ECE1747H Assignment 1 Report

Yizhou Shen (1004536308)
Yuanze Yang (1009209269)
Haocheng Wei (1008498261)

Implemented Improvements

In our parallel version code, we tried 3 approaches and successfully implemented 2 ways of transforming the for-loop from the function “sequentialSolution”. Here we present the different versions of the code.
The original sequential logic goes as the Figure 1.

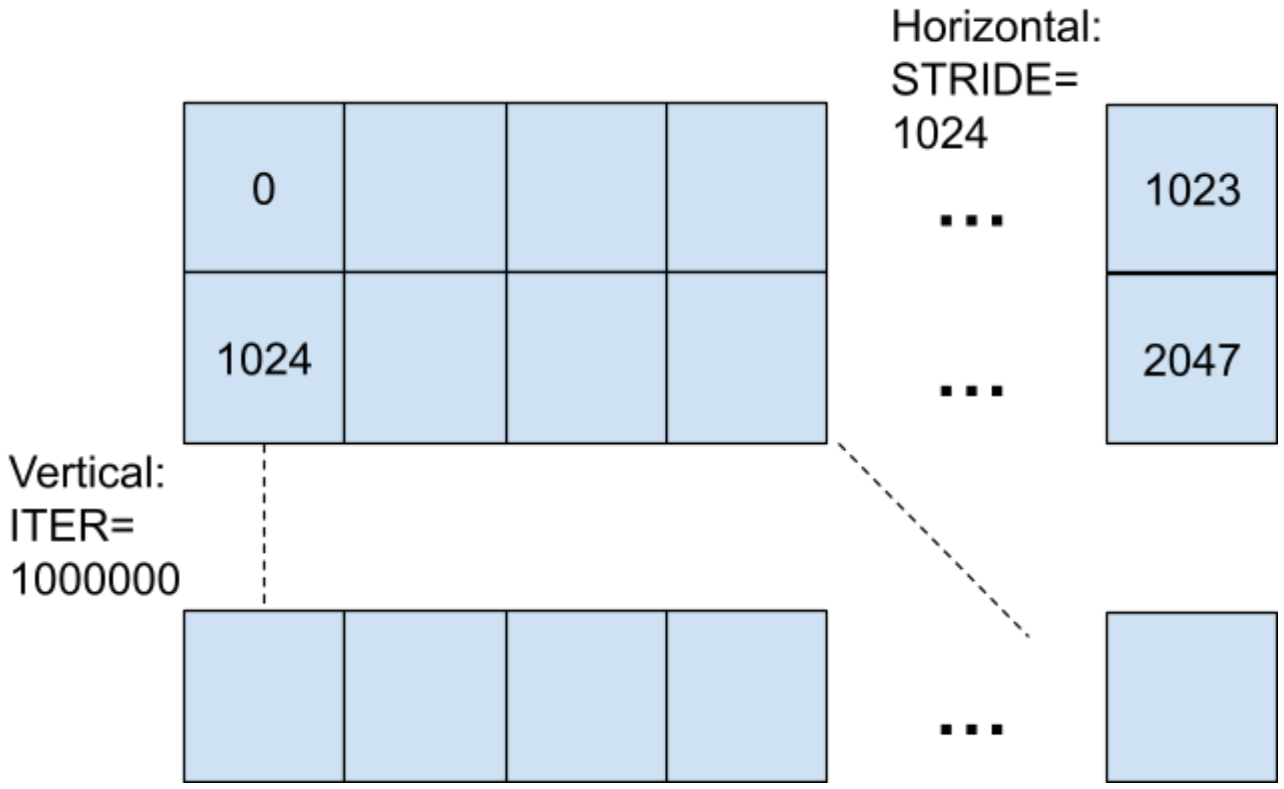

Figure 1: Original sequential logic

In the original logic, the 2 rows which are vertically adjacent have dependencies. The upper row would write the lower row with its value returned by the function “transform”. In the next iteration, the lower row will become the upper row and it’s value will be further written into the adjacent row below it.

Version one tries to parallelize the operations within each row since the indexs in the same row have no dependency. For example, writing the transform of A[0] into A[1024] will happen in parallel with writing the transform of A[1] into A[1025], A[2] into A[1026], etc.

```
46
47 void parallelSolution1(int *A){
48     for (int i = 0; i < ITER; i++)
49         #pragma omp parallel for
50         for (int j = i * STRIDE; j < (i + 1) * STRIDE; j++)
51             A[j + STRIDE] = transform(A[j]);
52 }
```

