COMP90025 Assignment 02a Written Report

Xulin Yang
904904
The University of Melbourne
`xuliny@student.unimelb.edu.au`

September 12, 2020

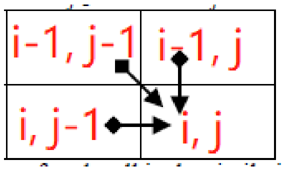


Figure 1: Matrix cell dependency

As we have the cell calculation dependencies (figure 1) for the dynamic programming (dp) approach in sequential algorithm.

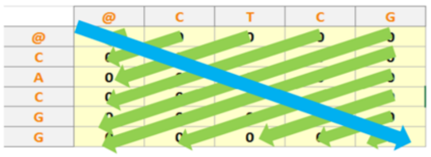


Figure 2: Anti-diagonal matrix traverse

Instead of changing to a new algorithm, first intuitive optimization is to traverse the dp matrix in an anti-diagonal manner (figure 2). Sequentially iterate through each anti-diagonal and parallel dp cell calculation on each anti-diagonal rather than calculating cells in sequential row by row manner. However, this achieves no speedup because there can be a lot of cache missing and false sharing due to the way we traverse the matrix.

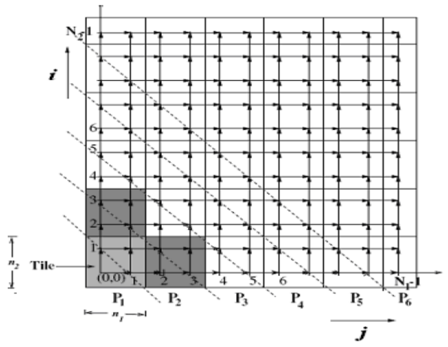
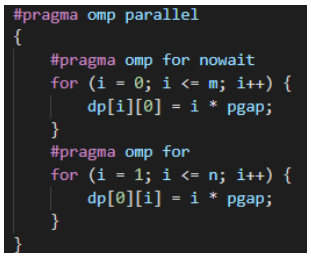


Figure 3: Tiled matrix & Diagonal tile parallelism

Sathe and Shrimankar (2011) describes a tiling parallel algorithm. Rather than parallel each cell diagonally, we parallel **tiles** diagonally. Each tile has $\frac{matrix\ width}{p} * \frac{matrix\ height}{p}$ cells where p =#processors (figure 3). Then we apply the anti-diagonal parallelism technique on tiles and calculate cells sequentially in each tile (figure 5) to reduce cache missing. On spartan, I found that 22 threads performs best with this algorithm.

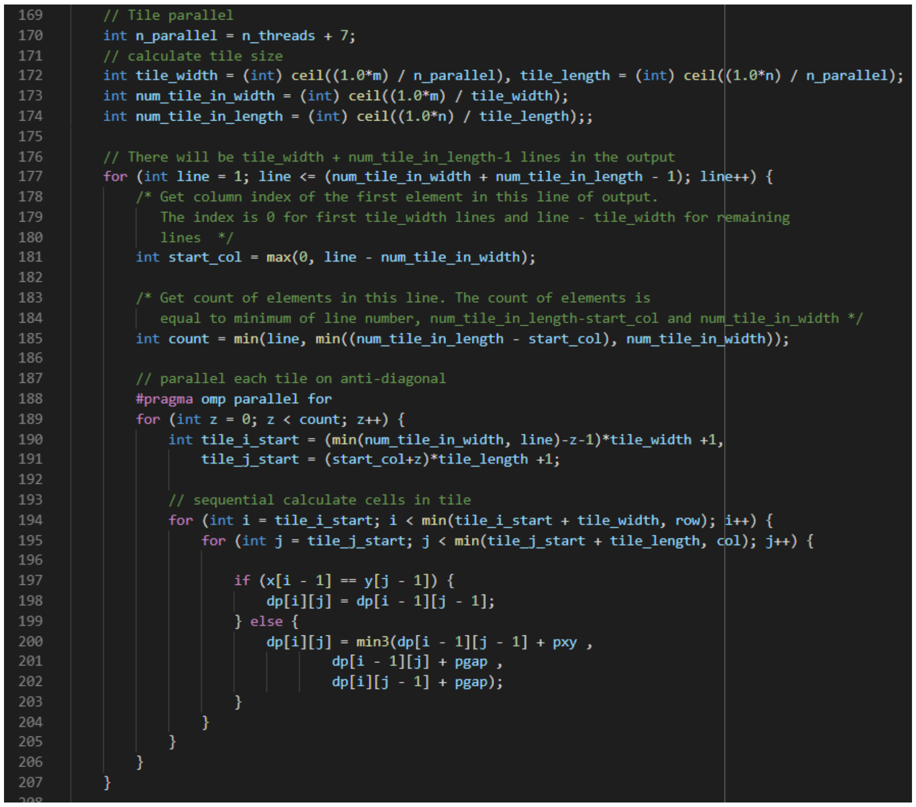
Other speedup techniques:

1. add "#pragma omp parallel for" for the two matrix initialization for loops without implicit barriers (figure 4).
2. delete unnecessary "memset (dp[0], 0, size);" in given code.
3. delete "delete[] dp[0]; delete[] dp;" to speedup.
4. no need to do last branching condition (figure 6).
5. Rather than choose p =#processors, choose $p = \frac{4}{3}$*#processors to maximize threads usage.



