Bedir Esen, 1520350910
Mustafa Baran Ercan, 2881055520

IMAGE FILTERING
EDGE DETECTION

threads (t) / Image Size	1	2	3	4	8	16	20
10MB	3.209s	2.163s	1.763s	2.183s	2.165s	2.241s	2,302s
50MB	12.592s	10.692s	10.254s	9.61s	8.881s	8.01s	8.973s

Filter That We Use

```
void apply_filter(int *img_data, int *img_result, int width, int height, int local_start, int local_end, int img_colors)
{
    // 2. Find the horizontal edges of the image
    // 2.1 Define the edge detection filter.
    // The filter used here is a 3x3 Sobel filter
    int filter[3][3] = {{-1, 0, 1},
                        {-2, 0, 2},
                        {-1, 0, 1}};

    // 2.2 Apply the edge detection filter to the image
    for (int y = local_start; y < local_end; y++) {  // Iterate over the height, skip border pixels
        for (int x = 1; x < width - 1; x++) {   // Iterate over the width, skip border pixels
            int sum = 0;    // Sum to store the convolution result. Higher values indicate edges. Lower values indicate smooth areas.
            for (int filter_y = 0; filter_y < 3; filter_y++) {  // We are applying a 3x3 filter
                for (int filter_x = 0; filter_x < 3; filter_x++) {
                    int image_x = x + filter_x - 1;
                    int image_y = y + filter_y - 1;
                    int image_value = img_data[image_y * width + image_x];
                    sum += image_value * filter[filter_y][filter_x];
                }
            }
            // If the sum is negative, set it to 0. If it is positive, set it to max pixel value.
            sum = sum < 0 ? 0 : sum;
            sum = sum > img_colors ? img_colors : sum;
            img_result[y * width + x] = sum;
        }
    }
}
```

Our filter iterates on every pixel of the image, and by multiplying the values in the filter it creates a new pixel. So that, at the end an image with edges extracted is created.

MPI Broadcast

```
// Broadcast the image data to all the processes
MPI_Bcast(&width, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&height, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&img_colors, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Allocate memory for the image in all other processes
if (my_rank != 0) {
    img_data = (int*)malloc(width * height * sizeof(int));
    img_result = (int*)malloc(width * height * sizeof(int));
}
MPI_Barrier(MPI_COMM_WORLD);
// Broadcast the image data to all the processes
MPI_Bcast(img_data, width * height, MPI_INT, 0, MPI_COMM_WORLD);
```

Here, our master node shares the data to slaves using MPI_Bcast method.

Times, Efficiency and Speedup

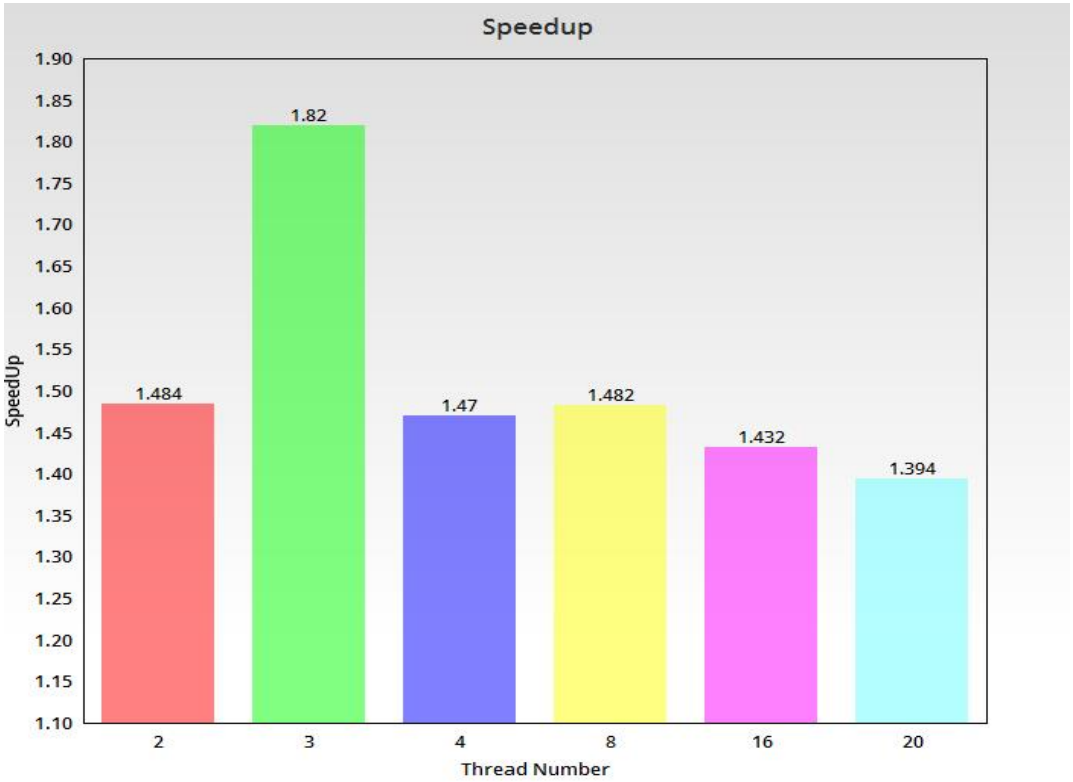
n = 10mb t = comparison between 1, 2, 3, 4, 8, 16, 20 threads

