Student: Rusu Florin

Lab 2: Image Reconstruction

Exercise 1: **Modify the sequential code provided so that it prints out the time taken by the function encaja, which is the one in charge of trying to reconstruct the original image.**

While to print out the time taken by the function encaja we need to add the following code:

```
double t1, t2;   // declare 2 variables of type double
t1 = omp_get_wtime();  //get the time until the execution of the function
encaja(&ima);  //call function
t2 = omp_get_wtime(); // get the time after the execution
 printf("Time: %f", t2 - t1);
```

Exercise 2: **Parallelize the function encaja. For each of its three loops (separately), indicate if it can be parallelized or not, and in case it can be, perform the parallelization. Is it advisable to modify OpenMP's default schedule option for any of the loops? Make the programs in this exercise print out the number of threads running, and the time taken by the function encaja.**

First loop: for (i = 0; i < n; i++)

This loop can´t be parallelized because implies many dependencies in his running.

Second loop:

```
#pragma omp parallel for private(x, distancia)
    for (j = i + 1; j < ima->alto; j++) { .. }
```

The second loop can be parallelized so, as we see the x and dinstancia will be private in order to obtain a higher speed-up. And there are a needing to parallelize this if with critical as shown below.

```
  if (distancia < distancia_minima)
      {
#pragma omp critical
          if (distancia < distancia_minima)
              {
               distancia_minima = distancia;
               linea_minima = j;    }
```

Third loop:

#pragma omp parallel for reduction(+:distancia)

for (x = 0; x < ima->ancho ; x+=0)

In this case, it is needed to parallelize in the following order, so the distancia will be reduction, because it is initialized with = 0 before the loop.

While to printout the number of threads with a minimum loosing of time, we will implement this piece of code:

```
int nthread;

#pragma omp parallel
{
        nthread = omp_get_num_threads();  //get number of threads
 }

printf("Number of threads = %d\n", nthread); // Display the number of threads!!!
```

The result of parallelizing loop p-j and p-x, including time & number of threads

Threads	P-j	P-x
1	131.6	130.3
2	66.01	71.05
3	44.12	51.23
4	34.22	40.33
5	28.44	37.31
6	23.89	33.25
7	21.05	31.57
8	18.31	30.23
9	16.63	33.24
10	15.19	33.69
11	13.95	33.69
12	13.18	33.17
13	12.36	32.84
14	11.49	33.1
15	10.79	35.63
16	9.99	34.21

Fig.1 – Data table

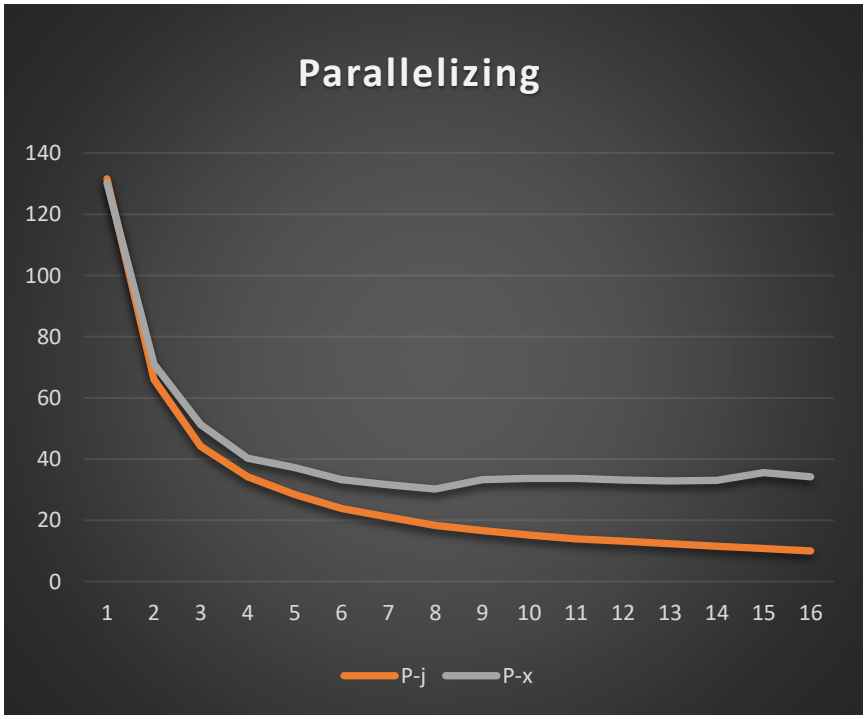

Fig.2 – Graphic result

Student: Rusu Florin

Speed-up results

Threads	Speed-up(j)	Speed-up(x)
2	1.99	1.83
4	3.84	3.25
8	7.18	4.31
16	13.29	3.81