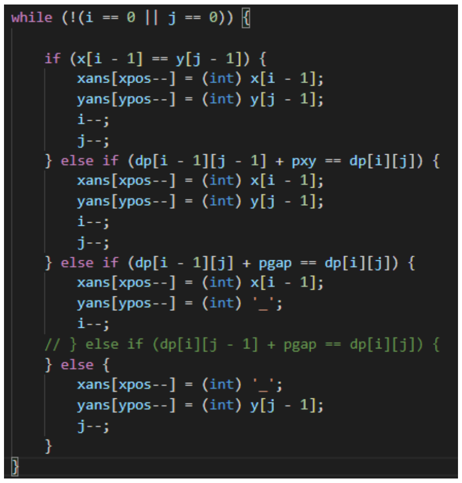
```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i <= m; i++) {
        dp[i][0] = i * pgap;
    }
    #pragma omp for
    for (i = 1; i <= n; i++) {
        dp[0][i] = i * pgap;
    }
}
```

Figure 4: Matrix initialization parallelism



```
    // Tile parallel
    int n_parallel = n_threads + 7;
    // calculate tile size
    int tile_width = (int) ceil((1.0*m) / n_parallel), tile_length = (int) ceil((1.0*n) / n_parallel);
    int num_tile_in_width = (int) ceil((1.0*m) / tile_width);
    int num_tile_in_length = (int) ceil((1.0*n) / tile_length);;

    // There will be tile_width + num_tile_in_length-1 lines in the output
    for (int line = 1; line <= (num_tile_in_width + num_tile_in_length - 1); line++) {
        /* Get column index of the first element in this line of output.
           The index is 0 for first tile_width lines and line - tile_width for remaining
           lines  */
        int start_col = max(0, line - num_tile_in_width);

        /* Get count of elements in this line. The count of elements is
           equal to minimum of line number, num_tile_in_length-start_col and num_tile_in_width */
        int count = min(line, min((num_tile_in_length - start_col), num_tile_in_width));

        // parallel each tile on anti-diagonal
        #pragma omp parallel for
        for (int z = 0; z < count; z++) {
            int tile_i_start = (min(num_tile_in_width, line)-z-1)*tile_width +1,
                tile_j_start = (start_col+z)*tile_length +1;

            // sequential calculate cells in tile
            for (int i = tile_i_start; i < min(tile_i_start + tile_width, row); i++) {
                for (int j = tile_j_start; j < min(tile_j_start + tile_length, col); j++) {

                    if (x[i - 1] == y[j - 1]) {
                        dp[i][j] = dp[i - 1][j - 1];
                    } else {
                        dp[i][j] = min3(dp[i - 1][j - 1] + pxy ,
                                dp[i - 1][j] + pgap ,
                                dp[i][j - 1] + pgap);
                    }
                }
            }
        }
    }
```

Figure 5: Anti-diagonal tile parallelism & sequential traverse in tile



```
while (!(i == 0 || j == 0)) {

    if (x[i - 1] == y[j - 1]) {
        xans[xpos--] = (int) x[i - 1];
        yans[ypos--] = (int) y[j - 1];
        i--;
        j--;
    } else if (dp[i - 1][j - 1] + pxy == dp[i][j]) {
        xans[xpos--] = (int) x[i - 1];
        yans[ypos--] = (int) y[j - 1];
        i--;
        j--;
    } else if (dp[i - 1][j] + pgap == dp[i][j]) {
        xans[xpos--] = (int) x[i - 1];
        yans[ypos--] = (int) '_';
        i--;
    // } else if (dp[i][j - 1] + pgap == dp[i][j]) {
    } else {
        xans[xpos--] = (int) '_';
        yans[ypos--] = (int) y[j - 1];
        j--;
    }
}
```

Figure 6: Backtracking optimization

References

S. R. Sathe and D. D. Shrimankar. Parallelization of DNA sequence alignment using OpenMP. *ACM International Conference Proceeding Series*, pages 200–203, 2011. doi: 10.1145/1947940.1947983.