Algorithm 1 Parallel Horizontal Image Flip

```
procedure FlipHorizontalParallel(shared_data, rows, cols, channels, rank, num_processes)
    half_rows ← rows / 2
    block_size ← half_rows / num_processes
    start_row ← rank × block_size
    end_row ← (rank + 1) × block_size
    for e doach row in parallel
        corresponding_row ← rows - 1 - i
        Swap(row, corresponding_row) for all color channels
    end for
end procedure
```

Algorithm 2 Parallel Vertical Image Flip

```
procedure FlipVerticalParallel(shared_data, rows, cols, channels, rank, num_processes)
    block_size ← rows / num_processes
    start_row ← rank × block_size
    end_row ← (rank + 1) × block_size
    for i = start_row to end_row - 1 do
        for j = 0 to (cols / 2) - 1 do
            left_idx ← (i × cols + j)
            right_idx ← (i × cols + (cols - 1 - j))
            swap(left_idx, right_idx) for all color channel
        end for
    end for
end procedure
```

Algorithm 3 90-Degree Image Rotation (Parallel with MPI)

procedure ROTATEIMAGE90PARALLEL(input_data, output_data, rows, cols, channels, rank, num_processes)
    block_size ← rows / num_processes
    start_row ← rank × block_size
    end_row ← (rank+1) × block_size
    if direction = CLOCKWISE **then**
        for r = start_row to end_row-1 **do**
            for c = 0 to cols-1 **do**
                out_r ← c
                out_c ← rows - 1 - r
                Assign at (out_r, out_c) with value at (in_c, in_r)
            end for
        end for
    else ▷ COUNTERCLOCKWISE
        for r = start_row to end_row-1 **do**
            for c = 0 to cols-1 **do**
                out_r ← cols - 1 - c
                out_c ← r
                Assign at (out_r, out_c) with value at (in_c, in_r)
            end for
        end for
    end if
end procedure

Algorithm 4 Parallel Image Color Channel Transformation

Require: Image of dimensions $rows \times cols$, number of processes P
Require: Channel increments: red_inc, $green\_inc$, $blue\_inc$
$block\_size \leftarrow \lfloor rows/P \rfloor$
$start\_row \leftarrow rank \times block\_size$
$end\_row \leftarrow \begin{cases} rows & \text{if } rank = P-1 \\ (rank+1) \times block\_size & \text{otherwise} \end{cases}$
for $r \leftarrow start\_row$ to $end\_row - 1$ **do**
    for $c \leftarrow 0$ to $cols - 1$ **do**
        $new\_color \leftarrow$ Calculate the new magnitude of each channels
        $row[r, c] \leftarrow new\_color$
    end for
end for
Synchronize all processes

Algorithm 5 1D Fast Fourier Transform

```
procedure FFT(x)
    n ← length(x)
    bits ← log_2(n)
    result ← new array of size n
                                                ▷ Bit reversal stage
    for i = 0 to n-1 do
        result[BitReverse(i, bits)] ← x[i]
    end for
                                                ▷ Butterfly operations
    for stage = 1 to bits do
        m ← 2^stage
        half_m ← m/2
        w_m ← e^(−2πi/m)
        for k = 0 to n-1 step m do in parallel
            w ← 1
            for j = 0 to half_m-1 do
                t ← w × result[k + j + half_m]
                u ← result[k + j]
                result[k + j] ← u + t
                result[k + j + half_m] ← u - t
                w ← w × w_m
            end for
        end for
    end for
     return result
end procedure
```

Algorithm 6 2D Fast Fourier Transform

```
procedure FFT2D(channel)
    rows ← channel.rows
    cols ← channel.cols
    padded_rows ← NextPowerOf2(rows)
    padded_cols ← NextPowerOf2(cols)
    complex_image ← new 2D array[padded_rows][padded_cols]
    Convert image to complex numbers and pad
                                                ▷ Apply FFT to rows
    for i = 0 to padded_rows-1 do in parallel
        row ← complex_image[i]
        row ← FFT(row)
    end for
                                                ▷ Apply FFT to columns
    for j = 0 to padded_cols-1 do in parallel
        col ← new array[padded_rows]
        col ← FFT(col)
    end for
     return complex_image
end procedure
```

Algorithm 7 Parallel Gaussian Blur using OpenMP

Require: Image I of size $w \times h$, radius r, sigma σ

```
k ← CreateGaussianKernel(r, σ)
T ← temporary buffer of size w × h × channels
for each pixel in each channel of a row in parallel do
    sum ← 0
    for i ← −r to r do
        sum ← sum + I[y, srcX, c] × k[i + r]
    end for
    T[y, x, c] ← sum
end for

Implicit barrier synchronization
for each pixel in each channel of a column in parallel do
    sum ← 0
    for i ← −r to r do
        sum ← sum + T[srcY, x, c] × k[i + r]
    end for
    I[y, x, c] ← sum
end for
```