SPM Assignment - 1

Simone Passèra

April 09, 2024

1 Introduction

This assignment was intended to parallelize an algorithm that exhibits an upper-triangular wavefront computation pattern on an 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 $k + 1$).

I implemented two parallel versions of the UTWavefront.cpp code using C++ Threads, the first using a static approach with block-cyclic distribution and the second through a dynamic assignment strategy. The implementation of the two versions and the tests performed on the clusters provided by the University of Pisa are presented in more detail below.

2 Code Implementation

2.1 Static distribution

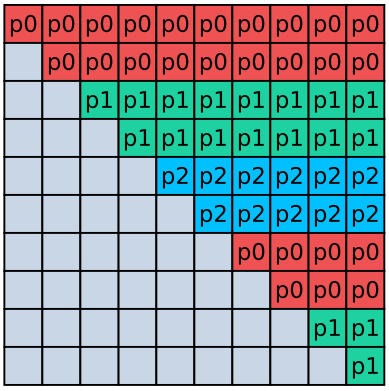


Figure 1: Example of block cyclic distribution, 10x10 matrix, three threads. The chunk size is equivalent to 2 rows.

The static version is implemented using cyclic block distribution. Each thread is assigned a set of matrix rows equal to the chunk size. Figure 1, shows an example of how a 10x10 matrix is divided by rows with a thread count of three. The code takes into account the maximum number of threads that can run on each diagonal and therefore does not allow more threads to be created even if specified by the user via command line. In this particular example, Figure 1, the leading diagonal can be parallelized by a maximum of 5 threads ($N/chunk\_size$ (+1 if there is a remainder)).

```
1  void wavefront_parallel_static(const std::vector<int> &M, uint64_t N, uint64_t num_threads,
2                                 uint64_t chunk_size) {
3    num_threads = std::min(num_threads, N/chunk_size + (N%chunk_size ? 1 : 0));
4    std::barrier sync_point(num_threads);
5
6    auto block_cyclic = [&] (const uint64_t id) -> void {
7      // precompute offset, stride, and number of diagonals
8      const uint64_t off = id * chunk_size;
9      const uint64_t str = num_threads * chunk_size;
10     const uint64_t num_diagonals = N-off;
11
12     // for each upper diagonal
13     for(uint64_t k = 0; k < num_diagonals; k++) {
14       // for each block of size chunk_size in cyclic order
15       for (uint64_t lower = off; lower < (N-k); lower += str) {
16         // compute the upper border of the block (exclusive)
17         const uint64_t upper = std::min(lower+chunk_size, N-k);
18         // compute task
19         for (u_int64_t i = lower; i < upper; i++)
20           work(std::chrono::microseconds(M[i*N + (i+k)]));
21       }
22
23       // barrier
24       if (k == (num_diagonals-1))
25         sync_point.arrive_and_drop();
26       else
27         sync_point.arrive_and_wait();
28     }
29   };
30
31   .....
32 }
```

Listing 1: Wavefront parallel static block-cyclic version.

The code in Listing 1, shows the `block_cyclic` lambda function executed by each thread. Each of them, line 7, calculate the right indices and which diagonals they will have to perform. At line 13, each thread executes its tasks for the current diagonal, it will then move on to the next diagonal after synchronizing with the other threads through the barrier at line 27. If the thread will not have to compute other tasks on the next diagonals then it removes itself from the barrier with the *arrive_and_drop()* function, line 25, and terminates.

2.2 Dynamic distribution

The dynamic version is a generalization of the static version. In this case, each thread is not aware of which tasks it will need to compute. During the execution of a diagonal, a generic thread, in a loop, takes a set of tasks on the diagonal specified by the chunk size, executes them, and tries to process other tasks until the diagonal is completed. The synchronization between diagonals occurs with the barrier. Again, the code takes into account the maximum number of threads that can run on each diagonal and therefore does not allow more threads to be created even if specified by the user via command line. As with the static version the active threads in each diagonal never exceed the maximum value ($(N - current\_diagonal)/chunk\_size$ (+1 if there is a remainder)).

