CSCI 5451 Assignment 1: Parallel Dynamic Time Warping in OpenMP

Overview

In this assignment, you will implement a parallel variant of Dynamic Time Warping (DTW) — a classical dynamic programming algorithm used for aligning two ordered sequences. Your task is to compute the minimum alignment cost between two sequences of vectors under the following constraints:

- Every vector in both sequences must be used at least once.
- The alignment must be *monotonic* and *ordered*: if vector $\mathbf{A}[i]$ is paired with $\mathbf{B}[j]$ and vector $\mathbf{A}[k]$ is paired with $\mathbf{B}[\ell]$, then

$$i < k \implies j \leq \ell.$$

Formally, given sequences

- $\mathbf{A} = [a_1, a_2, \ldots, a_m]$
- $\mathbf{B} = [b_1, b_2, \ldots, b_n]$

where each a_i and b_j are vectors in $\mathbb{R}^d$, we seek a sequence of pairings that minimizes:

$$\text{Cost} = \sum_{(i,j)\in P} \|a_i - b_j\|_2$$

subject to the ordering and coverage constraints above (note that this is the ℓ_2 norm, so you should take the square root after computing the squared sum of distances).

Serial Algorithm + Pseudocode

We describe the structure of the serial algorithm in greater detail here.

The standard DTW algorithm computes a dynamic programming (DP) table D of size $m \times n$, where $D[i][j]$ is the minimum cost of aligning the prefixes $\mathbf{A}[0..i]$ and $\mathbf{B}[0..j]$. The recurrence is:

$$D[i][j] = \|a_i - b_j\|_2 + \min \begin{cases} D[i-1][j] \\ D[i][j-1] \\ D[i-1][j-1] \end{cases}$$

with boundary conditions:

- $D[0][0] = \|a_0 - b_0\|_2$
- $D[i][0] = \|a_i - b_0\|_2 + D[i-1][0]$
- $D[0][j] = \|a_0 - b_j\|_2 + D[0][j-1]$

At the end, $D[m-1][n-1]$ gives the Minimum Alignment Cost. The pseudocode for this algorithm is as follows:

Algorithm 1: Dynamic Time Warping (DTW)

Input: Sequences $\mathbf{A}[0..m-1]$, $\mathbf{B}[0..n-1]$ (vectors of dimension d)
Output: Minimum alignment cost $D[m-1][n-1]$
function DTW($\mathbf{A}, \mathbf{B}$);

```
let D[0..m-1][0..n-1];
let dist[0..m-1][0..n-1];
// Set first timer after allocating these arrays
for i ← 0 to m − 1 do
    for j ← 0 to n − 1 do
        dist[i][j] ← ‖A[i] − B[j]‖₂;
    end
end
D[0][0] ← dist[0][0];
for i ← 1 to m − 1 do
    D[i][0] ← dist[i][0] + D[i − 1][0];
end
for j ← 1 to n − 1 do
    D[0][j] ← dist[0][j] + D[0][j − 1];
end
for i ← 1 to m − 1 do
    for j ← 1 to n − 1 do
        D[i][j] ← dist[i][j] + min(D[i − 1][j], D[i][j − 1], D[i − 1][j − 1]);
    end
end
// Set last timer after completing the above computation
return D[m − 1][n − 1];
```

Tests Format

In the zip file for this assignment on the course site, you will find the following:

- An executable file named `dtw_serial`. This can be run with

  ```
  ./dtw_serial <test_name>
  ```

  where `<test_name>` is one of the tests described in the next bullet point (e.g. `tests/testin_0.txt`. The output of this file will be two lines - one of which prints the timing of the computation (as defined in Algorithm 1) and the other the Minimum Alignment Cost. The output should look as follows (with different numbers depending on the test you ran).

  ```
  The Final Cost is:  67082.067588
  The Total Completion time (ms) is:  20499.982
  ```

  You will be using this file to benchmark your times and see if your own implementation is correct. You should expect to see improvements in the time taken over this baseline when running in parallel (the exact improvements are defined later in this homework description).

- The zip file will also contain a folder named `tests`. There will be a set of unit tests in this folder which you will use to determine the correctness of your program as well as show speedups. Each single test corresponds to an input file (e.g. `tests/testin_0.txt`) and an output file (e.g. `tests/testout_0.txt`). Each input file will have the following structure:

  ```
  Matrix A
  1 4
  2 7
  3 8
  4 9
  Matrix B
  2 10
  3 13
  ```

  Each output file will simply consist of a single double value (e.g. `67082.067588`) corresponding to the correct Minimum Alignment Cost for the input file. You can assume that all tests will be properly formatted. That is, they will have integers in [-10, 10], each row (besides the rows labeled `Matrix A` and `Matrix B`) will have the same number of entries. However, the number of rows in **A** and **B** can be different. You must use these files to determine the correctness of your approach. Additionally, you will benchmark your speedups using Test 5 (the other tests are too small for the purposes of demonstrating speedups, and are meant only to show correctness).