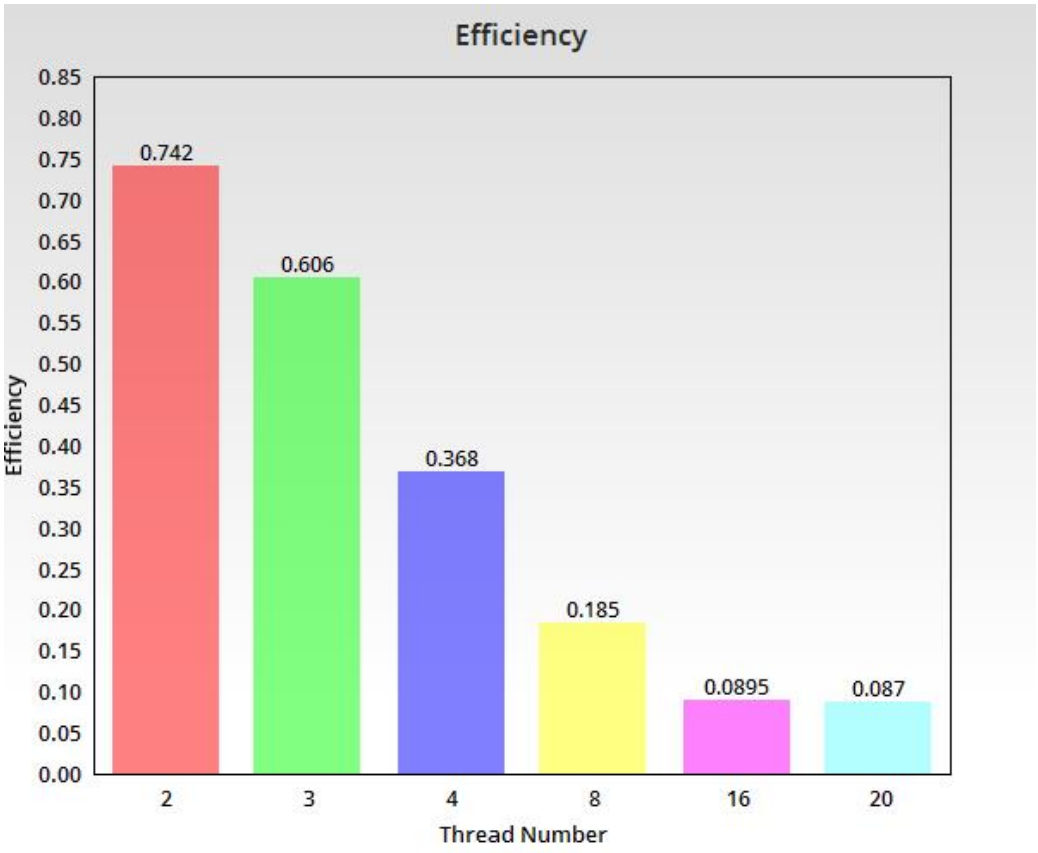
T_{serial} = 3.209s
$T_{t=2}$ = 2.163s
$T_{t=3}$ = 1.763s
$T_{t=4}$ = 2.183s
$T_{t=8}$ = 2.165s
$T_{t=16}$ = 2.241s
$T_{t=20}$ = 2.302s

