# Parallel Architectures and Programming Models, 2021W

Assignment 2: OpenMP Tasking

# Mandelbrot + Parallel Image Filter

**Your assignment consists of parallelizing provided sequential application with OpenMP Tasking. The application consists of two parts:**

- **Mandelbrot**
  - Calculates the Mandelbrot set and draw it on the .ppm image
- **Parallel Image Filter**
  - Iterative 25-point image convolution filter on the produced linearized 2D image.
  - In each of the `nsteps` (=20) iteration steps, the average RGB-value of each pixel `p` in the source array is computed, considering `p` and its k (=24) neighbor pixels (in 2D), and written to the destination array.

**The image should be processed in parallel with OpenMP tasking**

- Your implemenation should integrate into the exsiting code (in Moodle)
  - `mandelbrot`, and `convolution_2d` funtions
- You need to submit your source code(s) and a report (see detail later)

# Mandelbrot Algorithm

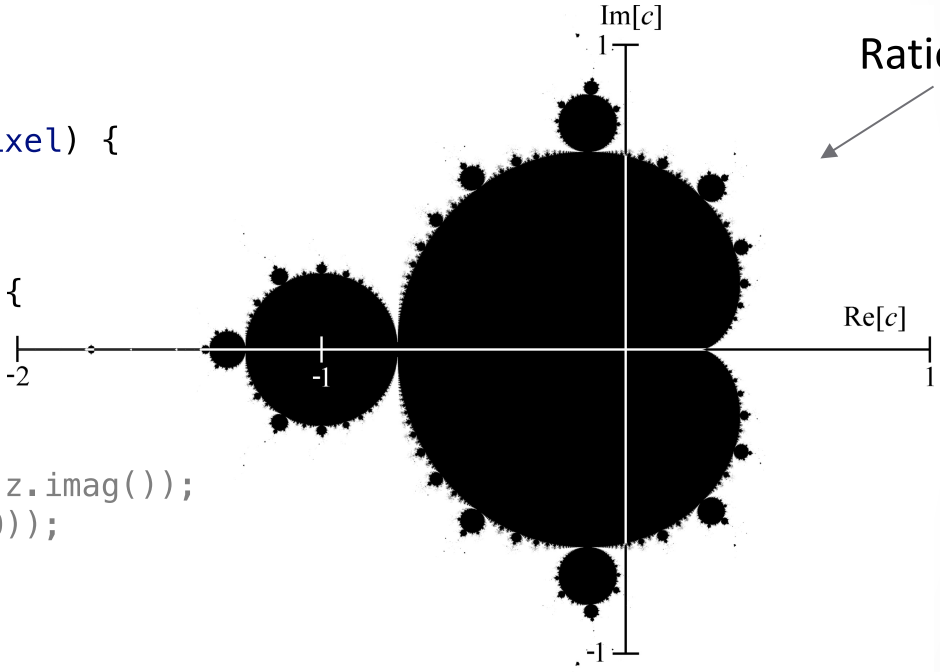
**A complex number c is a member of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of $z_n$ remains bounded for all**

```
bool mandelbrot_kernel(complex<double> c, vector<int>& pixel) {
   int max_iterations = 2048, iteration = 0;
   complex<double> z(0, 0);

   while ( abs(z) <= 4 && (iteration < max_iterations) ) {
      z = z * z + c;
      iteration++;
   }

   double length = sqrt(z.real() * z.real() + z.imag() * z.imag());
   long double m = (iteration + 1 - log(length) / log(2.0));
   double q = m / (double)max_iterations;

   q = iteration + 1 - log(log(length)) / log(2.0);
   q /= max_iterations;

   colorize(pixel, q, iteration, gradients);

   return (iteration < max_iterations);
}
```

Check if in the set

Calculation of gradient for colorizing outside pixels

Ratio = 1.5



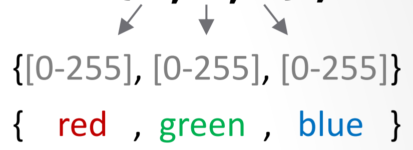
$$z_{n+1} = z_n^2 + c$$

Lear more about the Mandelbrot set here:
https://en.wikipedia.org/wiki/Mandelbrot_set

