Malay Pandey
CS 610A

Assignment II

Roll No: 22111090
September 13, 2023

Problem 1

Command for running the code:

```
g++ 22111090-prob1.cpp -lpthread -o prob1.out
./prob1.out <N> <M> <path>
```

Here, `<N>` is the number of producer threads, `<M>` is the number of consumer threads, and `<path>` is the path for the input file that lists the full paths of all the `N` source files that are to be processed by `N` producers. Please note that the number of lines in the input file should be the same as `<N>` given as the command line argument, otherwise the assert function will fail and the code wouldn't produce result in that case. The frequency of each word is printed in lexicographical order to the terminal.

Problem 2

Command for running the code:

```
g++ 22111090-prob2.cpp -lboost_system -o prob2.out
./prob2.out <N> <M> <path>
```

The code in problem 1 has been re-implemented using *Boost Lockfree* library. Here, `<N>` is the number of producer threads, `<M>` is the number of consumer threads, and `<path>` is the path for the input file that lists the full paths of all the `N` source files that are to be processed by `N` producers. Please note that the number of lines in the input file should be the same as `<N>` given as the command line argument, otherwise the assert function will fail and the code wouldn't produce result in that case. The frequency of each word is printed in lexicographical order to the terminal.

Problem 3

```
int i, j, t, k;
for (t = 0; t < 1024; t++) {
    for (i = t; i < 1024; i++) {
        for (j = t; j < i; j++) {
            for (k = 1; k < j; k++) {
                S(t, i, j, k);
            }
        }
    }
}
```

Assignment II

3.a

We are given d_1 = `(1,0,-1,1)`, d_2 = `(1,-1,0,1)` and d_3 = `(0,1,0,-1)`. A permutation is valid if it's not lexicographically negative.

- `tijk` → `jitk`: Invalid permutation since d_1 = `(1,0,-1,1)` → `(-1,0,1,1)`
- `tijk` → `itjk`: Invalid permutation since d_2 = `(1,-1,0,1)` → `(-1,1,0,1)`
- `tijk` → `kijt`: Invalid permutation since d_3 = `(0,1,0,-1)` → `(-1,1,0,0)`
- `tijk` → `tkji`: Invalid permutation since d_3 = `(0,1,0,-1)` → `(0,-1,0,1)`
- `tijk` → `tjki`: Invalid permutation since d_3 = `(0,1,0,-1)` → `(0,0,-1,1)`

From the above observations, we can conclude that no other loop except t can be the outermost loop. Also, loop i cannot go inside loop k, since it would lead to an invalid permutation of d_3. Hence, `tijk`, `tikj`, `tjik` are the only valid permutations of the loop.

3.b

Complete unroll/jam of a loop is equivalent to a loop permutation that moves that loop innermost, without changing the order of other loops. We have seen in the answer to 3.a, that we cannot have any valid permutation, wherein we change the position of loop t. Therefore, unroll/jamming cannot be performed on the loop t. Similarly, unrolling i would lead to it being the innermost loop, even inside loop k, which is invalid as seen in the previous answer. Therefore, unroll/jamming cannot be performed on loop i as well.

- `t`: Invalid to unroll/jam since d_2 = `(1,-1,0,1)` → `(-1,0,1,1)` (Invalid permutation)
- `i`: Invalid to unroll/jam since d_3 = `(0,1,0,-1)` → `(0,0,-1,1)` (Invalid permutation)
- `j`: Valid to unroll/jam since d_1 = `(1,0,-1,1)` → `(1,0,1,-1)`
  d_2 = `(1,-1,0,1)` → `(1,-1,1,0)` & d_3 = `(0,1,0,-1)` → `(0,1,-1,0)` (All valid permutations)
- `k`: Valid to unroll/jam since it's the innermost loop, hence the dependence vectors don't change. The innermost loops are always unrollable.

3.c

A contiguous band of loops can be tiled if they are fully permutable, i.e. all permutations of the loops in that band are valid. We have already seen in the answer to the previous questions that we cannot permute the loop band `tijk` since the permutations will be invalid. We can also notice

that the outermost loop t cannot form a band with any of the inner loops, therefore we cannot tile the loop t with anyone else. Similarly, tiling of the loop band `ijk` would also be invalid, since some of the permutations (eg. `jki`) are invalid. However, we can tile the loop band `ij`, since all the permutations of this band are valid. Similarly, we can tile the loop band `jk`, since all the permutations of this band are valid as well.

- `tijk`: Invalid tiling (Invalid permutations - `itjk`, `kijt`, etc.)
- `tij`: Invalid tiling (Invalid permutations - `itj`, `jit`, etc.)
- `ti`: Invalid tiling (Invalid permutation - `it`)
- `ijk`: Invalid tiling (Invalid permutation - `jki`)
- `ij`: Valid tiling
- `jk`: Valid tiling

3.d

	t	**i**	**j**	**k**
$\mathbf{d_1}$	+	0	-	+
$\mathbf{d_2}$	+	-	0	+
$\mathbf{d_3}$	0	+	0	-

As can be seen from the table above, the first two dependences are being carried by loop t, while the third dependence is being carried by loop i. Therefore, we cannot parallelize/shift these loops, we can only parallelize loops j and k.