Speedup (S) = $T_{serial} / T_{parallel}$
S_2 = 1,484
S_3 = 1,820
S_4 = 1,47
S_8 = 1,482
S_{16} = 1,432
S_{20} = 1,394

Efficiency E = S / p

E_2 = 0.742
E_3 = 0.606
E_4 = 0.368
E_8 = 0.185
E_{16} = 0.0895
E_{20} = 0.087


Speedup
SpeedUp
1.90
1.85
1.80
1.75
1.70
1.65
1.60
1.55
1.50
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
1.484
1.82
1.47
1.482
1.432
1.394
2
3
4
8
16
20
Thread Number


Efficiency
Efficiency
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.742
0.606
0.368
0.185
0.0895
0.087
2
3
4
8
16
20
Thread Number

n = 50mb t = comparison between 1, 2, 3, 4, 8, 16, 20 threads

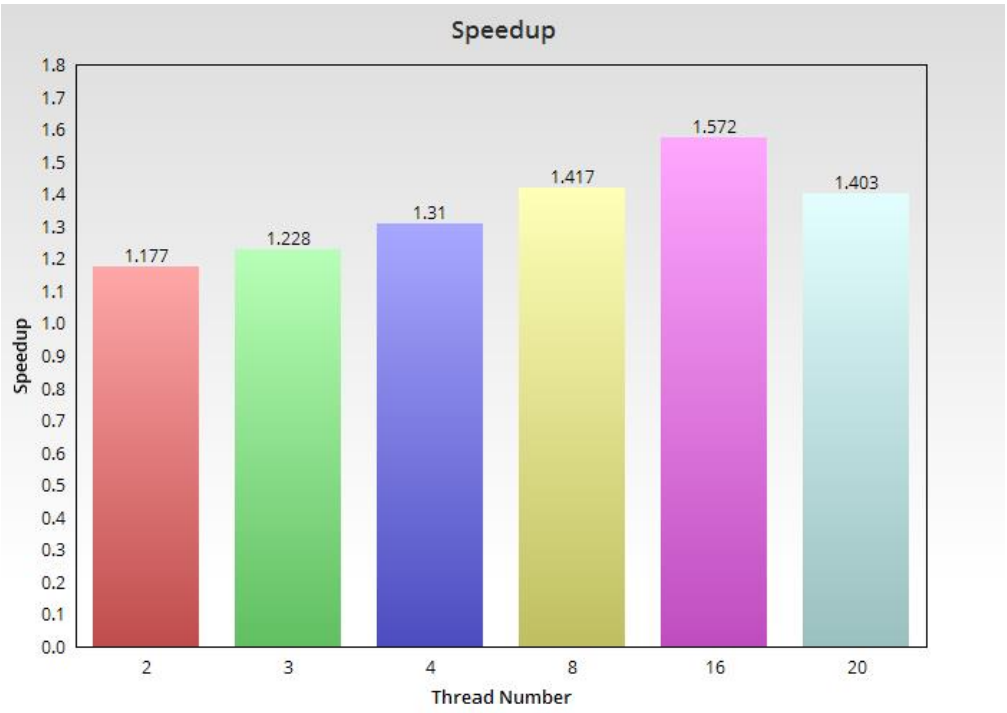
T_{serial} = 12.592s
$T_{t=2}$ = 10.692s
$T_{t=3}$ = 10.254s
$T_{t=4}$ = 9.61s
$T_{t=8}$ = 8.881s
$T_{t=16}$ = 8.01s
$T_{t=20}$ = 8.973s

