# 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 3 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.

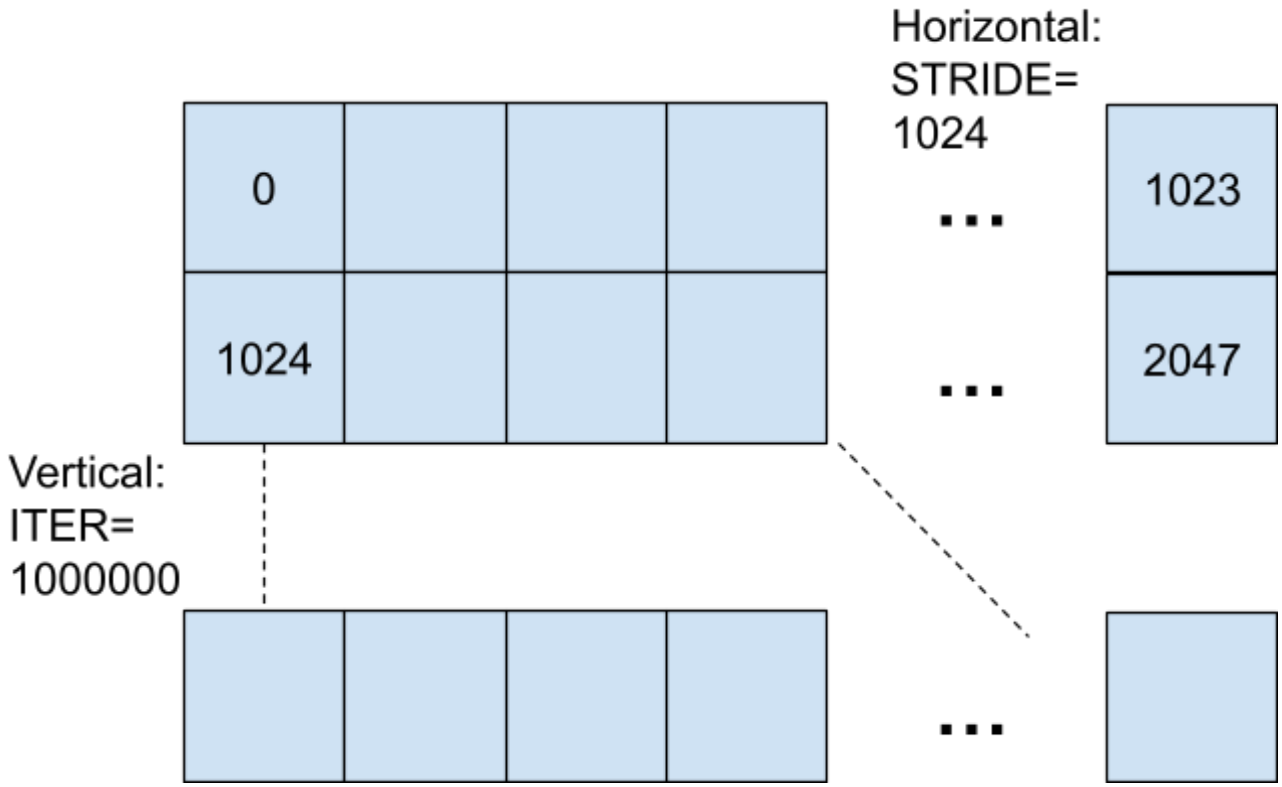


**Figure 1:** Original sequential logic

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.

