Assignment 2

Aniket Sanghi (170110)

3rd October 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

1 Solution 1

```
for i = 1, N-2
    for j = i+1, i+N-2
        A(i, i-j) = A(i, i-j-1) - A(i-1, i-j) + A(i+1, i-j+1)
```

- $A(i, i-j) = ... \ A(i, i-j-1)$ Lets try by assuming a flow dependence
  $I_0 = I_0 + \Delta I \implies \Delta I = 0$
  $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J - 1 \implies \Delta J = -1$
  So, it is an **Anti Dependence (0, 1)** with direction vector **(0, +)**
- $A(i, i-j) = ... \ A(i-1, i-j)$ Lets try by assuming a flow dependence
  $I_0 = I_0 + \Delta I - 1 \implies \Delta I = 1$
  $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J \implies \Delta J = 1$
  So, it is a **Flow Dependence (1, 1)** with direction vector **(+, +)**
- $A(i, i-j) = ... \ A(i+1, i-j+1)$ Lets try by assuming a flow dependence
  $I_0 = I_0 + \Delta I + 1 \implies \Delta I = -1$
  $I_0 - J_0 = I_0 + \Delta I - J_0 - \Delta J + 1 \implies \Delta J = 0$
  So, it is an **Anti Dependence (1, 0)** with direction vector **(+, 0)**
- No output dependence as different indices accessed each time

2 Solution 2

```
int i, j, t, k;
for(t = 0; i < 2048; t++) {
    for(i = t; i < 1024; i++) {
        for(j = t; j < i; j++) {
            for(k = 1; k < j; k++) {
                S(t, i, j, k);
            }
        }
    }
}
```

Given data dependences **(1, 0, -1, 1), (1, -1, 0, 1), (0, 1, 0, -2)**

2.1 (a)

Each of **t, i, j, k** loops are valid to unroll as for unrolling we don't need any special case, we can just unroll and take care of edge cases (to avoid errors).
But only loops **j, k** are valid when we do both - **unroll and jamming**

2.2 (b)

(t, i, j, k) & (t, i, k, j) & (t, j, i, k) are valid permutations of the loops will all dependences valid. Reasoning -

- **i** can't be outer loop as it will make the dependence (1, -1, 0, 1) invalid
- **j** can't be outer loop as it will make the dependence (1, 0, -1, 1) invalid
- **k** can't be outer loop as it will make the dependence (0, 1, 0, -2) invalid
- **k** can't be second outer loop after t as it will make the dependence (0, 1, 0, -2) invalid
- **(t, j, k, i)** will make (0, 1, 0, -2) invalid

2.3 (c)

Following tilings are valid

- 1D tilings of **t, i, j, k** are valid (though it doesn't offer any benefits in performance)
- 2D tilings of **(j, k) & (i, j)** are valid

2.4 (d)

Loop **j & k** are parallelizable as they don't carry any dependences.
In the first 2 dependences t carry the dependence and in the third one i carry the dependence.

2.5 (e)

Following is the tikj form of the code

```
int i, j, t, k;
for(t = 0; i < 2048; t++) {
    for(i = t; i < 1024; i++) {
        for(k = 1; k < i-1; k++) {
            for(j = max(t, k+1); j < i; j++) {
                S(t, i, j, k);
            }
        }
    }
}
```

3 Solution 3

```
#define N 4096
double A[N][N];
int t, i, j;
for(t = 0; t < N-1; t++) {
    for(i = 1; i < N-1; i++) {
        for(j = 1; j < N-1; j++) {
            A[i][j] = 0.2*(A[i-1][j] + A[i][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]);
        }
    }
}
```

3.1 (a)

Following are the dependences in the above loop nest

- $A[i][j] = ... A[i-1][j]$ gives dependence **(*, 1, 0)**
  - **(1, 1, 0), Flow Dependence** with direction vector **(+, +, 0)**
  - **(0, 1, 0), Flow Dependence** with direction vector **(0, +, 0)**
  - **(1, -1, 0), Anti Dependence** with direction vector **(+, -, 0)**
- $A[i][j] = ... A[i][j]$ gives dependence **(*, 1, 0)**
  - **(1, 0, 0), Anti Dependence** with direction vector **(+, 0, 0)**
  - **(0, 0, 0), Anti Dependence** with direction vector **(0, 0, 0)**
  - **(1, 0, 0), Flow Dependence** with direction vector **(+, 0, 0)**
- $A[i][j] = ... A[i+1][j]$ gives dependence **(*, 1, 0)**
  - **(1, 1, 0), Anti Dependence** with direction vector **(+, +, 0)**
  - **(0, 1, 0), Anti Dependence** with direction vector **(0, +, 0)**
  - **(1, -1, 0), Flow Dependence** with direction vector **(+, -, 0)**
- $A[i][j] = ... A[i][j-1]$ gives dependence **(*, 1, 0)**
  - **(1, 0, 1), Flow Dependence** with direction vector **(+, 0, +)**
  - **(0, 0, 1), Flow Dependence** with direction vector **(0, 0, +)**
  - **(1, 0, -1), Anti Dependence** with direction vector **(+, 0, -)**
- $A[i][j] = ... A[i][j+1]$ gives dependence **(*, 1, 0)**
  - **(1, 0, 1), Anti Dependence** with direction vector **(+, 0, +)**
  - **(0, 0, 1), Anti Dependence** with direction vector **(0, 0, +)**
  - **(1, 0, -1), Flow Dependence** with direction vector **(+, 0, -)**
- $A[i][j] =$ gives output dependence as well
  - **(1, 0, 0) Output Dependence** with direction vector **(+, 0 0)**

3.2 (b)

(t, i, j) & **(t, j, i)** are valid permutations where all dependences are valid. Reasoning -

- **i** can't be outer-most loop as then dependences like (1, -1, 0) will become invalid.
- **j** can't be outer-most loop as then dependences like (1, 0, -1) will become invalid.
- **(t, j, i)** is valid as where-ever t is not carrying dependence, none of i / j is negative. So interchange is valid.

