Assignment 9

Task 1

Snippet 1

```
    for (int i=0; i < n-1; i++) {
1           tmp = (y[i] + x[i+1]) / 7;
2       x[i] = tmp;
    }
```

Since we read x[i+1] in line one and write the result into x[i] in line two, there is an antidependence across the loop iteration bettwen line one and line 2.
($1\delta^{-1}2$ with distance (-1) and direction (>)) .

Parallel Snippet 1

In order to parallelize snipped one, we introduced a fresh variable.

```
#pragma omp parallel for
for (int i = 0; i<len-1; i++){
  a2[i] = (y[i] + x[i+1])/7
}
```

Snippet 2

```
    for (int i=0; i<n; i++){
1    a = (x[i] + y[i]) / (i+1);
2    z[i] = a;
  }
    f = sqrt(a + k);
```

In this snipped, there is obvously a dependence between 1 and 2, but there are **no** dependences across loop iterations.

Parallel Snippet 2

To parallelize this snipped, the loop did not need any changes. The only thing to change was the parameter a of the sqrt function (since the value of a is undefined at the end of the loop).

```
#pragma omp parallel private(a)
for(int i = 0; i<n; i++){
    a = (x[i] + y[i])/(i+1);
  z[i] = a;
}
f = sqrt(z[n-1] + k);
```

Snippet 3

```
1   for (int i=0; i < n; i++) {
    x[i] = y[i] * 2 + b * i;
    }

2   for (int i=0; i < n; i++) {
    y[i] = x[i]  + a / (i+1);
    }
```

There is no data dependence per for loop. Therefore, loop 2 depends on loop1. This can be parallelized very easy though.

Parallel snippet 3

```
#pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i < len; i++) {
        x[i] = y[i] * 2 + b*i;
    }

    #pragma omp for
    for (int i=0; i < len; i++) {
        y[i] = x[i] + a/(i+1);
    }
    }
```

Task 2

Snippet 1

```
    double factor = 1;

    for (int i=0; i < ARR_LEN; i++) {
        x[i] = factor * y[i];
        factor = factor / 2;
    }
```

Data depencies: factor causes data depencies, because it is used to calculate "the new version of itself" and to calculate the result for array x at the current position.

Parallel version: removed unnecessary calculations and parallelized the for loop with `pragma omp for`

Parallel Snippet 1

Calculating the factor directly with pow(2,i).

```
omp_set_num_threads(8);
    #pragma omp parallel
        #pragma omp for
        for (int i=0; i < len; i++) {
            x[i] = 1 / pow(2, i) * y[i];
        }
```

The parallel version is slower than the serial version. for array size 30 (there is the last value about 1e-7):

- serial: 0.0000015 s
- parallel(8 threads): 0.0080 s

Also for bigger array sizes (like 1000) the parallel version doesn't have a better performance compared to the serial version.

Snippet 2

```
    for (int i = 1; i<len; i++) {
    x[i] = (x[i] + y[i-1]) / 2;
    y[i] = y[i] + z[i] * 3;
}
```

In this snipped, there is a dependence between x and y.

Parallel Snippet 2

To parallelize this snipped, we put the creation of y in a seperate loop and then we can calculate x in another loop, where y is initialized for sure.

```
omp_set_num_threads(8);
    #pragma omp parallel
        #pragma omp for
        for (int i=0; i < len; i++) {
            y[i] = y[i] + z[i] * 3;
        }

        #pragma omp for
        for (int i=0; i < len; i++) {
            x[i] = (x[i] + y[i-1]) / 2;
        }
```

But still for an array size of 100000 the serial version is way faster:

- serial: 0.0004383 s
- parallel: 0.0009 s

The point, where the execution time of the serial version and the parallel version are almost equal, is at array size 100.000.000 .

Snippet 3

```
x[0] = x[0] + 5 * y[0];
for (int i = 1; i<n; i++) {
    x[i] = x[i] + 5 * y[i];
    if ( twice ) {
        x[i-1] = 2 * x[i-1]
    }
}
```

This snippet shows a data dependency because y is needed to produce x. But because of the if-statement, we possibly got a control dependency, iff twice is associated to x.

Parallel snippet 3

We parallelize it with **Loop Unswitching** , so every calculation is in a single loop in this case and the if is draged outside.

```
omp_set_num_threads(8);
#pragma omp parallel
    x[0] = x[0] + 5 * y[0];

    if ( 1 ) {
        #pragma omp for
        for (int i = 1; i < len; i++) {
            x[i] = x[i] + 5 * y[i];
        }

        #pragma omp for
        for (int i = 1; i < len; i++) {
            x[i-1] = 2 * x[i-1];
        }
    }else{
        #pragma omp for
```

```
        for (int i = 1; i < len; i++) {
            x[i] = x[i] + 5 * y[i];
        }
    }
```

For array size of 100.000.000 again, and set twice to 1, so it is always true (edge case) we got these execution times for 8 threads:

- sequential: 0.093604 s
- parallel: 0.082397 s

And with twice = 0:

- sequential: 0.082208 s
- parallel: 0.038709 s

Task 3

Original snippet

```
for (int i = 0; i < 4; ++i) {
    for (int j = 1; j < 4; ++j) {
        a[i + 2][j - 1] = b * a[i][j] + 4;
    }
}
```

i	j	source	sink	distance vector	direction vector
0	1	a[0][1]	a[2][0]	(2,-1)	(<,>)
0	2	a[0][2]	a[2][1]	(2,-1)	(<,>)
0	3	a[0][3]	a[2][2]	(2,-1)	(<,>)
1	1	a[1][1]	a[3][0]	(2,-1)	(<,>)
1	2	a[1][2]	a[3][1]	(2,-1)	(<,>)
1	3	a[1][3]	a[3][2]	(2,-1)	(<,>)
2	1	a[2][1]	a[4][0]	(2,-1)	(<,>)
2	2	a[2][2]	a[4][1]	(2,-1)	(<,>)
2	3	a[2][3]	a[4][2]	(2,-1)	(<,>)
3	1	a[3][1]	a[5][0]	(2,-1)	(<,>)
3	2	a[3][2]	a[5][1]	(2,-1)	(<,>)
3	3	a[3][3]	a[5][2]	(2,-1)	(<,>)

The leftmost non-"=" component of the direction vector defines the dependency. In this case it is "<", which means there is a true dependency, this means the sink occurs before the source. The distance vector (and therefore the direction vector) stays the same through all iterations, since the distance vector originates from the offsets in the indices of the 2D array.

Parallelized snippet

Loop splitting

```
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp task
            {
                for(int i = 0; i < 2; ++i) {
                    for(int j = 1; j < 4; ++j) {
                        a[i + 2][j - 1] = b * a[i][j] + 4;
                    }
                }
            }

#pragma omp task
            {
                for(int i = 2; i < 4; ++i) {
                    for(int j = 1; j < 4; ++j) {
                        a[i + 2][j - 1] = b * a[i][j] + 4;
                    }
                }
            }
        }
    }
```

The code can be parallelized with loop splitting into 2 loops, which themselfes are not independent on themselves.