Figure 2: Parallel version 1 code

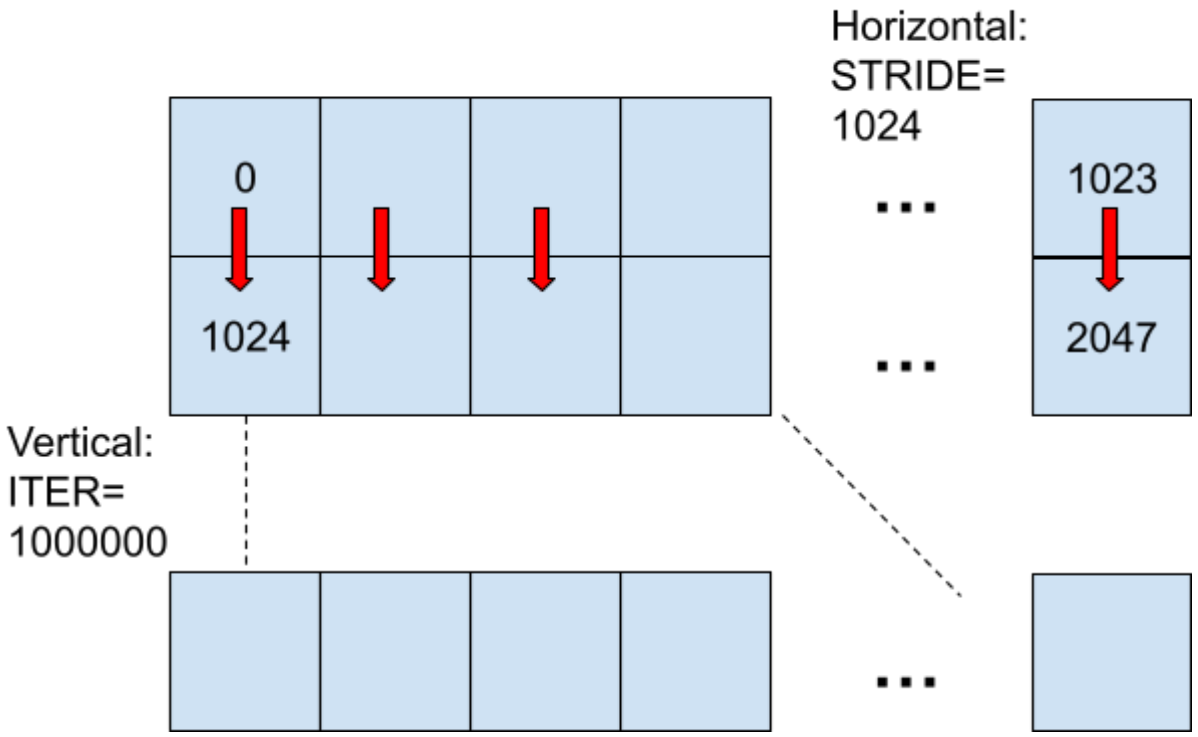

Figure 3: Version 1 workflow

In the second version of parallelization, we still firstly splitted the original logic into the inner and outer loop and parallelized the outer one. The key in this idea is that, we parallelize the “column” line of interdependent execution of writing. That is to say, we have different threads moving through the chain of execution for each column in the array. Diagram of this operation is shown in figure 5, in which the red arrow shows one particular chain of dependency for a thread.

```
53
54 void parallelSolution2(int *A){
55     #pragma omp parallel for
56     for (int i = 0; i < STRIDE; i++) {
57         for (int j = 0; j < ITER; j++) {
58             A[i + STRIDE * (j + 1)] = transform(A[i + j * STRIDE]);
59         }
60     }
61 }
62
```

Figure 4: Parallel version 2

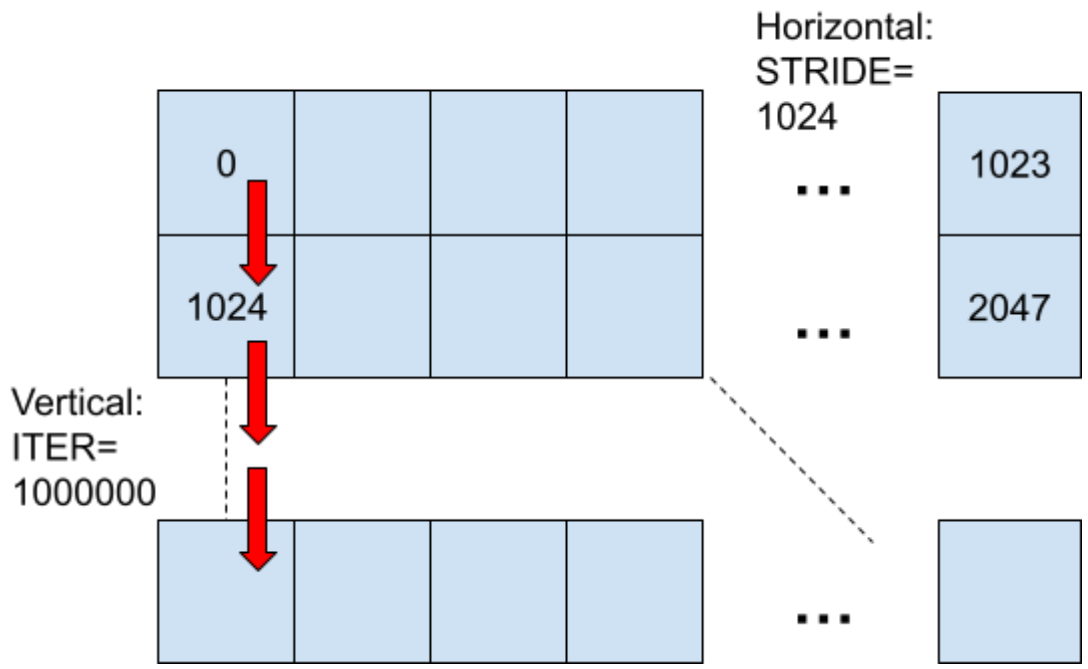

Figure 5: Version 2 workflow

Performance Comparison

From table 1, we can conclude that version 1 reduces the processing time from 6.47 seconds to 2.86 seconds due to the parallelization within each row. However, version 2 cost even more time. The potential costs come from the poor cache locality since the same thread jumps through the array, resulting in a high cache miss rate.

	Sequential	Parallel_1	Parallel_2
Time	6.473971	2.858041	9.112596

Table 1: Time used for sequential and different parallel versions of code