Submission Format

For the purposes of this assignment, you must turn in the following two items in Canvas:

- A file containing the entirety of your parallel program named `dtw_parallel.c` (this file should not be zipped - it should just be the file itself). You *must* place your whole program in this file. We will compile this program with the following lines

  ```
  OMP_PROC_BIND=spread
  OMP_PLACES=cores
  gcc -O3 -fopenmp dtw_parallel.c -o dtw_parallel -lm
  ```

  The two environment variables above ensure that we will be using each core separately. The above *must* run without error on the `plate` machines.

  We will then run your programs with different numbers of threads + unit tests using the following format

  ```
  ./dtw_parallel <test_name> <num_threads>
  ```

  where `<test_name>` is the input test file and `<num_threads>`. Your file *must* accept these arguments and *must* ensure that the number of threads we set at the command line is used by your program. Your program *must* print out the following lines exactly:

  ```
  The Final Cost is:  <cost>
  The Total Completion time (ms) is:  <time>
  ```

  where `<cost>` is the Minimum Alignment Cost and `<time>` is the time it takes for your program to execute in milliseconds. You *must* place your timers at the locations defined inside of Algorithm 1 (we only want to consider the amount of time required by actual computation and will be omitting allocation/de-allocation/file-reading). Additionally, you *must* use the `omp_get_wtime` function at both of these locations (this function records the wall-clock time in seconds, which you then must multiply by 1000 to get the correct program time in milliseconds).

  We will be running tests to ensure that your program compiles, runs without error, has no memory leaks, correctly outputs the appropriate `<cost>` for each test and demonstrates speedups on problems by examining the `<time>` outputs. For the purposes of achieving speedups, you should use the `tests/testin_5.txt` file as the other tests you are given will be too small to see speedups.

- A pdf report, which contains (1) a section describing what parts of the serial program you parallelized and how. And (2) the times reported by your program when run on the `tests/testin_5.txt` test with `(1, 2, 4, 8, 16, 32, 64)` threads. You should show these timings by presenting either a table or a line plot.

The two files above are due

October 12 @ 11:59pm on Canvas

Grading

This homework assignment will be worth 15 points in total. The points will be allocated as follows:

- (4 points) Correctness. The program compiles, runs without error, has no memory leaks, & runs correctly on all unit tests (including some hidden tests). For ensuring your program has no memory leaks, see Valgrind.
- (8 points) Speedups. The program achieves sufficient speedups for each *num_threads* value on different unit tests (some of which will be internal). Your program must achieve speedups as described below in order for full points. Additionally, in order to get points at this stage, you must make use of openmp (if your program does not use openmp, and instead relies on external libraries, you will receive no points at this stage).

  Relative expected speedups will be based on Table 1, where we will use the serial executable in the homework zip file (`dtw_serial`) to determine what the *relative* time is when running on 1 thread.

num_threads	**relative_time**
1	1.000
2	0.600
4	0.360
8	0.216
16	0.130
32	0.078
64	0.047

Table 1: Relative time for program execution with increasing number of threads

  As an example of how this works in practice, Table 2 shows the execution times we will be using for `tests/testin_5.txt`. Your program *must* be

num_threads	absolute_time (ms)
1	21000
2	12600
4	7560
8	4536
16	2730
32	1638
64	987

Table 2: Absolute execution time with increasing number of threads on `tests/testin_5.txt`

lower than each of the values in this table in order to get full points for this unit test.

- (3 points) Report. The report accurately describes parallelization strategies and includes a speedup figure.

Tips

- Check to see that the `OMP_DYNAMIC` environment variable is set to false.
- Be sure to set `OMP_PROC_BIND=spread` and `OMP_PLACES=cores` as described above to ensure that you are using separate cores (hyperthreading does not *always* help with speedups - especially when the same execution units are ued). There are only 68 physical cores on this machine, so we deliberately use fewer than this.
- You may have to contend with many other students running their programs at similar times. If you do not see speedups, it can be helpful to run a few times, and with a smaller number of threads as a sanity check.
- The tests in this assignment are not exhaustive - we have some of our own tests which we will be using to ensure that you properly achieve speedups and are correct. Be sure to think through potential edge cases to make sure your program is correct.
- If you have a nested for loop, and are only parallelizing the inner loop (i.e. you are sequentially going through the outer for-loop), it can be useful to only dispatch threads once, rather than repeatedly fork-join on each iteration of the outer loop.
- Keep an eye out for dependencies in the serial version of program to ensure that your parallel program will have the same outputs
- You will have to convert from the integers in each file to double in your program as computing the distances will require that you take square

roots. When reading from the files, first load in as integers, then convert to doubles.

- Start by getting a serial version working first which passes the unit tests and matches the `dtw_serial` executable on speed. Only once you have successfully programmed this should you move on to adding OpenMP directives.
- In order to get closer to the time of `dtw_serial`, do not use double pointers when allocating `D` and `dist` - these should be single pointers and the logic for accessing an entry at a given row and column should be performed manually by you. Doubly nested pointers will not be contiguous in memory and can slow down your program.