Homework #2

The goal of this assignment is to

- Improve the performance of a fractal image generator using the Python multiprocessing package.
- Use a least two different Pool methods to speed up your fractal image generator.
- Produce a unique fractal image along with timing results. Submit your results to a class poll.

For details on how fractals are generated, see practice notebooks from Week #4.

In [1]:
```
%matplotlib inline
import multiprocessing as mp
mp.set_start_method('fork')
from matplotlib.pyplot import *
from numpy import *
```

In [2]:
```
import warnings
warnings.simplefilter("ignore")    # Suppress overflow run time warnings

def julia_set(ax,bx,ay,by, Nx, Ny, kmax,c):

    # Generate points in complex planes D
    xe = linspace(ax,bx, Nx+1)
    ye = linspace(ay,by, Ny+1)
    dx = (bx-ax)/Nx
    dy = (by-ay)/Ny
    xc = xe[:-1] + dx/2
    yc = ye[:-1] + dy/2

    # Set of initial values Z0 : zk is a Nx x Ny matrix
    zk = xc + yc[:, None] * 1j

    # Constant needed for Julia fractal : g(z) = z^2 + c
    C = zeros_like(zk) + c

    # Divergence criteria
    rho = 2.0

    # Vectorize the computation of g(z); Use
    escape_time = zeros_like(zk,dtype=int) + kmax
    for n in range(kmax):
        escaped = less(escape_time,kmax)
        if all(escaped):
            break
        I = logical_and(greater(abs(zk), rho),logical_not(escaped))
        escape_time[I] = n
        notI = not_equal(I,True)
        zk[notI] = zk[notI]**2 + C[notI]
    Iz = equal(escape_time,kmax)
    nz = count_nonzero(Iz)  # Number of zero values who never escaped
    return escape_time,nz
```

Problem 1 : Fractal generation (serial)

Generate a fractal using a serial code. Time your results and report the timing results. Your resulting image should be at least 2048 x 2048. Use the original domain of $[-2, 2] \times [-2, 2]$ and original c values.

Your timing results should include the time required to generate the entire matrix M, but does not need to include the actual plotting, using either `imshow` or `imsave`.

- Use `imshow` to show your image in the notebook
- Use `imsave` to save the fractal to a PNG file. Submit your PNG file along with your notebook.

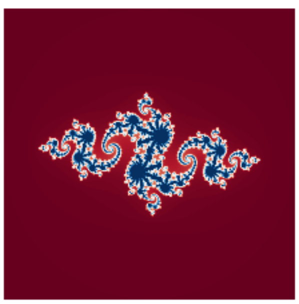
In [3]:
```
res = 2048
kmax = 1000

domain_width = 4
xc = 0
yc = 0

ax0, bx0 = xc - domain_width/2, xc + domain_width/2
ay0, by0 = yc - domain_width/2, yc + domain_width/2

# Choice from Mandelbrot set
c = -0.8+0.156*1j
```

In [4]:
```
%%time

N0 = res

M, nz = julia_set(ax0, bx0, ay0, by0, N0, N0, kmax,c)
print(f"Number of values that did not escape {nz}")
```

```
Number of values that did not escape 726
CPU times: user 1min 26s, sys: 11.6 s, total: 1min 38s
Wall time: 1min 38s
```

In [5]:
```
%%time

dpi = 16


imshow(M,vmin=0,vmax=kmax/5,origin='lower',cmap=cm.RdBu)
gca().axis('off')

imsave("Problem1_serial.png",M,vmin=0,vmax=kmax/5,cmap=cm.RdBu,dpi=dpi,origin='lower')

```

```
CPU times: user 371 ms, sys: 49.7 ms, total: 421 ms
Wall time: 427 ms
```



Problem 2 : Fractal generation (parallel)

Generate the fractal from problem #1 using a parallel code.

- Use a Pool method. Report your timing results.
- Include in your timing results any post-processing needed to create full M matrix needed to generate the fractal image.