# Data Structure

**Image C++ struct** (a2-helpers.hpp)
- Contiguous data
- Altering the structure is allowed (e.g. to use (y,x,ch))
- or create static/dynamic C++ arrays…

```
vector<int> pixel = {0,0,0};
```



pixel: 3-values [0-255] (red, green, blue)

```
Image image(channels, height, width);

// example
for (int i = 0; i < height; i++) {
   for (int j = 0; j < width; j++) {

      // change pixel values (value for each rgb channel)
      for (int ch = 0; ch < channels; ch++) {
         image(ch,i,j) = pixel[ch];
      }
   }
}
```

Assign pixel value to the output image

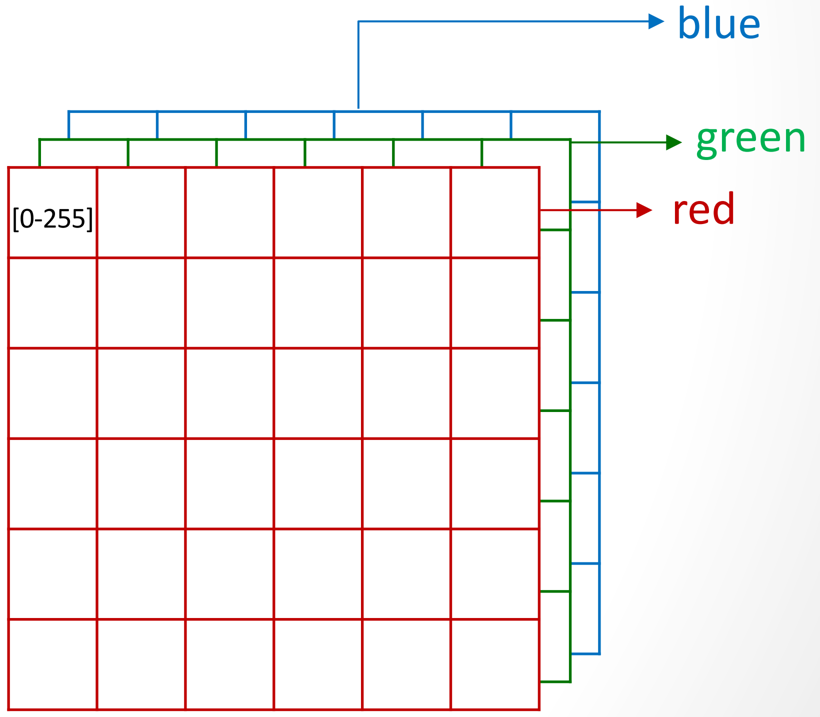


Image data structure

**Output file format** (text):
https://en.wikipedia.org/wiki/Netpbm#PPM_example

## Mandelbrot Part

```
bool mandelbrot_kernel(complex<double> c, vector<int>& pixel) {...}
```

- **Applies the algorithm for a single pixel**
- **`max_iterations` is set to 2048**
- **Can be though of as two step process**
  1. Checking if the pixel belong in the set
  2. Calculating the gradient for the pixel based on the number of iterations

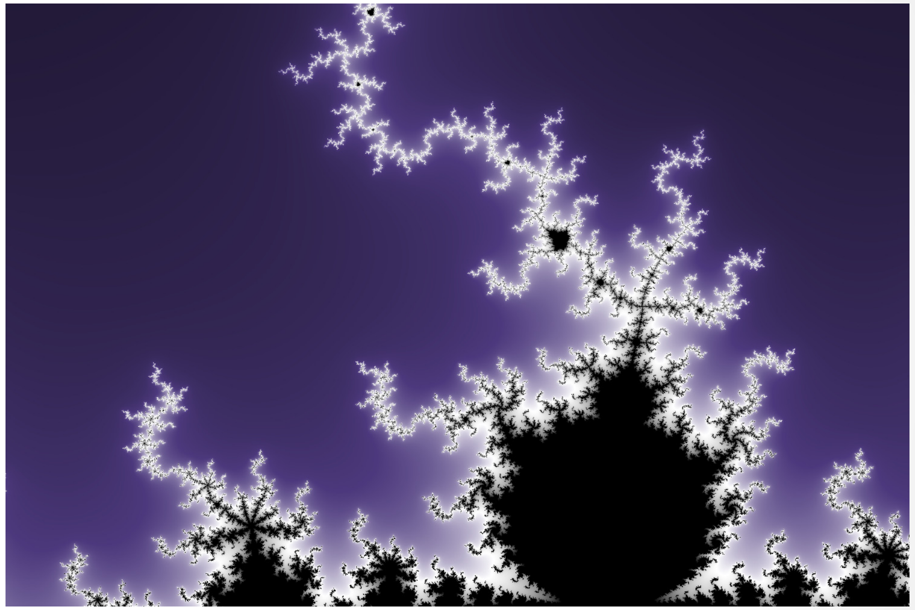
```
int mandelbrot( image, ratio )
```

- **Only image argument is required**
- **Generates the image on the right**
- **Image changes if the following values change** (see the sources)

```
double dx = (double)i / (w)*ratio - 1.10;
double dy = (double)j / (h)*0.1 - 0.35;
```

