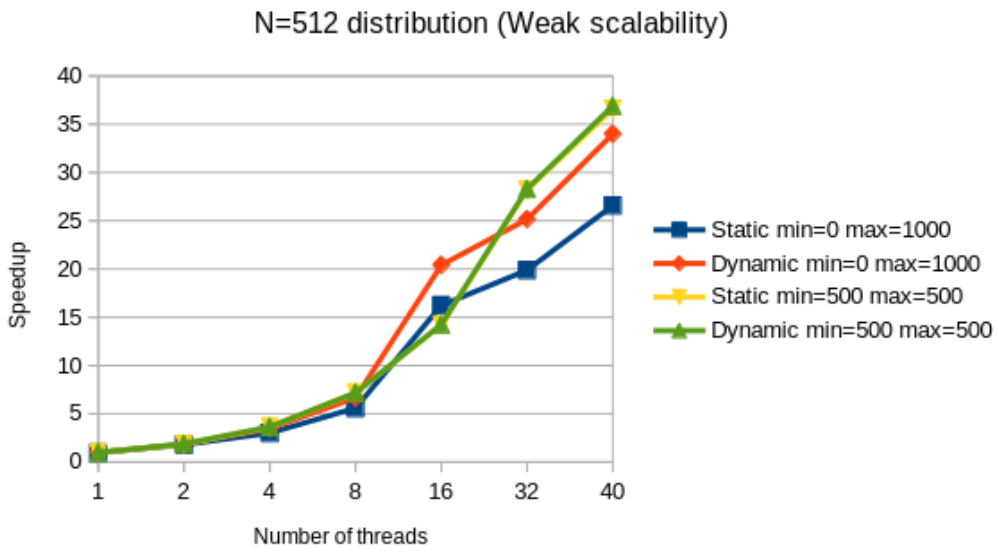

Figure 4: Weak scalability.

Also, the results obtained were compared with an implementation using the OpenMP library and for both static and dynamic approach we have a strong correspondence. In conclusion, the analysis revealed distinct behaviors between the two approaches, particularly evident in scenarios with unbalanced workload distributions. The study underscores the effectiveness of dynamic workload distribution in maximizing parallel execution efficiency, especially in scenarios with heterogeneous computational requirements.