3.e

Ignoring the dependencies mentioned in the question, to find the correct iteration space for the *tikj* permutation, we'll have to perform Fourier-Motzkin elimination. The current iteration space:

$$0 \leq t,\ t < 1024$$
$$t \leq i,\ i < 1024$$
$$t \leq j,\ j < i$$
$$1 \leq k,\ k < j$$

Since, the goal is to interchange the loops j & k, we'll perform Fourier-Motzkin elimination on j, find the correct iteration space for k (modified constraints), and then we can move j to the innermost location. The inequalities involving j are:

$$t \leq j,\ j < i,\ k < j$$

We are given lower bounds as t, k and upper bound as i, therefore, we'll make two equations, pairing the upper bound with each of the lower bounds:

$$t \leq i,\ k < i$$

Since, the first inequality is already present in the set, the new modified constraints after removing the loop j are:

$$0 \leq t,\ t < 1024$$
$$t \leq i,\ i < 1024$$
$$1 \leq k,\ k < i$$

Adding the constraints for loop j at the end gives us:

$$0 \leq t \quad t < 1024$$
$$t \leq i \quad i < 1024$$
$$1 \leq k \quad k < i$$
$$\max(k+1, t) \leq j \quad j < i \qquad (\because\ t \leq j,\ k < j \Rightarrow max(k+1, t) \leq j)$$

```
int i, j, t, k;
for (t = 0; t < 1024; t++) {
    for (i = t; i < 1024; i++) {
        for (k = 1; k < i; k++) {
            for (j = max(k+1, t); j < i; j++) {
                S(t, i, k, j);
            }
        }
    }
}
```

Problem 4

Part i

For the provided reference code, the iteration space is quite huge (13^9 for the provided input text file), and we are performing redundant floating point operations inside the innermost loop, which

would lead to poor performance. We can try many optimizations like loop permutation, loop tiling, loop unrolling, etc. to enhance the performance, but it really depends on how we optimize the operations on the innermost loop. Since the iteration space for all the loops is symmetric, trying out any loop permutation would lead to the same performance. We have approximately 150 double variables to handle in our GridLoopSearch function, which occupies approximately $150 * 8$ Bytes $= 1200$ Bytes ≈ 1 KB in total. The L1-d cache in majority of the computers nowadays have a capacity much more than this limit, which means the entire set of variables would fit easily in our cache. This means that using techniques like loop tiling wouldn't optimize our performance, since they make use of spatial and temporal locality of variables, but since our variables already fit inside our cache, we'll face only cold misses. Loop unrolling could work since it utilizes instruction-level parallelism (pipelining).

The KD workstation used to evaluate the performances is CSEWS20. The system description is mentioned below:

- CPU max MHz: 4600.00
- CPU min MHz: 800.00
- Cache (L1d): 192 KiB (6 instances) $\Rightarrow$ 32 KiB (1 instance)

From the above cache configuration, we can see that our entire list of variables will fit inside one instance of the cache, so the techniques like loop tiling won't work over here. Applying loop unrolling in the innermost loops also didn't provide any further speedup (on the other hand, it decreased the base performance), but this may be due to some other factors like loop overhead may be negligible in comparison to the system calls, which resulted in poor performance. Two versions of loop unrolling has been submitted as code base (v1 - Loop unrolling 2 times on r_{10}; v2 - Loop unrolling 2 times on r_{10} and r_9), with v2 working the best amongst both taking approximately 405 seconds. The reference code took approximately 380 seconds to complete execution, thus the speedup is 0.94x. In the innermost loop, we have $x_{10} = dd_{28} + r_{10} \cdot dd_{30}$, which is being used to evaluate $q_1, q_2 \dots q_{10}$. Therefore, we can apply forward substitution to remove this redundancy. We didn't get any speedup with this change as well (same performance).

Command for running the code:

```
gcc -O3 -std=c17 -D_POSIX_C_SOURCE=199309L
22111090-prob4-<version>.c -o prob4-<version>.out
./prob4-<version>.out
```

Part ii

The innermost loop (r_{10}), has a lot of redundant calculations in computing the value of $q_1, q_2, \ldots q_{10}$. Let us take q_1 for instance. We are given:

$$q_1 = \text{fabs}\ (c_{11} \cdot x_1 + c_{12} \cdot x_2 + c_{13} \cdot x_3 + c_{14} \cdot x_4 + c_{15} \cdot x_5 + c_{16} \cdot x_6 + c_{17} \cdot x_7 + c_{18} \cdot x_8 + c_{19} \cdot x_9 + c_{110} \cdot x_{10} - d_1\ )$$

Here, the value of $c_{11} \cdot x_1$ is being calculated for every instance of the loop r_{10}, while there is no data dependency between the term and the loop r_{10}. Therefore, we can move this term out of the innermost loop and keep this value in the outermost loop (r_1). Similar arguments can be made for the other terms as well. We just have to ensure that moving these terms outside the innermost loop doesn't lead to any wrong updates in the code (correctness). Thus the values $c_{11}x_1, c_{12}x_2\ , \ldots c_{19}x_9$ are loop invariants for the innermost loop and we can move them to outer loops. This is called Loop Invariant Code Motion (LICM). When we perform the required operations, we see that the code performance becomes significantly better than before. The reference code (v0) took approximately 380 seconds to complete execution, while the optimized code (22111090-prob4-LICM.c) took approximately 200 seconds to complete execution, thus providing almost 1.9x speedup.

Command for running the code:

```
gcc -O3 -std=c17 -D_POSIX_C_SOURCE=199309L
22111090-prob4-LICM.c -o prob4-LICM.out
./prob4-LICM.out
```

End.