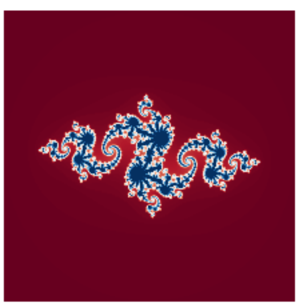
In [6]:
```
nquads = 4

dex = linspace(ax0,bx0,nquads+1)
dey = linspace(ay0,by0,nquads+1)

N = res//nquads
assert N*nquads == res, 'res must be divisible by nquads'
```

In [7]:
```
%%time

def create_square(args):
    ax,bx,ay,by,N,kmax,c = args
    M,nz = julia_set(ax, bx, ay, by, N, N, kmax,c)
    return M,nz

data = []
for i in range(nquads):
    for j in range(nquads):
        ax,bx = dex[i],dex[i+1]
        ay,by = dey[j],dey[j+1]
        data.append((ax,bx,ay,by,N,kmax,c))


with mp.Pool(processes=12) as pool:
    results_async=pool.map_async(create_square,data)
    pool.close()
    pool.join()

results=results_async.get()

F = empty((res,res),dtype=int)

nzt = 0
for i in range(nquads):
    for j in range(nquads):
        M,nz = results.pop(0)
        F[j*N:(j+1)*N,i*N:(i+1)*N] = M
        nzt += nz

print(f"Number of values that did not escape {nzt}")

```

```
Number of values that did not escape 726
CPU times: user 138 ms, sys: 227 ms, total: 365 ms
Wall time: 22.8 s
```

In [8]:
```
%%time

dpi = 16


imshow(F,vmin=0,vmax=kmax/5,origin='lower',cmap=cm.RdBu)
gca().axis('off')

imsave("Problem2_parallel.png",F,vmin=0,vmax=kmax/5,cmap=cm.RdBu,dpi=dpi,origin='lower')

```

```
CPU times: user 382 ms, sys: 56.8 ms, total: 439 ms
Wall time: 437 ms
```



Problem 3 : Generate a unique fractal

By varying the constant c, choosing a zoomed in region for the fractal, and choosing different colormaps, you can generate some spectacular fractal images.

For this problem, create a **unique, high resolution fractal image** from some Julia set. Specify the domain you used, the `kmax` value and the value of c you choose so that others can reproduce your image.

- Report your timing results for both a serial and parallel code.
- Submit your timing results and a PNG file of your image for an in-class competition.

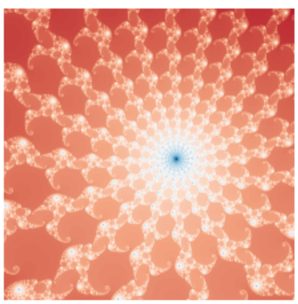
In [9]:
```
res = 4096
kmax = 1000

domain_width = 4/2**11
xc = 0.18680
yc = 0.21641


ax0, bx0 = xc - domain_width/2, xc + domain_width/2
ay0, by0 = yc - domain_width/2, yc + domain_width/2

c=0.28+0.008j
```

In [10]:
```
%%time

N0 = res

M, nz = julia_set(ax0, bx0, ay0, by0, N0, N0, kmax,c)
print(f"Number of values that did not escape {nz}")
```

```
Number of values that did not escape 0
CPU times: user 5min 48s, sys: 2min 20s, total: 8min 8s
Wall time: 8min 10s
```

In [11]:
```
%%time

dpi = 16

e=[ax0,bx0,ay0,by0]
imshow(M,vmin=0,extent=e,vmax=kmax/2,origin='lower',cmap=cm.RdBu)
gca().axis('off')

imsave("Problem3_serial.png",M,vmin=0,vmax=kmax/2,cmap=cm.RdBu,dpi=dpi,origin='lower')

```

```
CPU times: user 1.94 s, sys: 159 ms, total: 2.09 s
Wall time: 2.1 s
```



In [12]:
```
nquads = 4

dex = linspace(ax0,bx0,nquads+1)
dey = linspace(ay0,by0,nquads+1)

N = res//nquads
assert N*nquads == res, 'res must be divisible by nquads'
```

In [ ]:
```

```

In [13]:

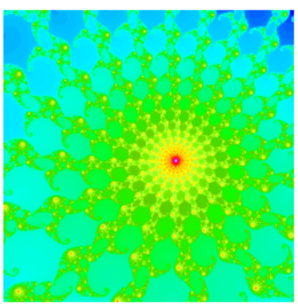
```
%%time
data = []
for i in range(nquads):
    for j in range(nquads):
        ax,bx = dex[i],dex[i+1]
        ay,by = dey[j],dey[j+1]
        data.append((ax,bx,ay,by,N,kmax,c))


with mp.Pool(processes=12) as pool:
    results_async=pool.map_async(create_square,data)
    pool.close()
    pool.join()

results=results_async.get()

F = empty((res,res),dtype=int)

nzt = 0
for i in range(nquads):
    for j in range(nquads):
        M,nz = results.pop(0)
        F[j*N:(j+1)*N,i*N:(i+1)*N] = M
        nzt += nz

print(f"Number of values that did not escape {nzt}")

```

```
Number of values that did not escape 0
CPU times: user 585 ms, sys: 901 ms, total: 1.49 s
Wall time: 2min 17s
```

In [14]:

```
%%time
dpi = 16


imshow(F,vmin=0,vmax=kmax/2,origin='lower',cmap=cm.gist_ncar)
gca().axis('off')

imsave("Problem3_parallel.png",F,vmin=0,vmax=kmax/2,cmap=cm.gist_ncar,dpi=dpi,origin='lower')

```

```
CPU times: user 1.98 s, sys: 210 ms, total: 2.19 s
Wall time: 2.18 s
```



Problem 4 : Using Pool methods (571)

Use the fractal project to demonstrate how to use at least two different Pool methods not discussed in class. You can read up on different pool methods here (https://superfastpython.com/multiprocessing-pool-python).

- Provide a discussion of what to consider when choosing a Pool method for the fractal generation problem. Some issues include whether to choose a synchronous or asynchronous method, whether to use a callback function, or what types of arguments the target function can take.
- Show any timing results you obtain. Are there obvious benefits of one method over the other?

In [15]:
```

res = 4096
kmax = 2000

domain_width = 4/2**8
xc = 0.02384
yc = 0.04007


ax0, bx0 = xc - domain_width/2, xc + domain_width/2
ay0, by0 = yc - domain_width/2, yc + domain_width/2

c=-0.7269 + 0.1889j
```

In [16]:
```
nquads = 4

dex = linspace(ax0,bx0,nquads+1)
dey = linspace(ay0,by0,nquads+1)

N = res//nquads
assert N*nquads == res, 'res must be divisible by nquads'
```

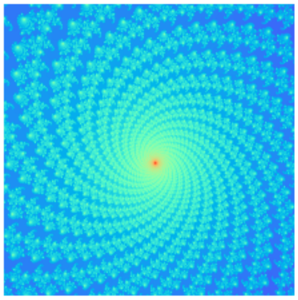
In [17]:
```
def create_square(ax,bx,ay,by,N,kmax,c):
    M,nz = julia_set(ax, bx, ay, by, N, N, kmax,c)
    return M,nz

```

In [18]:
```
%%time
F = empty((res,res),dtype=int)

data = []
for i in range(nquads):
    for j in range(nquads):
        ax,bx = dex[i],dex[i+1]
        ay,by = dey[j],dey[j+1]
        data.append((ax,bx,ay,by,N,kmax,c))

with mp.Pool(processes=12) as pool:
    results=pool.starmap_async(create_square,data)
    pool.close()
    pool.join()

output=results.get()

nzt=0
for i in range(nquads):
     for j in range(nquads):
        M,nz = output.pop(0)
        F[j*N:(j+1)*N,i*N:(i+1)*N] = M
        nzt += nz
print(f'Number of not escaped points = {nzt}')
```

```
Number of not escaped points = 14437
CPU times: user 399 ms, sys: 628 ms, total: 1.03 s
Wall time: 6min 24s
```

In [19]:
```
%%time
dpi = 16

imshow(F,vmin=0,extent=e,vmax=kmax,origin='lower',cmap=cm.rainbow)
gca().axis('off')

imsave("Problem4_parallel_1.png",F,vmin=0,vmax=kmax+kmax/5,cmap=cm.rainbow,dpi=dpi,origin='lower')