*Keep the default values for measurements

**You can modify `mandelbrot` and `mandelbrot_kernel` functions as needed**


Mandelbrot output image

# 2d Convolution

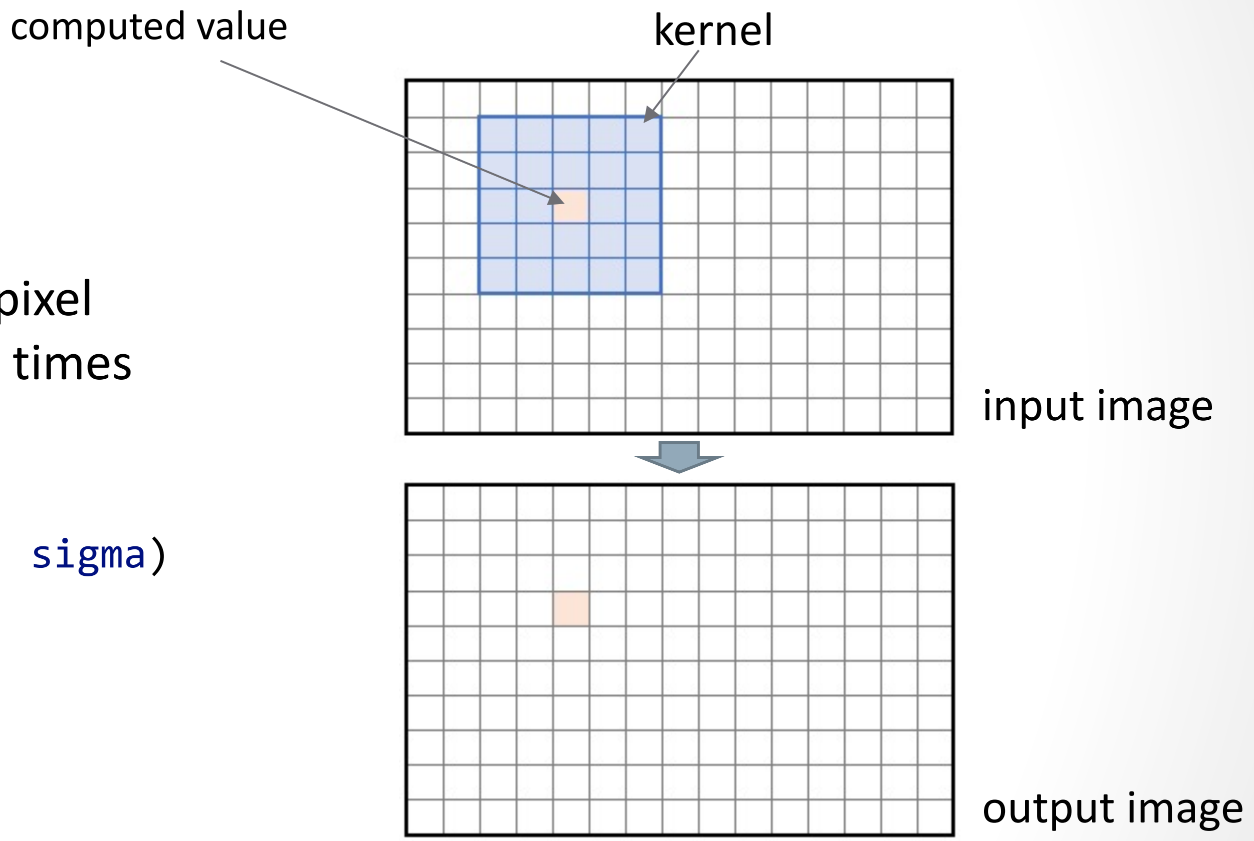
Kernel dimensions: **5x5 kernel**
Kernel type: **Gaussian filter kernel**

→ Compute for each pixel
  → 5x5 kernel → 25 multiplications per pixel
→ Iterate over the whole image for `nsteps` times
→ Do this for R, G and B channels

```
get_2d_kernel(kernel_width, kernel_width, sigma)
```