The code in Listing 2, shows the lambda function (`task`), executed by each thread. Each of them, atomically fetch the rows that need to be calculated, line 25, and compute the tasks, line 31. If the diagonal has been completed each thread stops on the barrier, line 48, after checking if the number of active threads exceeds the maximum number allowed for that diagonal, and in that case tries to terminate to have the correct active number of threads, line 36. Each thread shares a variable to keep track of the diagonal that currently needs to be computed (`global_diagonal`). The variable will then be incremented by the function passed to the barrier (`on_completion`), executed before unlocking all the threads stopped on the barrier.

```cpp
void wavefront_parallel_dynamic(const std::vector<int> &M, uint64_t N, uint64_t num_threads,
                                uint64_t chunk_size) {
  num_threads = std::min(num_threads, N/chunk_size + (N%chunk_size ? 1 : 0));
  uint64_t global_max_num_threads = N/chunk_size + (N%chunk_size ? 1 : 0);
  uint64_t global_diagonal = 0;
  std::atomic<uint64_t> global_current_num_threads {num_threads};
  std::atomic<uint64_t> global_lower {0};

  auto on_completion = [&global_diagonal, &global_lower, &global_max_num_threads, N,
                        chunk_size] {
    global_diagonal++;
    global_max_num_threads = (N-global_diagonal)/chunk_size +
                             ((N-global_diagonal)%chunk_size ? 1 : 0);
    global_lower.store(0);
  };

  std::barrier sync_point(num_threads, on_completion);

  auto task = [&] (const uint64_t id) -> void {
    uint64_t lower;
    uint64_t current_num_threads;

    while (true) {
      // fetch and add current global lower
      lower = global_lower.fetch_add(chunk_size);

      if (lower < (N-global_diagonal)) {
        // compute the upper border of the block (exclusive)
        const uint64_t upper = std::min(lower+chunk_size, N-global_diagonal);
        // compute task
        for (u_int64_t i = lower; i < upper; i++)
          work(std::chrono::microseconds(M[i*N + (i+global_diagonal)]));
      } else {
        current_num_threads = global_current_num_threads.load(std::memory_order_acquire);

        while (current_num_threads > global_max_num_threads) {
          if (global_current_num_threads.compare_exchange_weak(current_num_threads,
              current_num_threads-1, std::memory_order_release, std::memory_order_acquire)) {
            // barrier
            if (global_current_num_threads != 0)
              sync_point.arrive_and_drop();

            return;
          }
        }

        // barrier
        sync_point.arrive_and_wait();
      }
    }
  };

  .....
}
```

Listing 2: Wavefront parallel dynamic version.

3 Evaluation

In this section, the tests performed on the two implemented versions, static and dynamic, are discussed in order to evaluate their performance. The tests were conducted on the course machines (spmcluster.unipi.it and spmnuma.unipi.it). Each measurement was performed 10 times and the average value was considered. The standard deviation is negligible, and is not present in this document, because the values taken are very stable. The tests were performed by compiling the program with the following command:

```
g++ -std=c++20 -O3 -march=native -I ./include UTW.cpp -o UTW
```

During the compilation phase, you can define the `TEST` macro to enable an additional check that checks whether all matrix tasks have been executed.

