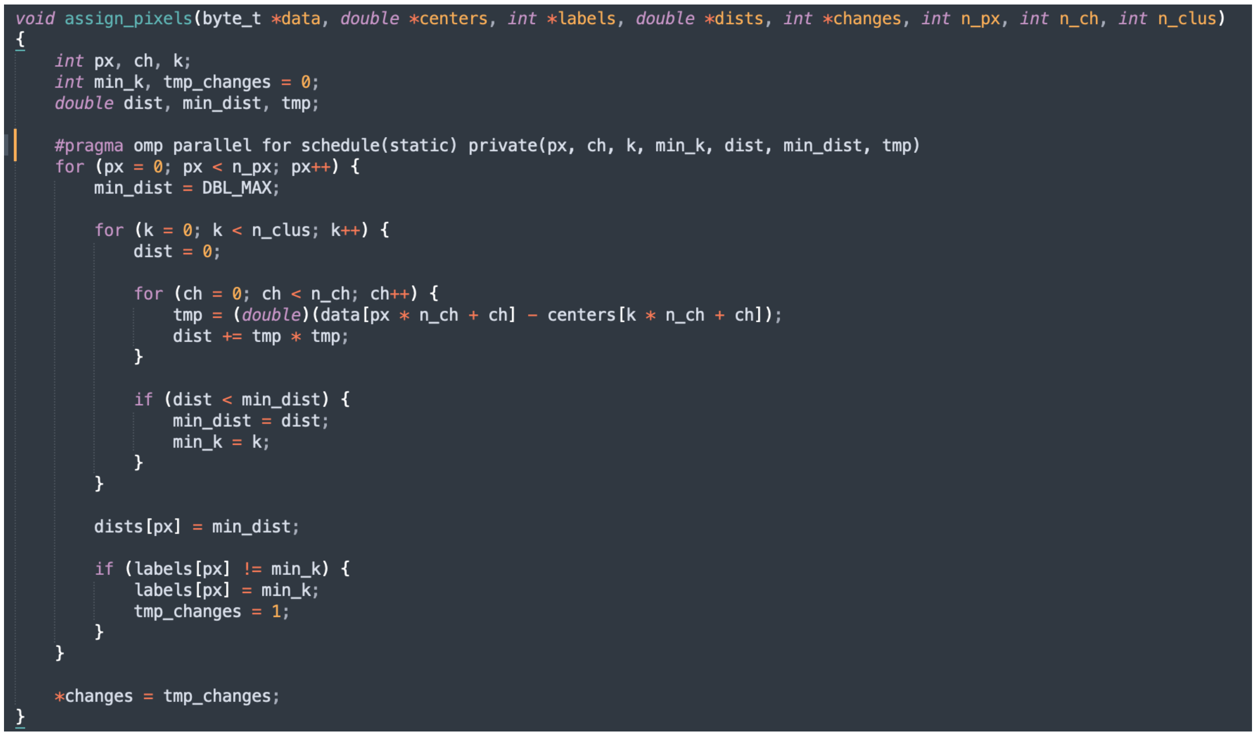
Abelardo Riojas
ISC 4933 Spring 2022
Prof. Wang

