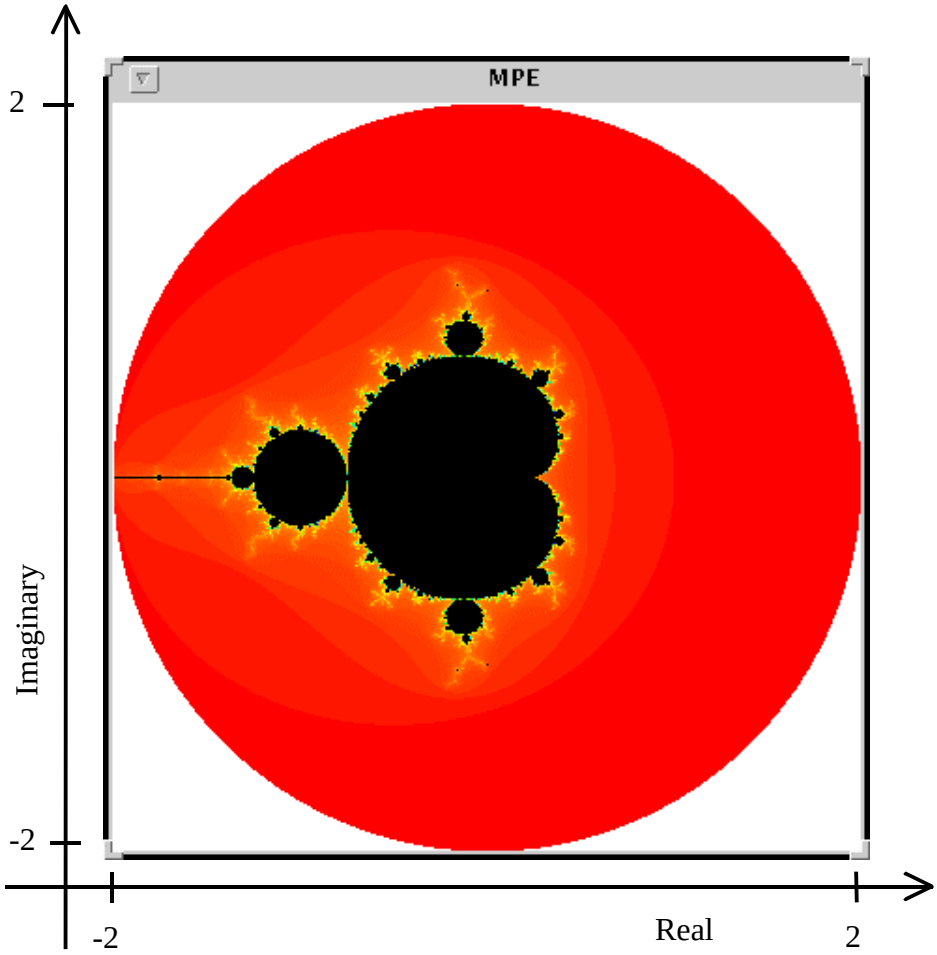
High Performance Computing

Assignment. The Mandelbrot set

Write a MPI program that calculates the Mandelbrot set and presents the result graphically using the MPE graphics library.



The Mandelbrot set

A Mandelbrot set is a set of points in the complex plane. A complex number $z = a+bi$ consists of two components: the real part a and the imaginary part b (where i is $\sqrt{-1}$). Imaginary numbers can for instance be represened by a structure

```
typedef struct {
    double real;
    double imag;
} complex_type;
```

The Mandelbrot function computes iterations of a complex number *z* using the function $z_{k+1} = z_k^2 + c$, where z_k is iteration number *k* of the complex number *z* and *c* is a complex number giving the position of the point in the complex plane. The initial value of *z* is zero. The iterations are continued until the magnitude of *z* is greater than 2 (which indicates that *z* will go to infinity) or until the number of iterations reaches some pre-defined limit (often 256). The magnitude of *z* is the length of the vector described by *z*, i.e. $\sqrt{a^2 + b^2}$.

To compute the function $z_{k+1} = z_k^2 + c$, we note that $z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$ (because $i = \sqrt{-1}$). Thus, the square of a complex number *z* is a complex number with a real part $a^2 - b^2$ and an imaginary part *2ab*.

Hence, if z_{real} is the real part and z_{imag} is the imaginary part of a complex number *z*, the next iteration can be computed by $z_{real} = z_{real}^2 - z_{imag}^2 + c_{real}$ and $z_{imag} = 2z_{real}z_{imag} + c_{imag}$, where c_{real} and c_{imag} of course are the real respectively imaginary parts of the complex number *c*.

Computing the Mandelbrot set

A function for computing the value of one point in the complex plane and returning the number of iterations, using the structure for imaginary numbers defined above, looks like this:

```
int calc_mandel(complex_type c)
{
  int count, max;
  complex_type z;
  double len2, temp;
  max = 255;          /* Max nr of iterations */
  count = 0;
  z.real = z.imag = 0.0;
  do {                /* Mandelbrot function */
    temp = z.real*z.real - z.imag*z.imag + c.real;
    z.imag = 2.0*z.real*z.imag + c.imag;
    z.real = temp;
    len2 = z.real*z.real + z.imag*z.imag;
    count++;
  } while ((len2<4.0) && (count<max));
  return count;
}
```

Note that we compare the square of the length (in `len2`) against 4.0 instead of comparing the length against 2.0. This saves us a square root operation.