Makes the following kernel (width = 5):

```
0.0001 0.0020 0.0055 0.0020 0.0001
0.0020 0.0422 0.1171 0.0422 0.0020
0.0055 0.1171 0.3248 0.1171 0.0055
0.0020 0.0422 0.1171 0.0422 0.0020
0.0001 0.0020 0.0055 0.0020 0.0001
```

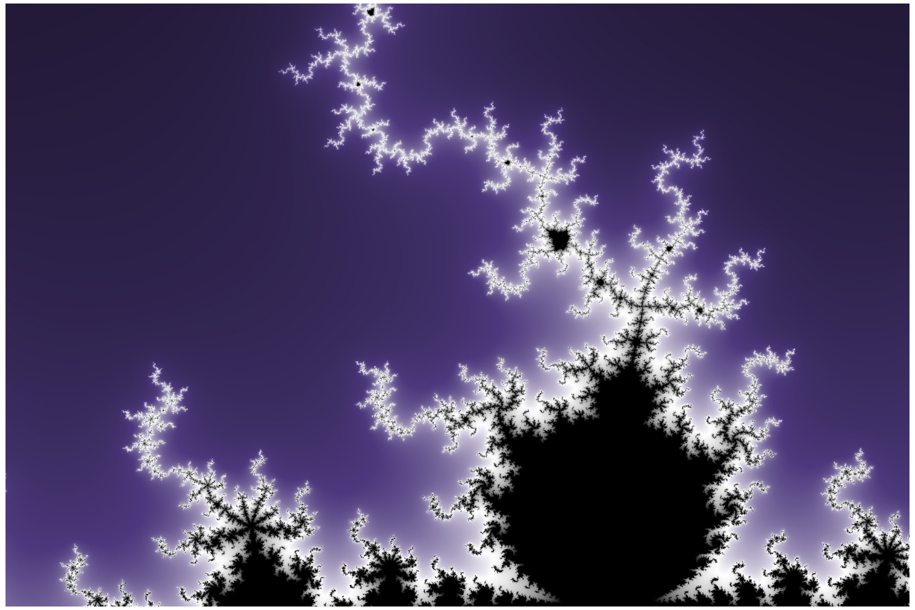


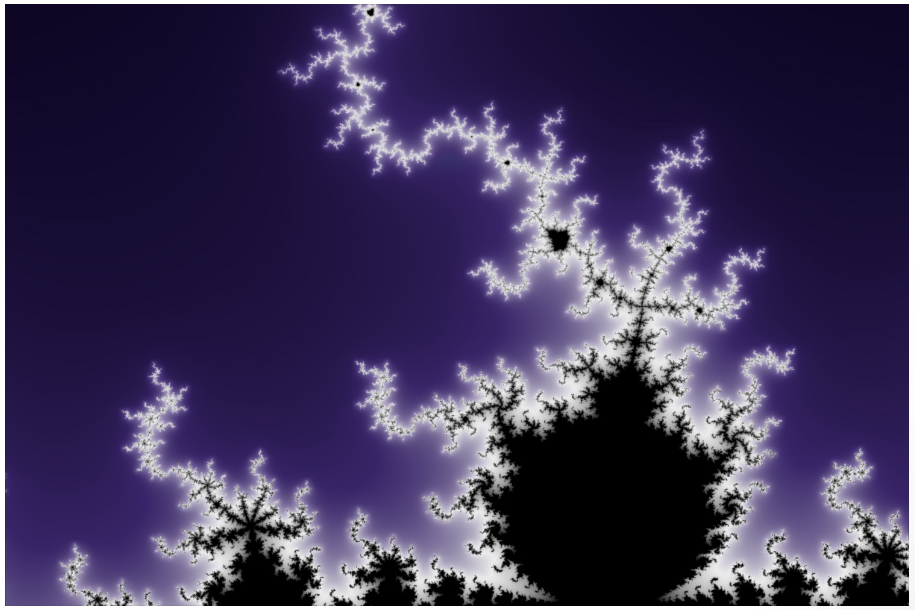
See more: https://en.wikipedia.org/wiki/Gaussian_blur

# 2D Convolution

```
void convolution_2d(Image &src, Image &dst, int kernel_width, double sigma, int nsteps=1) {…}
```

- Leave the defaults for the measurements (e.g., kernel_width, sigma, and 20 steps)
- Always compare-by-content with the sequential output
  - Images may visual appear the same, but they need to be exactly the same!
- You can alter some parameters if you want to see a different blurring effect
  - Adjusting sigma and kernel width will produce different results (kernel width must be an odd number)