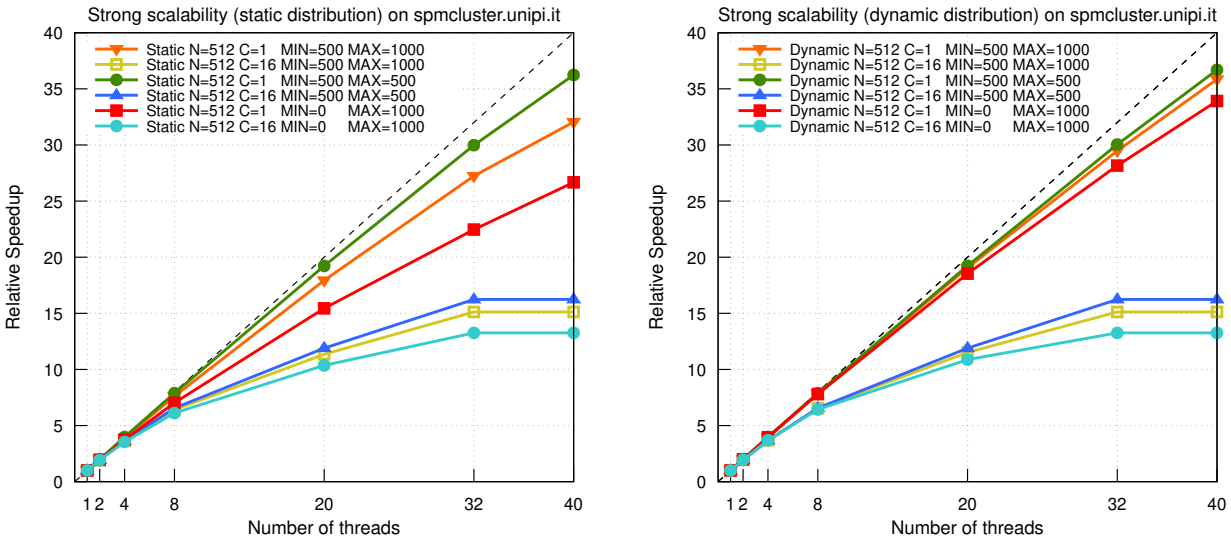
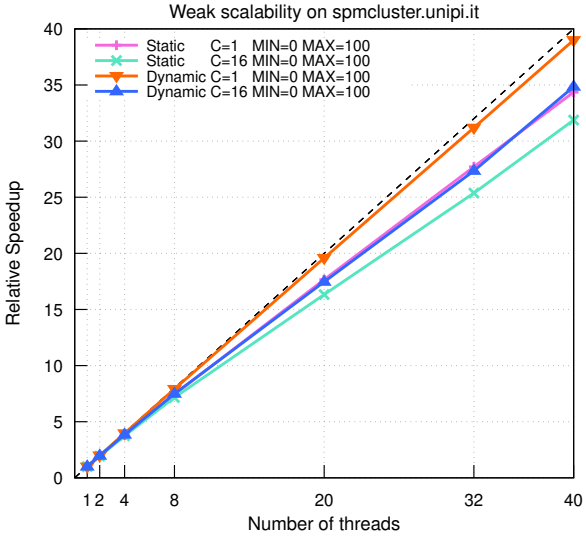


Figure 2: Relative Speedup vs Number of threads on spmcluster.unipi.it. (Strong scalability) `Left`: static version. `Right`: dynamic version.



Number of threads	N
1	512
2	768
4	1152
8	1728
16/20	2592
32	3888
40/64	5832

Figure 3: `Left`: Relative Speedup vs Number of threads on spmcluster.unipi.it. (Weak scalability). `Right`: matrix size used for each thread, valid for both clusters. (Weak scalability).

Scalability (or Relative Speedup) is the metric used to evaluate the performance of the two versions. Absolute Speedup was not considered because $T_{seq} \simeq T_{par}(1)$. Tests were performed by varying the number of processors while keeping the matrix fixed (strong scaling) and also by roughly proportionally increasing the size (weak scaling).

The graphs above show the scalability as the number of threads changes for the *spmcluster.unipi.it* cluster, which has a maximum of 40 logical cores.

In Figure 2, the static and dynamic versions are compared (strong scaling) by varying the workload (perfectly balanced $[500, 500]$; slightly unbalanced $[500, 1000]$; very unbalanced $[0, 1000]$) and the chunk size. As can be seen, the two versions are very similar if the workload is balanced. However, the dynamic version performs better on more unbalanced workloads. Regarding the chunk size, it is evident from the graph that increasing the chunk size makes the workload more unbalanced and reduces the maximum number of threads that can work on each diagonal, thus going to limit the performance for both versions.

In Figure 3, the two versions are analyzed by increasing the size of the matrix as the number of threads increases (weak scaling). As can be seen, the dynamic version has an almost perfectly linear performance.