OpenMP Good Performance Report

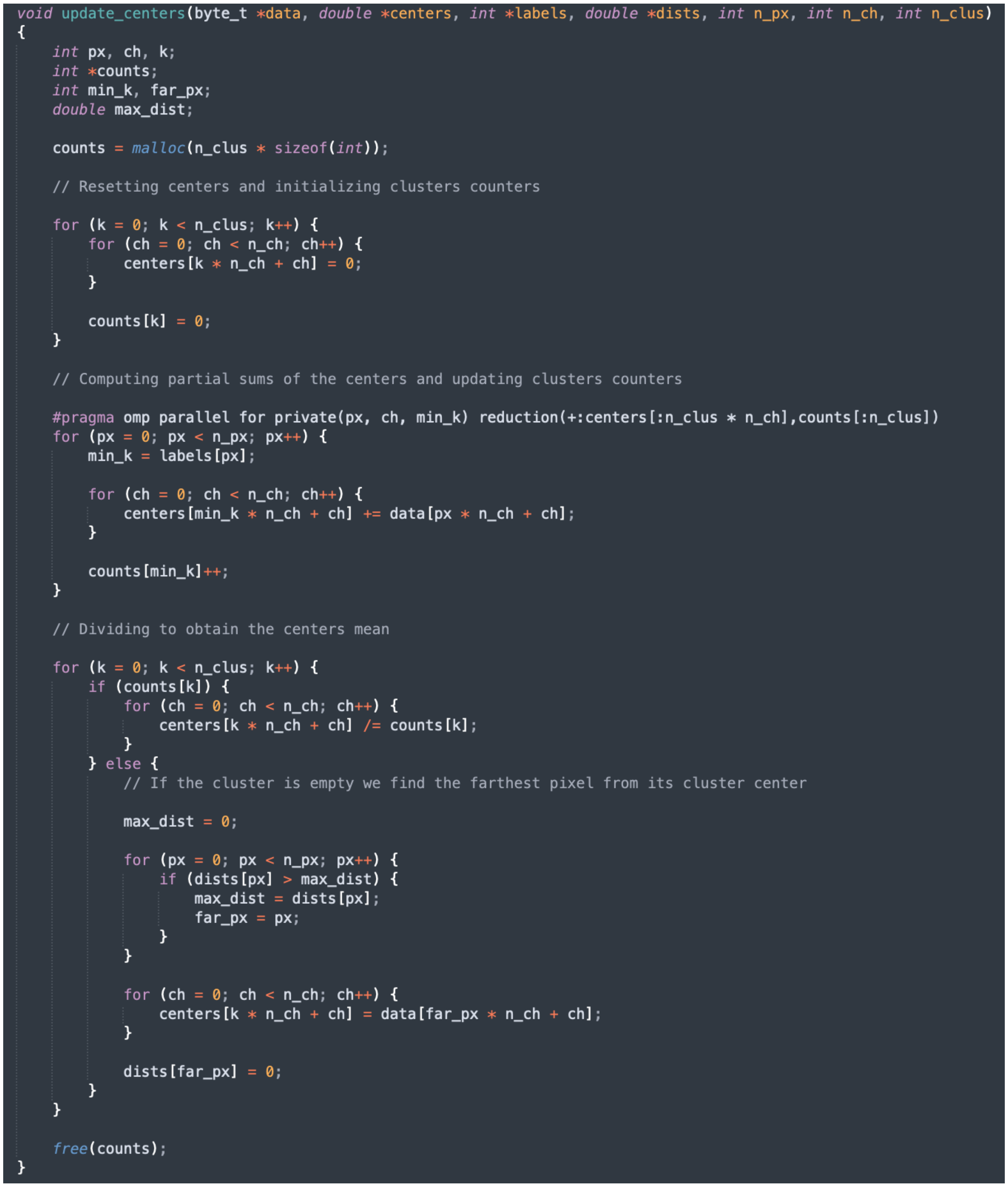
There are four OpenMP pragma directives in my source code: let's take a look at all four, and see if they pass the test.

The first one is in assign_pixels(). This one parallelizes the task of assigning a pixel to its closest center.