Fig.3 – Speed-up data

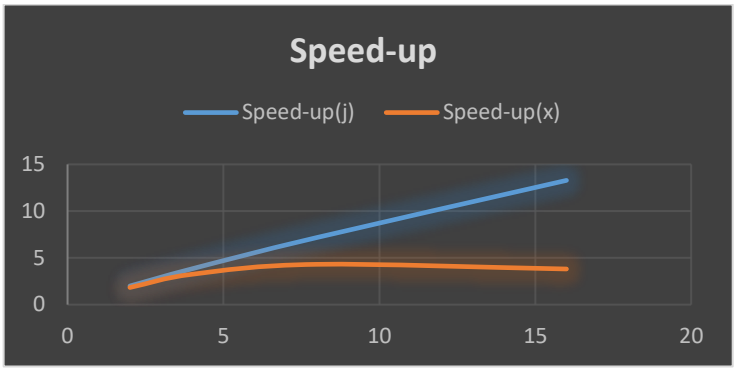

Fig.4 – Speed-up graphic

Efficiency

Threads	Efficiency(j)	Efficiency(x)
2	0.99	0.91
4	0.96	0.81
8	0.89	0.53
16	0.83	0.23

Fig.5 – Efficiency data

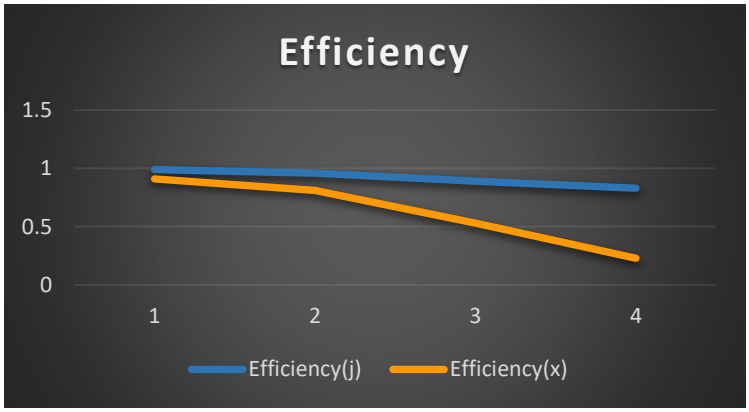

Fig.6 – Efficiency graphic

Conclusion:

As we see upper there, it is better to parallelize the outer loop, because the inner loop performs a little piece of all the work (only the sum of distance). Comparing with the outer that includes more operations. Therefore, it gives a higher speed-up and it is more efficient than the inner. The value of 89% efficiency with 8 threads confirms this affirmation, comparing with 53% of the other loop.

Exercise 3: Create a new sequential version with a modification of loop x so that the loop ends as soon as the partial sum of the current distance reaches a value greater than the minimum distance computed so far (therefore the line can be discarded right away, without finishing the computation of its distance).

While to reach the purposed goal, we need to implement the following code:

```
 for (x = id; x < ima->ancho ; x+=nt)
        {
                distancia += diferencia(&A(x, i), &A(x, j));
                        if (distancia >distancia_minima) break;
        }
```

OR

```
 for (x = id; x < (ima->ancho) && (!distancia>distance_minima); x+=nt)
```

Exercise 4: Starting from the code developed in exercise 3, parallelize (separately) each of the loops that can be parallelized. Obtain the execution time and performance indexes for the best parallel version implemented in this exercise (comparing it with the new sequential version).

Parallel version for loop x

```
#pragma omp parallel private(x) reduction(+:distancia)
{
        int id = omp_get_thread_num();
        int nt = omp_get_num_threads();
  for (x = id; x < (ima->ancho) && (!distancia>distancia_minima) ; x+=nt) {
                distancia += diferencia(&A(x, i), &A(x, j));
        }
}
```

While to check if the new parallelization is working properly, it is mandatory to compare the output files. To do this we need to use command cmp <file_1> <file_2>.

