Lab 2: Parallel Longest Common Subsequence

姜海天 19307110022

0.1 Problem formulation

This report is to record the performance optimization for longest common subsequence. Let A[0..M-1] be a string of length M and B[0..N-1] be a string of length N. Then the state-transition equation is:

$$dp(i,j) = \begin{cases} 0, & i < 0 \text{ or } j < 0 \\ dp(i-1,j-1)+1, & A[i] = B[j] \\ \max\{dp(i,j-1), dp(i-1,j)\}, & A[i] \neq B[j] \end{cases}$$

The naive implementation is

```
int naive_lcs_2d(const std::string A, const std::string B) {
    const int M = A.length();
    const int N = B.length();
    int * dp = (int *)calloc(M * N, sizeof(int));
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            int up = (i > 0) ? dp[(i - 1) * N + j] : 0;
            int left = (j > 0) ? dp[i * N + j - 1] : 0;
            int upleft = (i > 0 && j > 0) ? dp[(i - 1) * N + j - 1] : 0;
            if (A[i] == B[j])
                dp[i * N + j] = upleft + 1;
            else
                dp[i * N + j] = max(up, left);
        }
    }
    int res = dp[M * N - 1];
    free(dp);
    return res;
}
```

0.2 Test environment

All the code are tested on the testing machine of this course. It has a 8-core, 8-thread `Intel Xeon E5-2650` CPU. L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 20MiB

1 1-D Space Usage for LCS

One important observation in the naive 2-D implementation is, in each iteration of the outer loop we only need values from all columns of the previous row. So there is no need to store all rows in our dp matrix, we can just store two rows at a time and use them. In that way, used space will be reduced from `dp[m+1][n+1]` to `dp[2][n+1]`. Below is the implementation of the above idea. Here I use the transpose of the dp matrix to have a better cache hit rate. Notice that few codes need to be revised compared with the 2-D version.

```
int lcs_basic(const char * _A, const char * _B, int M, int N) {
    int (*dp)[2] = (int(*)[2])calloc(N * 2, sizeof(int));
    bool bi;
    for (int i = 0; i < M; ++i) {
        bi = i & 1;
        for (int j = 0; j < N; ++j) {
            int up = (i > 0) ? dp[j][1 - bi] : 0;
            int left = (j > 0) ? dp[j - 1][bi] : 0;
            int upleft = (i > 0 && j > 0) ? dp[j - 1][1 - bi] : 0;
            if (_A[i] == _B[j]) {
                dp[j][bi] = upleft + 1;
            } else {
                dp[j][bi] = max(up, left);
            }
        }
    }
    int res = dp[N - 1][bi];
    free(dp);
    return res;
}
```

2 1-D by Anti-Diagonal

In this case, we can fill the dp matrix by looping over s=i+j from 0 to M+N-2. The inner loop can be t=j-i from -M+1 to N-1. To make the index bigger than 0, we can let t=j-i+M-1 ranging from 0 to M+N-2. So now we can have the sequential version of the anti-diagonal method, and there is no data dependency in the inner loop.

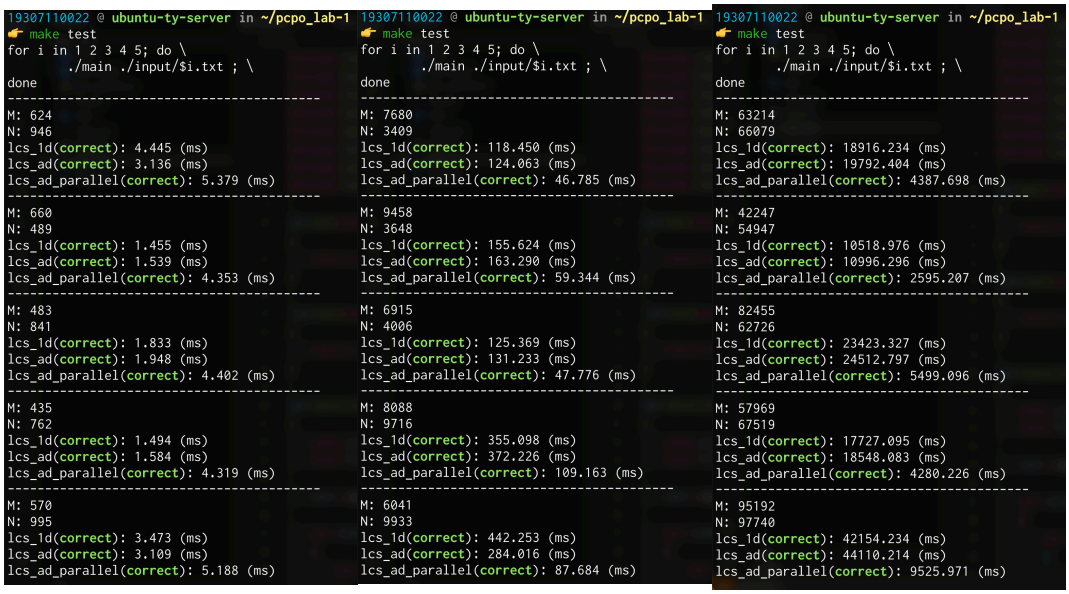
```
int lcs_ad(const char * _A, const char * _B, int M, int N) {
    int* dp = (int*)calloc(M + N - 1, sizeof(int));
    for (int s = 0; s < M + N - 1; ++s) {  // s = i + j
        int start = (s < M) ? (M - s -1) : (s - M + 1);
        int end = (s < N) ? (M + s + 1) : (N * 2 + M - s - 1);
        for (int t = start; t < end; t += 2) {  // t = j - i + M - 1
            int i = (s + M - t - 1) / 2;
            int j = (s + t + 1 - M) / 2;
```

```
            if (_A[i] == _B[j]) {
                ++dp[t];
            } else {
                dp[t] = max(dp[t-1], dp[t+1]);
            }
        }
    }
    int res = dp[N-1];
    free(dp);
    return res;
}
```

3 Parallel Version

To parallelize the above code, we can just add the OpenMP primitive for the inner loop.

By just adding the `#pragma omp parallel for`, I can get the following result:



We can see that for small M and N below 1K, the overhead of creating threads takes the majority of time so that the parallel version takes longer to run. But for M and N below 10K, the parallel version takes dramatic less time to run, and the speedup is about **3**. When the length of string goes larger to below 100K, the speedup comes to about **4.5**.

Considering different threads will write to nearby positions of the memory simultaneously in the parallel version, the written value in the cache line of one processor may cause the cache line in other processor to be invalidated. And this kind of situation may happen back and forth so that the invalidation storm causes lots of communication in the hardware. One thought is to unroll the loop and make it parallel at the same time. So that one processor will be in charge of the whole cache line, and do not interfere with other processors. In OpenMP, this is simply a longer primitive: `#pragma omp parallel for schedule(static 8)`. But it actually makes the computation time longer.