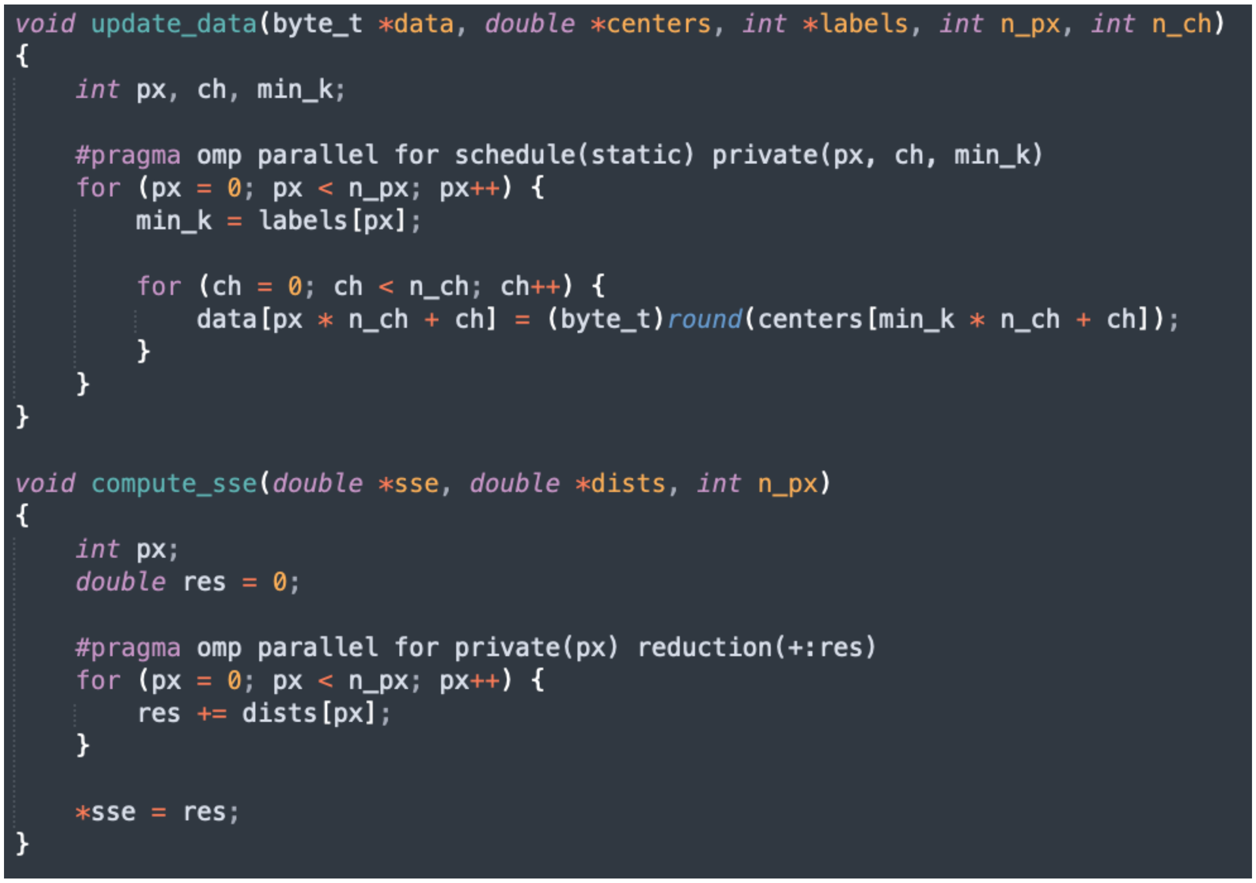


```c
void assign_pixels(byte_t *data, double *centers, int *labels, double *dists, int *changes, int n_px, int n_ch, int n_clus)
{
    int px, ch, k;
    int min_k, tmp_changes = 0;
    double dist, min_dist, tmp;

    #pragma omp parallel for schedule(static) private(px, ch, k, min_k, dist, min_dist, tmp)
    for (px = 0; px < n_px; px++) {
        min_dist = DBL_MAX;

        for (k = 0; k < n_clus; k++) {
            dist = 0;

            for (ch = 0; ch < n_ch; ch++) {
                tmp = (double)(data[px * n_ch + ch] - centers[k * n_ch + ch]);
                dist += tmp * tmp;
            }

            if (dist < min_dist) {
                min_dist = dist;
                min_k = k;
            }
        }

        dists[px] = min_dist;

        if (labels[px] != min_k) {
            labels[px] = min_k;
            tmp_changes = 1;
        }
    }

    *changes = tmp_changes;
}
```

The next one is in update_centers(). This parallel section does the task of computing the partial sums of the average color of each cluster. We even specify a reduction clause on an array which gives each thread a copy of centers and counts, computes the sum, and adds them to the original value of clusters and counts (initialized to zero).



```c
void update_centers(byte_t *data, double *centers, int *labels, double *dists, int n_px, int n_ch, int n_clus)
{
    int px, ch, k;
    int *counts;
    int min_k, far_px;
    double max_dist;

    counts = malloc(n_clus * sizeof(int));

    // Resetting centers and initializing clusters counters

    for (k = 0; k < n_clus; k++) {
        for (ch = 0; ch < n_ch; ch++) {
            centers[k * n_ch + ch] = 0;
        }

        counts[k] = 0;
    }

    // Computing partial sums of the centers and updating clusters counters

    #pragma omp parallel for private(px, ch, min_k) reduction(+:centers[:n_clus * n_ch],counts[:n_clus])
    for (px = 0; px < n_px; px++) {
        min_k = labels[px];

        for (ch = 0; ch < n_ch; ch++) {
            centers[min_k * n_ch + ch] += data[px * n_ch + ch];
        }

        counts[min_k]++;
    }

    // Dividing to obtain the centers mean

    for (k = 0; k < n_clus; k++) {
        if (counts[k]) {
            for (ch = 0; ch < n_ch; ch++) {
                centers[k * n_ch + ch] /= counts[k];
            }
        } else {
            // If the cluster is empty we find the farthest pixel from its cluster center

            max_dist = 0;

            for (px = 0; px < n_px; px++) {
                if (dists[px] > max_dist) {
                    max_dist = dists[px];
                    far_px = px;
                }
            }

            for (ch = 0; ch < n_ch; ch++) {
                centers[k * n_ch + ch] = data[far_px * n_ch + ch];
            }

            dists[far_px] = 0;
        }
    }

    free(counts);
}
```



```c
void update_data(byte_t *data, double *centers, int *labels, int n_px, int n_ch)
{
    int px, ch, min_k;

    #pragma omp parallel for schedule(static) private(px, ch, min_k)
    for (px = 0; px < n_px; px++) {
        min_k = labels[px];

        for (ch = 0; ch < n_ch; ch++) {
            data[px * n_ch + ch] = (byte_t)round(centers[min_k * n_ch + ch]);
        }
    }
}

void compute_sse(double *sse, double *dists, int n_px)
{
    int px;
    double res = 0;

    #pragma omp parallel for private(px) reduction(+:res)
    for (px = 0; px < n_px; px++) {
        res += dists[px];
    }

    *sse = res;
}
```