After running of the cmp command, we obtain a good result, no one of the pixels differs, so the construction of the photos are the same. Now, we need to input the data in the table while to display the graphic to see the actually comparison between the parallelized loops.(See fig.7)

Comparison of new parallelization of x with the previous versions

Threads	P-j	P-x	New-x
1	131.6	130.3	20.36
2	66.01	71.05	22.73
3	44.12	51.23	23.04
4	34.22	40.33	22.4
5	28.44	37.31	24.01
6	23.89	33.25	24.94
7	21.05	31.57	24.82
8	18.31	30.23	25.6
9	16.63	33.24	28.65
10	15.19	33.69	30.28
11	13.95	33.69	32.05
12	13.18	33.17	31.88
13	12.36	32.84	33.61
14	11.49	33.1	36.63
15	10.79	35.63	36.52
16	9.99	34.21	36.89

Fig.7 – Result in seconds including new-x.

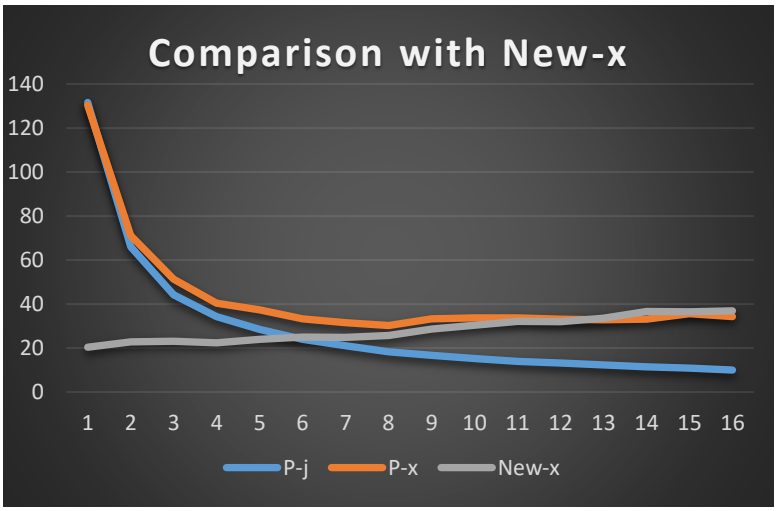

Fig.8 – Graphic representation of comparison.

While we use thread ID and NUM we can distribute the workload in a better convenient way, and during the running, we can observe a gap. This is the overhead, caused by the big number of threads. Let's try to parallelize the j loop. (fig.9)

Parallel version for a loop j (the fastest)

```
#pragma omp parallel for private(x, distancia)
   for (j = i + 1; j < ima->alto; j++)
   {
     distancia = 0;
     for (x = 0; x < (ima->ancho) && distancia < distancia_minima; x++)
     {
       distancia += diferencia(&A(x, i), &A(x, j));
            //if (distancia > distancia_minima) break;
     }
   }
   if (distancia < distancia_minima) {
```

#pragma omp critical

```
     if (distancia < distancia_minima)
```

Results for best parallel version

Threads	Loop p-j
1	18.52
2	9.41
4	5.07
8	3.13
16	1.99

Fig.9 – Parallelization of loop(j)

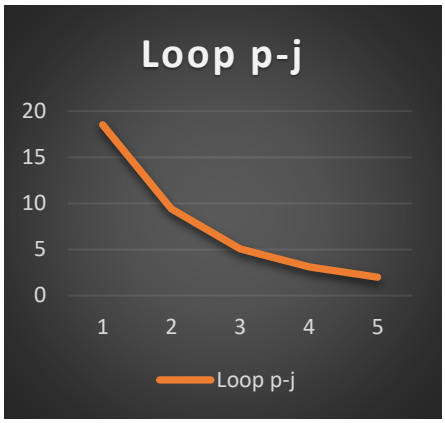

Fig.10 – Results of loop(j)

In this example we can really observe the power of parallelization and the speed that we can achieve during running.

While to work in a faster way with Kahan, we can use the following script code:

```
#!/bin/sh
#PBS –l nodes=1,walltime=00:10:00
#PBS –q cpa
#PBA –d .
gcc –fopenmp –o encaja encaja.c –lm
i=1;
while [ $i –le 16 ]; do
OMP_NUM_THREADS=$i ./encaja –t
echo $i;
i=$((i+1));
done
```