We compute the Mandelbrot function for all points of a rectangular window of size HEIGHT*WIDTH (where in our case HEIGHT=WIDTH). A suitable window size is for instance 800*800 pixels (but this depends on what resolution your display has).

A point *(x,y)* in the graphical window is translated to a point *c* in the complex plane by multiplying the *x* and *y* coordinates with the scaling factors

```
scale_real = (real_max – real_min) / WIDTH
scale_imag = (imag_max – imag_min) / HEIGHT
```

The sequential code for computing the Mandelbrot function over the pixels in a window of size WIDTH*HEIGHT will then be something like this:

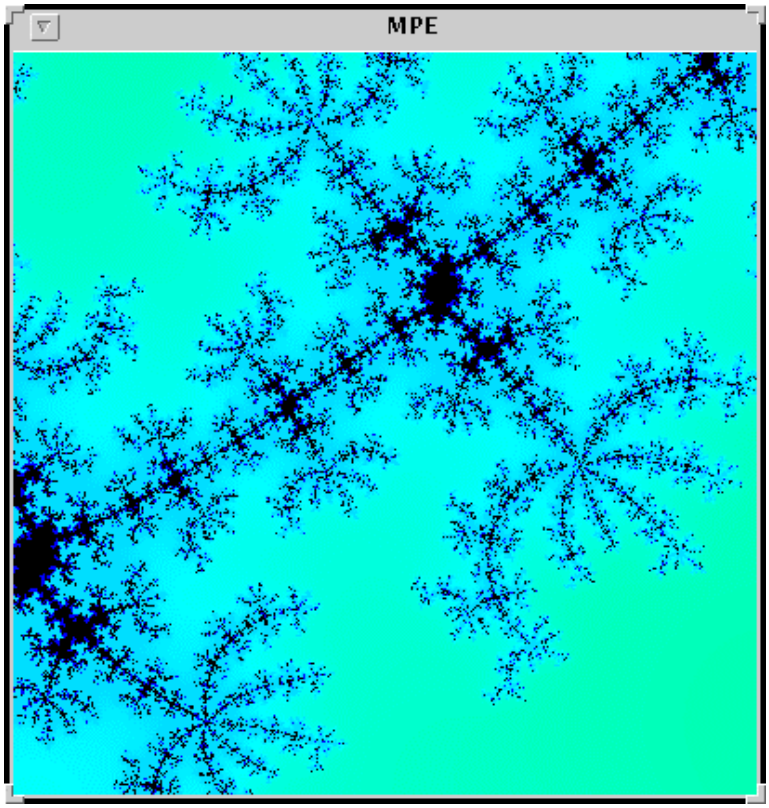
```
real_min = imag_min = -2.0;
real_max = imag_max =  2.0;
scale_real = (real_max-real_min)/(double)WIDTH;
scale_imag = (imag_max-imag_min)/(double)HEIGHT;

for (y=0; y<HEIGHT; y++) {  /* Compute one row at a time */
  for (x=0; x<WIDTH; x++) {
       c.real = real_min + ((double) x*scale_real);
       c.imag = imag_min + ((double) y*scale_imag);
      color = calc_mandel(c);
      set pixel (x,y) in the graphical window to color;
  }
}
```

Zooming in

When the current area of the Mandelbrot set has been calculated and displayed, the user should be able to zoom in on a rectangular area of the window. Use the procedure `MPE_Get_drag_region` to select a rectangular area defined by two points *(x1,y1)* and *(x2,y2)*. The selected area in the graphical window can be translated to an area in the complex plane in the folowing way:

```
new_real_min = real_min + (real_max-real_min)*(double)x1/WIDTH;
new_imag_min = imag_min + (imag_max-imag_min)*(double)y1/HEIGHT;
new_real_max = real_min + (real_max-real_min)*(double)x2/WIDTH;
new_imag_max = imag_min + (imag_max-imag_min)*(double)y2/HEIGHT;
```

We also have to update the scaling factors, in the same way as above. After this, we calculate and draw the zoomed region in the same way as above. You can continue to zoom in repeatedly, and find interesting pictures for instance like this:



Parallel implementation

Implement the parallel program as a master–slave computation. You can for instance give each slave one row of the window to compute. The master only keeps track of which rows have been computed, and gives slaves a new row as soon as they have finished the previous one. Note that the slaves can display their results themselves, they do not have to send the calculated pixels to the master.

When a window is ready, the master waits for the user to zoom in on a region. It sends the new coordinates of the selected region to the slaves and starts sending out new rows to the slaves in the same way as above.

Termination

Make sure that your program terminates gracefully. The user can for instance indicate that he/she wants to quit by just clicking in the graphical window instead of dragging the mouse. Thus, if the area that is selected by `MPE_Get_drag_region` is smaller than 10 pixels (for instance), the program terminates instead of zooming in.

Timing measurement

The master process should measure how log time it takes to compute a window, and print out this value to standard output after each window has been completed. Try to make your implementation efficient. Check that you don't have any unnecessary communication in the program. Also try to keep the slaves as busy as possible, while there are calculations left to be done.

Please try to code your programs so that somebody else also may understand what they do. Write plenty of comments in your code!