ASSIGNMENT 2: GABRIELE RUGGERI

BLURRING IMPLEMENTATION

To blur the image I defined a function `apply_kernel` that blurs elementwise and that takes in input an array of *unsigned short int*, its sizes, the indices of the elements where we apply the kernel, the kernel size and the *margin* that is a parameter dependent on *k_size* that for sake of simplicity was not computed in the function itself. The code is the following:

```
1  unsigned short int apply_kernel(const unsigned short int* A, const int R, const
       int C, const int i, const int j,const float* K, const int k_size, int
       margin)
2  {
3      unsigned short int blurred = 0;
4      float cont = 0;
5      int row_k, col_k;
6      for(int row = i - margin; row < i + margin + 1; row++){
7          for(int col = j - margin; col < j + margin + 1; col++){
8              row_k = row - i + margin;
9              col_k = col - j + margin;
10             cont += ((float)A[row*C+col]) * K[row_k*k_size+col_k];}
11     }
12     blurred = (unsigned short int)cont;
13     return blurred;
14 }
```

Initially I thought to use a
`#pragma omp parallel for collapse(2) reduction(+:cont) schedule(static)`
and then a
`#pragma omp atomic update`
on line 12 to parallelize the computations but the size of the kernel was not big enough to actually take an advantage out of that so I removed it.
In the code I used a double *for* loop to make the computations over all the elements involved, at first I declared the *int row_k, col_k* that allowed me

to "overlap" the kernel to the original image. It is worth to mention that since the kernel is of type *float* I had to perform a cast from *unsigned short int* to *float* on the elements of the image and in the end I casted *cont*, that stores the value of the blurred pixel, fromfloat to *unsigned short int*. This may lead to a small loss of information that anyway did not affect noticeably the result.

OMP IMPLEMENTATION

The program was designed such that the master thread is responseble to allocate the memory for all the quantities needed,it creats the kernel, generates the gradient and swaps the endianism. Now the first parallel region was created and used by the threads to read and store the image in a larger array such that it contains a noul frame to take care of the border effect and then, in an other parallel region, the threads blurred the image storing it in the array *image_o*: (For a better readability $xsize = C$ and $ysize = R$)

```
1  #pragma omp parallel
2  {
3      #pragma omp for collapse(2) schedule(static)
4      for(int i = 0; i < R; i++){
5          for(int j = 0; j < C; j++){
6              big_image_i[(i+margin)*(C+2*margin)+j+margin] = ((unsigned
                   short int*)ptr)[i*C+j];}
7      }
8  }
9  #pragma omp parallel
10 {
11     #pragma omp for collapse(2) schedule(dynamic)
12     for(int i = 0; i < R; i++){
13         for(int j = 0; j < C; j++){
14             image_o[i*C+j] = apply_kernel(big_image_i,R+2*margin,C+2*
                   margin,i+margin,j+margin,kernel,k_size,margin);}
15         }
16 }
```

In the end the master thread swapped again the endianism, wrote the blurred image and freed the memory.

MPI IMPEMENTATION

In this implementation, that adapts the idea behind the *omp* implementation to the *mpi* paradigm, the master process is meant to handle communications and the division of data. I decided to divide the input image in chunks with an equal number of columns and to send every chunk to a process. The first quantity computed was `int myrows = R/numproc`, where *numproc* is the number of processes in the communicator, so that every process has *myrows* rows of the input image (actually the last process has `myrows+Rmod(numproc)` rows). At this point every process stored a properly sized (`int little_size=(2*margin+myrows)*(C+2*margin) and int last_size=(last_rows+2*margin)*(C+2*margin)` for the last process) *unsigned short int* array for both input chunck and output chunk (to be sent back). To apply the kernel on elements in the first/last row of its own chunck, every process had to know some external informations from other processes (halo layers basically). To overcome this issue, given a process i, I computed the number of elements required from both process $i-1$ and $i+1$ (naturally it's only one of them for both the master and the last process) as `int layer_size=margin*(C+2*margin)`; at this point the master sent to every process in the communicator both the layers and its own part of the image. As we see in the communications there are some quantities like *send_starting_elem* and *init* that tell the master where to start sending at every iteration:

```
1 int send_starting_elem = myrows*(C+2*margin) - layer_size;
2 for(int dest=1;dest<numproc;dest++){
3     int init = send_starting_elem + (dest-1)*myrows*(C+2*margin);
4         if(dest<numproc-1){
5             MPI_Send(&big_image_i[init], little_size , MPI_TYPE, dest, TAG,
                  COMM);}
6         else{
7             MPI_Send(&big_image_i[init], last_size , MPI_TYPE, dest, TAG,
                  COMM);}
8 }
```

Now that every process had all it needed the computations went on in the same way as the *omp* version (they called *apply_kernel*) and then in the same way as above every process sent back the results to the master and the master stored them in *image_o*; after repeating the same steps as in the *omp* implementation the program ends.