The next two are for after we finish K means clustering. update_data() writes the cluster values to the final segmented photo, by pixel and then by channel. Then compute SSE just gives us some error between the original photo and the segmented photo using the dists array we calculated in assign_pixels.

Does it pass the test?

1. False sharing: Does your code have a potential false sharing problem?
    False sharing arises when several threads maintain their respective partial result in a vector indexed by the thread rank. We do not do that anywhere here, therefore we're good! Labels are indexed by pixel.
2. Cache locality
    I do access variable elements in the same order as they are stored in memory. In all the arrays we access: data, centers, dists, and labels, they are all accessed in the order they are stored in memory.

3. Coarse grain model

    Even though I did not use parallel sections, I feel like I can call this a coarse grain model. The tasks are split up into different functions which are all done by an independent team of threads. There is a parallel ‘section’ in kmeans_segm_omp() where assign_pixels() and update_centers() are called repeatedly.

4. Loop fusion, loop fission, loop collapse

    Loop fusion problems: don’t see any as we can’t combine two different loops to do the same task.

    Loop fission problems: also don’t see any as we cannot break up tasks into two different loops using this scheme.

    Collapse: I do not see an opportunity to use collapse due to the fact that we do not have perfectly nested loops. If we were looping through the rows and columns of the image, maybe. Every other nested loop needs to get a label for its closest cluster before cycling through the channels.

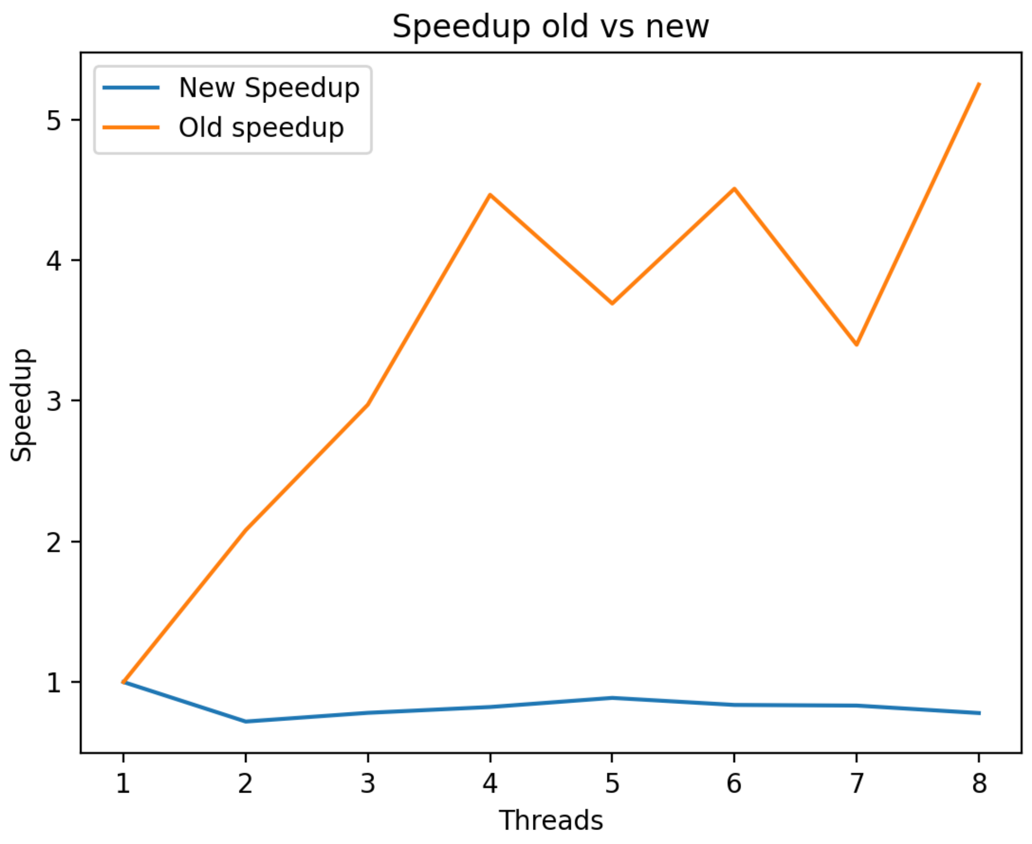
5. Use dynamic for load balancing

    Sure, let’s use dynamic. It might give a slight speedup.