Original image


Blurred image

## Code and Results

**Best effort parallel codes with OpenMP Tasking:**

1. Using **`omp task`** and the corresponding clauses for task generation
   - you can combine other tasking constructs except the **`taskloop`**
2. Using **`omp taskloop`** and the corresponding clauses for task generation
   - You can combine with any other tasking construct, including the `omp task`
   - Also all clauses, including `num_tasks` and `grainsize` (see the OpenMP Spec)
3. Try out “small” and “big” tasks
   - You can choose task granularity based on what you want to be your tasks
   - Try at least 2 significantly different scenarios (for both Mandelbrot and Convolution) e.g., small tasks could execute one or just a few ops
   - No need for an extra source codes, but report your findings in documentation, and explain what you tried

You can use basic OpenMP constructs like “omp parallel”, “omp single”, ”omp master”, but no other worksharing and synchronization constructs (omp for, omp sections, omp critical,…). OpenMP Tasking and clauses are required for this task.

**Explicitly define data scope for all variables (shared/private) in every OpenMP directive**

Note that you can modify the original code to some extent, but you should not remove things that affect the original output (e.g., number of pixels inside Mandelbrot).

# Code and Results

## Testing if your results are correct

- The output of total mandelbrot pixels must be the same as in the sequential version in each run!
  - "mandelbrot.ppm" image (1536x1024 resolution)
  - `Output: "Total Mandelbrot pixels: 1478025"`
- You parallel code must produce the same results as the sequential version (pixel-by-pixel)
  - Compare files by content to check (e.g., `diff`)
  - Example: $ diff mandelbrot.ppm mandelbrot-sequential.ppm should produce no output

## Compile on Alma (you need the newest GCC):

- `/opt/global/gcc-11.2.0/bin/g++ -O2 a2-sequential.cpp (sequential code)`
- `/opt/global/gcc-11.2.0/bin/g++ -O2 -fopenmp a2-openmp.cpp (for OpenMP)`

# Report: Measurements and Discussions

**Speedup graph(s) and table(s) comprising all measurements for all variants you produced during development and tested on Alma**

- Measure the speedup with respect to sequential code
  - Mandelbrot → Good solutions should achieve speedup >=13 for 16 threads.
  - Convolution → Good solutions should achieve speedup >=9 for 16 threads.

**Discuss why some versions perform good, and include at least:**

- Were there any performance differences between task and taskloop versions in performance?
- Explain why you think task granularity matters (small vs big tasks), and were there any performance differences? Were all use cases possible? Try to address the worst and the best and discuss why.
- Explain how your task operate on the given image and how you distributed the work among tasks
  - Try to give some quantitative information on the number of tasks you generate when using “omp task”
- Describe any differences in speedup you observed with different clauses (if any)
- And similar interesting findings …

# Code Remarks

**Compile on Alma (you need the newest GCC):**

- `/opt/global/gcc-11.2.0/bin/g++ -O2 a2-sequential.cpp` **(sequential code)**
- `/opt/global/gcc-11.2.0/bin/g++ -O2 -fopenmp a2-openmp.cpp` **(for OpenMP)**

**Program produces following output:**

- "mandelbrot.ppm" image (1536x1024 resolution) (output should always be the same(!))
- Total number pixels inside the set (output should always be the same(!))
- Execution times for Mandelbrot, Convolution and Total

**Explicitly define data scope for all variables (shared/private) in every OpenMP directive**

Note: You can download the full sequential version of Mandelbrot set from the Moodle (inside of a zip file for Assignment 2 → pap/a2/a2-sequential.cpp).

* Alma nodes have 16 cores with hyperthreading enabled (use "cat /proc/cpuinfo" to get details on available number of "logical" cores, and perform experiments to find out the best number of threads for your code)

## General and Submission Remarks

**Use forum for questions**

**Remember to insert a table with speedup numbers, time measurements, and a discussion that address the performance of your code**

- The maximum score when the report or the source code is missing is 25%

**Develop on your PC/laptop, but test and measure early on Alma, to avoid having to wait for the nodes to free up, otherwise your measurements may not be accurate.**

**Lookup the OpenMP Specification 5.2 for additional information:**

- https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
- Sections 5 and 12 could be of a particular interest