### 1. Parallel Solution 1

Version one tries to parallelize the operations within each row since the indexs in the same row have no dependence. 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.

```
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

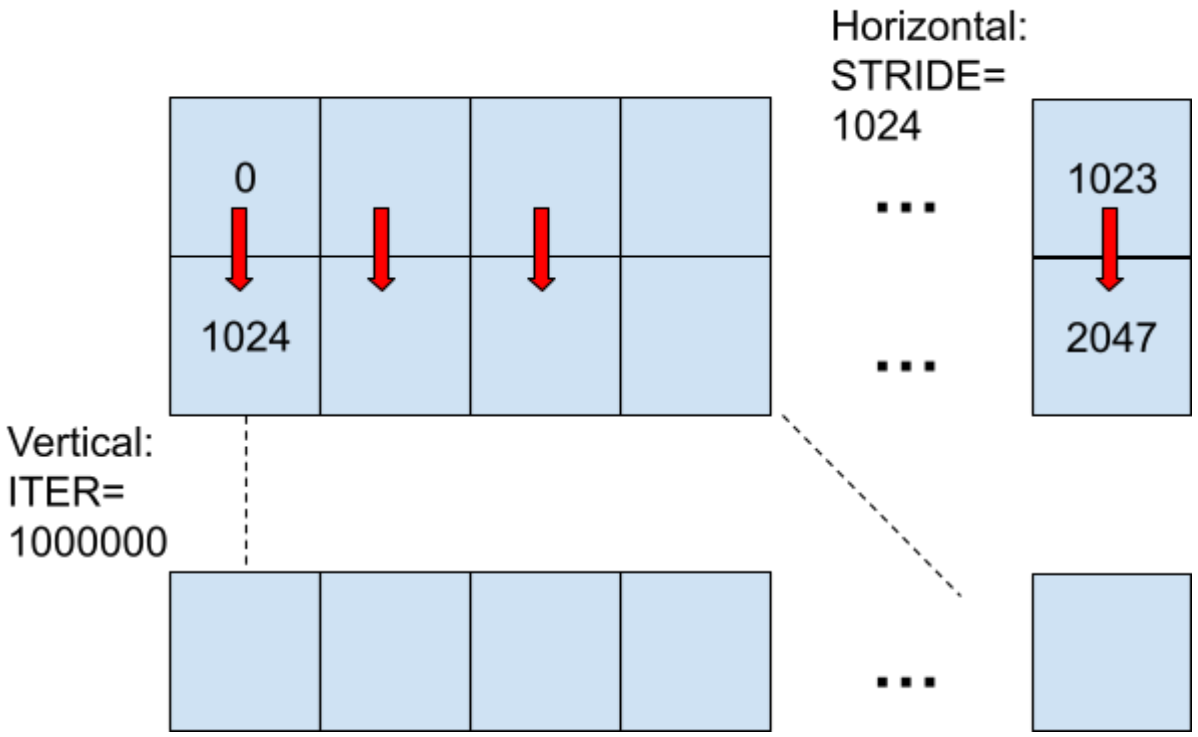


**Figure 3**: Version 1 workflow

## 2. Parallel Solution 2

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 dependence for a thread.

```
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  }
```

**Figure 4:** Parallel version 2

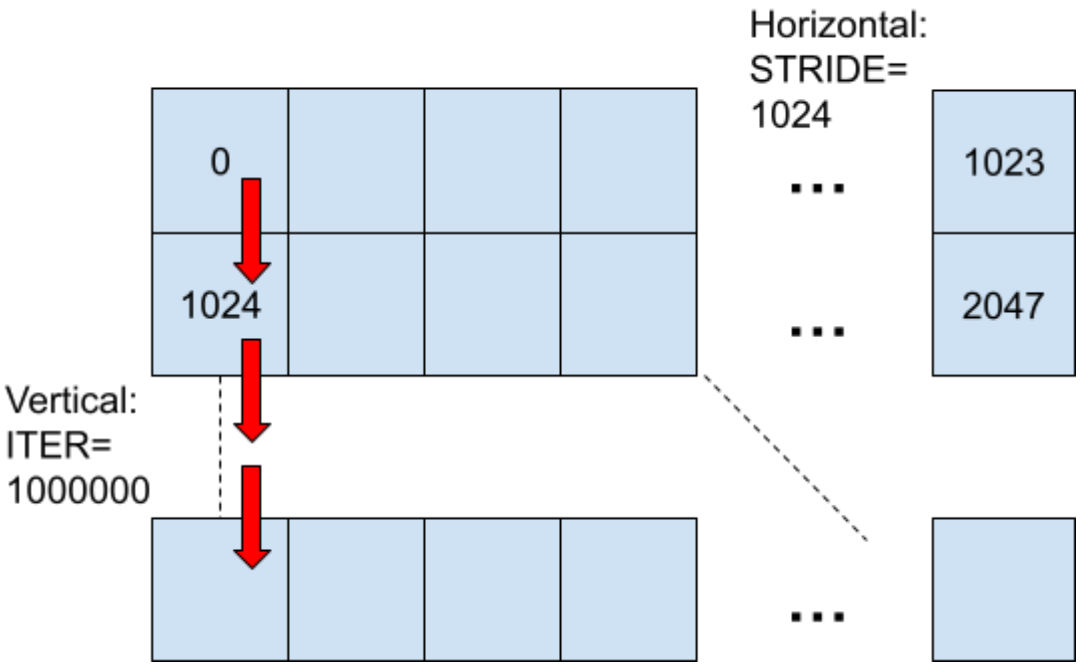


**Figure 5:** Version 2 workflow

### 3. Parallel Solution 3

In the third version, we reorgnized the structure of the code while still maintaining the original intent. The value in lower rows come from the transform of adjacent upper rows, causing the value in the j-th row A[(j-1)*STRIDE+i] equals to the value in the first row A[i] called by transform() function for (j-1) times. However, since the transform function converts an integer into a rounding perfect square number, calling transform function once returns the same value as calling transform function multiple times. Therefore, we reduce the dependences between adjacent rows such that values in whichever row only depend on the values in the first row, and these values are equal to the values in the first row called by transform function once.