6. Minimize barriers

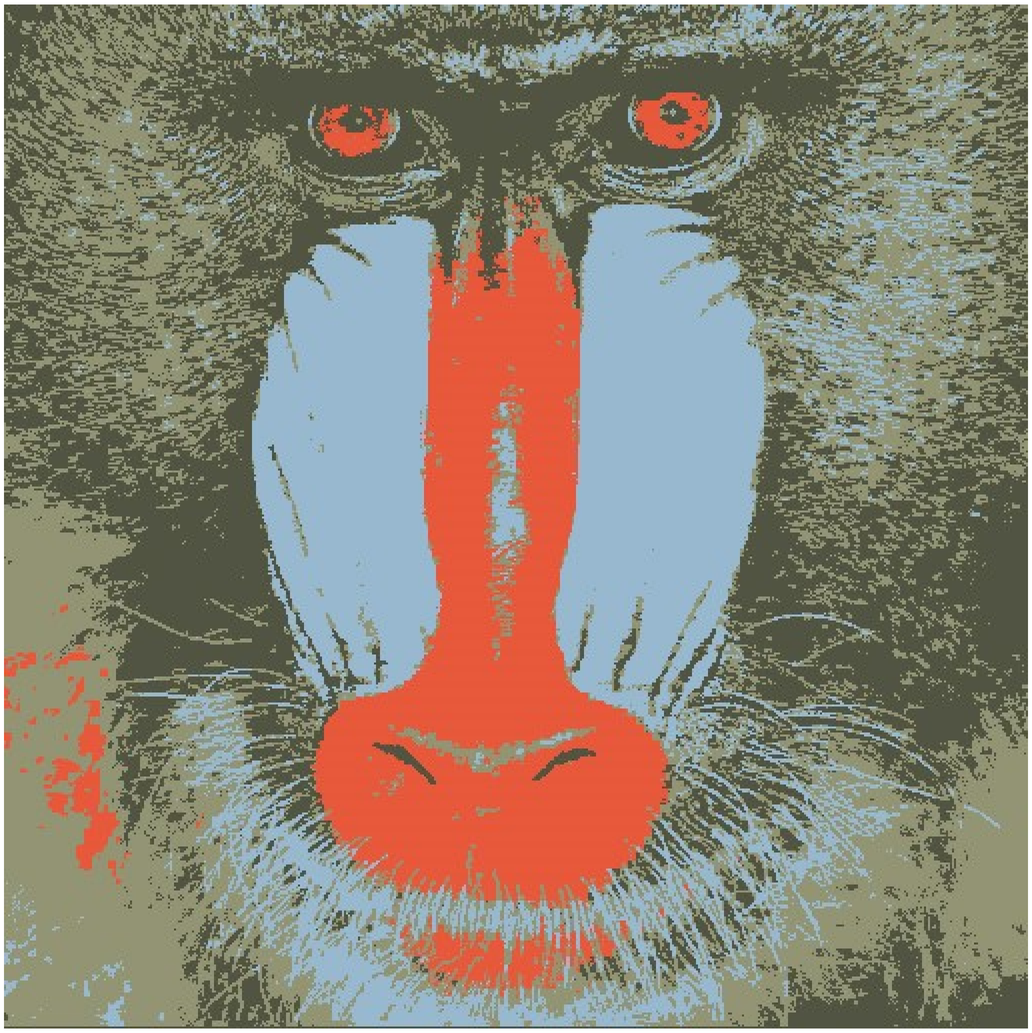
    Since we don’t use a barrier or critical clause, the only barriers are the four barriers for each separate task. Can’t minimize beyond four. And we also can’t fuse two tasks together to get rid of a barrier as they’re pretty independent of each other in terms of functionality.

Results below:



The old code which uses static scheduling is still much faster than the dynamic code.

Number of threads	Old execution time	New execution time
1	1.256539	1.217514
2	.604139	1.693374
3	.422646	1.558949
4	.281406	1.480517
5	.340327	1.372330
6	.278709	1.454376
7	.369698	1.462272
8	.239374	1.561860



Here’s the final segmented image too!