3.3 (c)

Each of **t, i, j** loops are valid to unroll as for unrolling we don't need any special case, we can just unroll and take care of edge cases (to avoid errors).
But only loops **i j** are valid when we do both - **unroll and jamming**

3.4 (d)

Following tilings are valid as permuting them don't make dependence invalid.

- 1D tilings of **t, i, j** are valid (though it doesn't offer any benefits in performance)
- 2D tilings of **(i, j)** only is valid (since (t, i, j) and (t, j, i) are both valid permutations as explained in (b))

4 Solution 4

How to Run

```
g++ -O3 problem4.cpp -lpthread
./a.out <number_of_customers>
```

where replace $< number\_of\_customers >$ with the count of customers, value in range [0 24]

Few Internal Details Highlights

- Each customer thread is given monotonically increasing token non-deterministically
- Each customer thread is enqueued non-deterministically
- Teller informs about the complete of transaction to customer, then only the customer leaves
- Customer just before leaving will inform the next person in queue that teller is free now
- Rest all things are briefly commented in the code to ease the understanding

5 Solution 5

How to Run

```
g++ problem5.cpp -lpthread
./a.out <number_of_threads> <blocking_size>
```

$< blocking\_size >= 4 \implies 4 \times 4$ *sized blocks* Make sure to keep blocking_size >= 16.

Tested various degrees of Unrolling in the optimised version 2, 4, 8, 16, 32. To test other unrolling factors you can use the other code in the directory which can be run as below

```
g++ problem5b.cpp -lpthread -o p5
time ./p5 <number_of_threads> <blocking_size> <unrolling_factor>
```

Make sure to keep $blocking\_size > unrolling\_factor$ and unrolling factor can be 1, 4, 8 only.

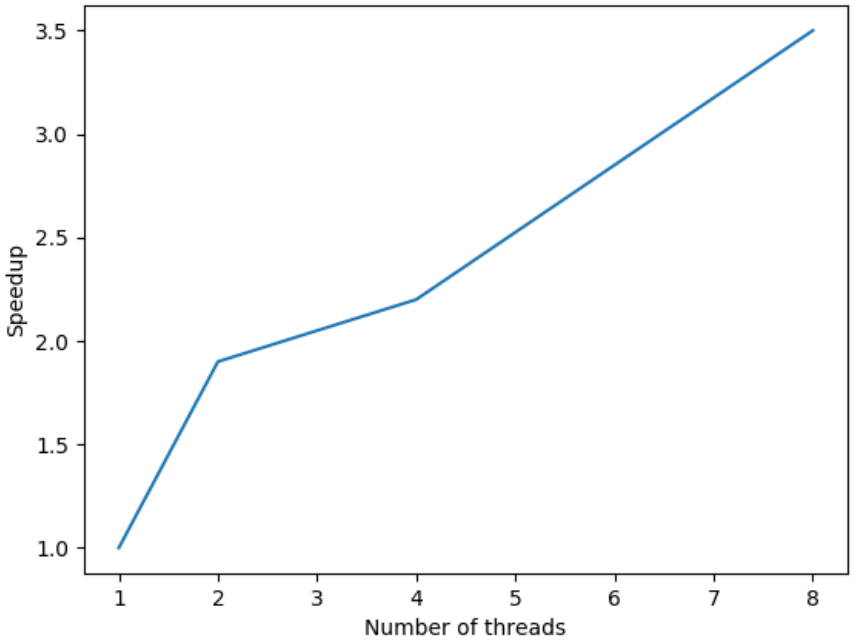
5.1 (a)

Tested on System with configuration

- Model - Intel(R) Core(TM) i7-8700
- CPU @ 3.20GHz
- Number of Cores - 6
- Number of threads per core - 2

The speedup for 2 threads is 2x but the speed up for 4 and 8 threads didn't scale similarly. Possible reasons -

1. Busy System, the machine I used is a CSE server used by many people
2. Context Switching overheads
3. Varying speed/performance of different cores, slowest core being the bottleneck



5.2 (b)

The steps to run and test various variants are mentioned above.

Best performing variant for the architecture I tested on was the one with *unrolling by a factor of 16* and *blocking by a factor of 32*

Sequential got a boost of about 5.5x with blocking and unrolling as compared to 3.5x, got with coarse parallelisation with 8 threads suggesting that blocking significantly can increase a lot of performance even when compared with coarse parallelisation.

System Cache Model

- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 12288K

The total size needed for all 3 arrays for blocks of size 32×32 is $3 * 2^5 * 2^5 * 2^3 B = 24K$ which is quite close to the 32K L1 cache availability. Actually 64 block size is expected to give better performance considering that total size of L1 cache is 64K but due to system being busy with various other programs as well the 32 block size variant gave best performance (64 block size variant gave almost similar performance).

Regarding unrolling, the further increase of unrolling factor from 16 to 64 doesn't seem to give benefits. It mostly decreased performance. Most plausible reason being the resistor overload and increased number of statements which led to performance degradation.

The variant which combined parallelisation benefits with blocking and unrolling gave around 24x improvement which is quite a lot of improvement. The program ran in around 40seconds as compared to 15-18mins of the sequential version.