```
63  void parallelSolution3(int *A){
64      #pragma omp parallel for
65      for (int i = 1; i < ITER; i++) {
66          int k = i * STRIDE;
67          for (int j = 0; j < STRIDE; j++) {
68              A[k + j] = transform(A[j]);
69          }
70      }
71  }
```

**Figure 5:** Parallel version 3

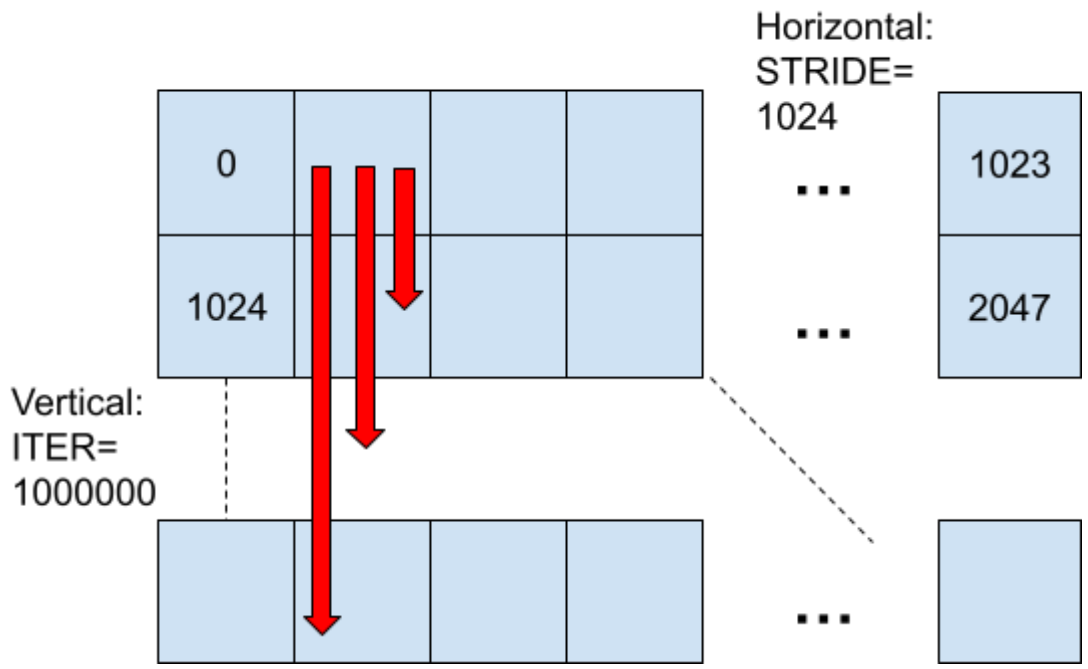


**Figure 6:** Version 3 workflow

### Performance Comparison

From table 1, we can conclude that version 1 reduces the processing time from 18.91 seconds to 6.63 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. Version 3 cost the least amount of time since the code is restructured with low synchronization and good cache locality.

| | Sequential | Parallel_1 | Parallel_2 | Parallel_3 |
|---|---|---|---|---|
| Time | 18.912055 | 6.625649 | 9.745826 | 3.123658 |

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