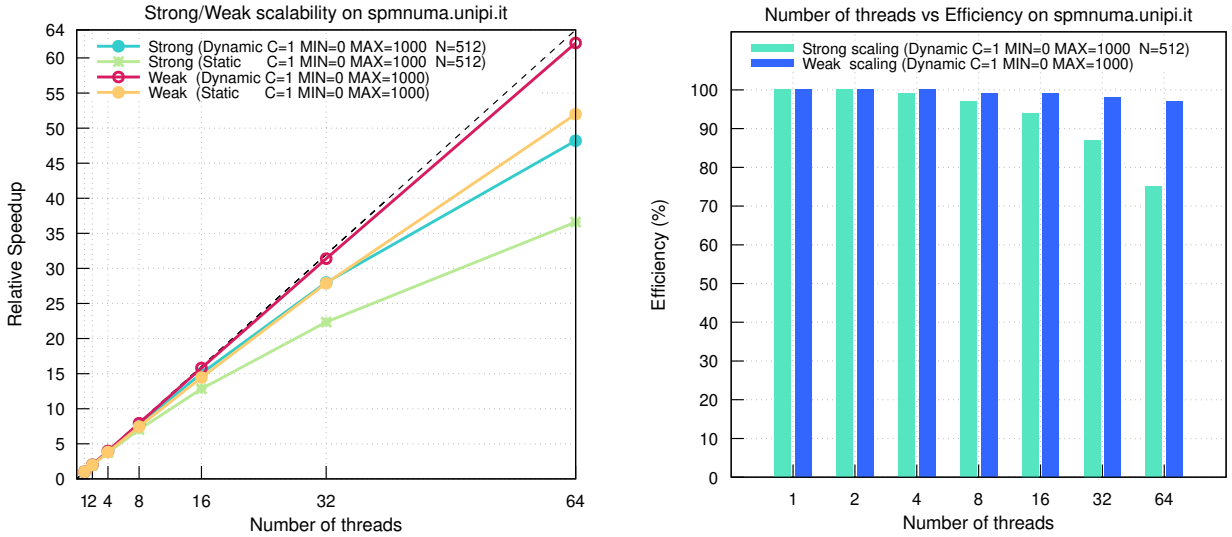


Figure 4: `Left`: Relative Speedup vs Number of threads on spmnuma.unipi.it. (Strong/Weak scalability). `Right`: Efficiency vs Number of threads on spmnuma.unipi.it. (Strong/Weak scalability, dynamic version).

In Figure 4, strong scalability is compared with weak scalability on the cluster *spmnuma.unipi.it*, which has 64 logical cores. The weak scalability as expected manages to scale much better, as seen in the graph on the left. On the right, however, the efficiency is shown, which is found to be constant for weak scalability and decreasing for strong scalability as the number of threads increases.

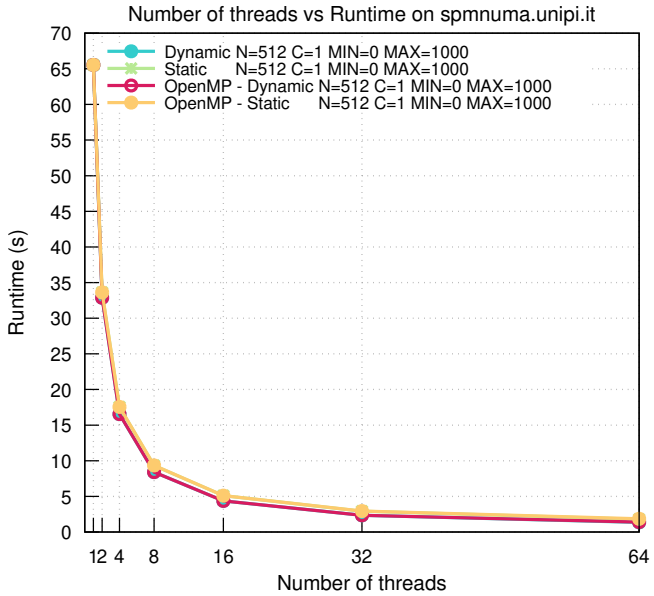


Figure 5: Runtime vs Number of Threads on spmnuma.unipi.it. (Strong scalability).

After these considerations the behavior in the above graphs is as expected, however to understand how good the two implementations are, in Figure 5, they were compared with the simpler implementation using the OpenMP library. What can be seen is that my implementation is slightly better by measuring the execution time. With this, it is not excluded that with more sophisticated implementations, using OpenMP, better results can be achieved.

4 Conclusions

To summarize, tests have shown that the dynamic version is generally better than the static version, using a small chunk size. The weak scaling as expected succeeds in achieving higher speedup than the strong scalability. The two versions as shown achieve the same speedup compared to the version made with the OpenMP standard.