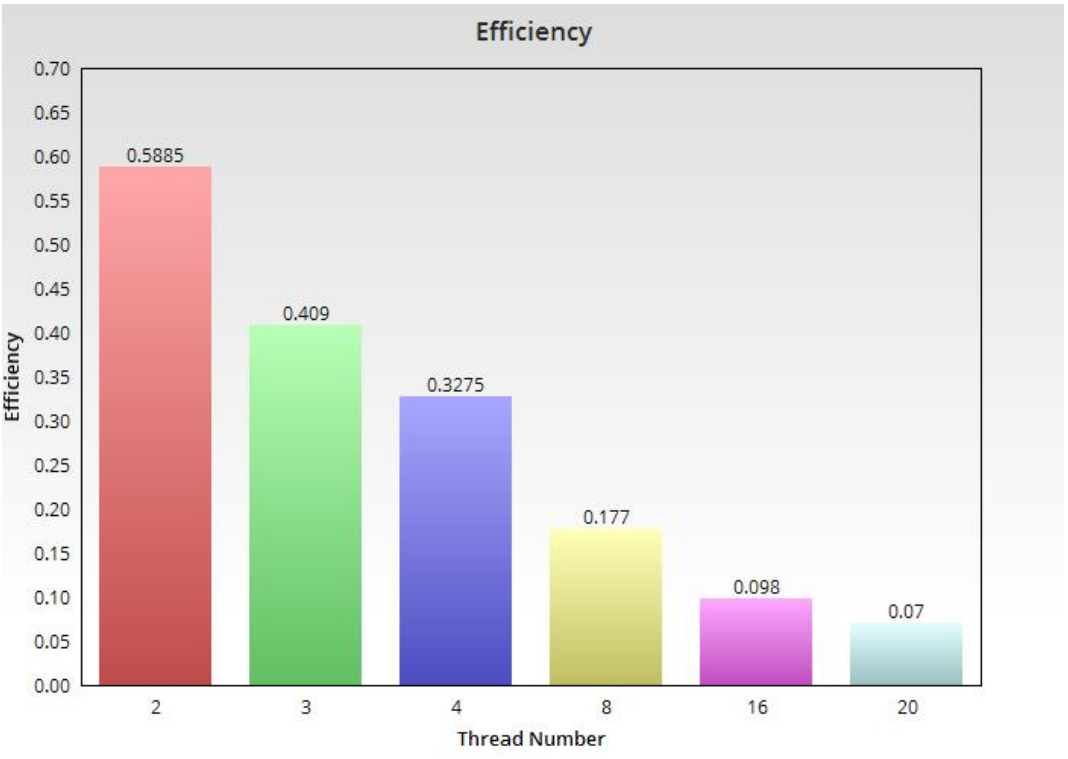
Speedup (S) = $T_{serial} / T_{parallel}$

S_2 = 1.177
S_3 = 1.228
S_4 = 1.31
S_8 = 1.417
S_{16} = 1.572
S_{20} = 1.403

Efficiency E = S / p

E_2 = 0.5885
E_3 = 0.409
E_4 = 0.3275
E_8 = 0.177
E_{16} = 0.098
E_{20} = 0.07


Speedup
Speedup
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1.177
1.228
1.31
1.417
1.572
1.403
2
3
4
8
16
20
Thread Number


Efficiency
Efficiency
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.5885
0.409
0.3275
0.177
0.098
0.07
2
3
4
8
16
20
Thread Number

Conclusion

To conclude, we observed that increasing the image size, makes our MPI parallelized filtering code faster. The efficiency in the greater size decreases but when concerning the speedup, we get better results. We think that the main reason of unbalanced situation in the speedup in the smaller sizes is nature of the MPI_Bcast method. Master sends so that shares the data to slave processes. This time-consuming situation creates an unbalanced situation in small data sizes.