```

```
CPU times: user 2.8 s, sys: 222 ms, total: 3.02 s
Wall time: 3.03 s
```



In [20]:

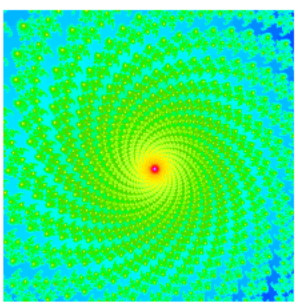
```
%%time
F = empty((res,res),dtype=int)

data = []
for i in range(nquads):
    for j in range(nquads):
        ax,bx = dex[i],dex[i+1]
        ay,by = dey[j],dey[j+1]
        data.append((ax,bx,ay,by,N,kmax,c))


with mp.Pool(processes=12) as pool:
    result=pool.starmap(create_square,data)
    pool.close()
    pool.join()

nzt=0
for i in range(nquads):
     for j in range(nquads):
        M,nz = result.pop(0)
        F[j*N:(j+1)*N,i*N:(i+1)*N] = M
        nzt += nz
print(f'Number of not escaped points = {nzt}')
```

```
Number of not escaped points = 14437
CPU times: user 469 ms, sys: 764 ms, total: 1.23 s
Wall time: 5min 57s
```

In [21]:

```
%%time

dpi = 16


imshow(F,vmin=0,vmax=kmax,origin='lower',cmap=cm.gist_ncar)
gca().axis('off')

imsave("Problem4_parallel_2.png",F,vmin=0,vmax=kmax,cmap=cm.gist_ncar,dpi=dpi,origin='lower')

```

```
CPU times: user 4.01 s, sys: 226 ms, total: 4.24 s
Wall time: 4.24 s
```



DISCUSSION

Certain factors must be considered when choosing a Pool method for fractal generation.

Firstly, determine whether a synchronous or an asynchronous pool method best suits the task. Asynchronous pool methods return the result as soon as a job is completed. On the other hand, a synchronous pool method would wait for all tasks to be completed before returning results.

Secondly, one should consider whether or not to employ a callback function. When a specific task is finished, a callback function is invoked. This can be helpful for tasks that depend on the outcomes of earlier jobs because the callback can be used to transfer those outcomes to subsequent jobs.

Finally, depending on your particular pool method, the target function may be limited to the primary data type or the number of arguments needed for successful compilation. If the fractal generation problem requires multiple-input arguments, it may be necessary to use a pool method that suits this requirement.

In this exercise, the synchronous method (starmap) and the asynchronous method (starmap_async) completed the task in approximately the same time; there is no obvious difference between them. However, this may change as the complexity of the problem increases.

Extra credit : Mandelbrot set.

Generate a high resolution image of the Mandelbrot set and describe the connection between the Mandelbrot set and the Julia sets we have been creating above.

- Report your timing results.

In [ ]:

```

```

In [22]:

```
def mandel_set(x0,y0,kmax):
    z0=complex(x0,y0)
    z = z0
    for t in range(kmax):
        if abs(z) > 2.0:
            return t
        z = z * z + z0
    return kmax

MAX = 500

n=4096
xc=-0.5
yc=0
sizze=2
```

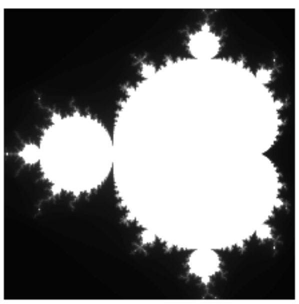
In [23]:

```
%%time
array = zeros((n,n),dtype=np.uint8)
for col in range(n):
    for row in range(n):
        x0 = xc - (sizze / 2) + (sizze * col / n)
        y0 = yc - (sizze / 2) + (sizze * row / n)
        result=mandel_set(x0,y0, MAX)
        array[row, col] = result
```

```
CPU times: user 5min 59s, sys: 601 ms, total: 5min 59s
Wall time: 6min
```

In [24]:

```
imshow(array,vmin=0,vmax=MAX/5,origin='lower',cmap=cm.gray)
gca().axis('off')
imsave("Problem5_serrial.png",array,vmin=0,vmax=MAX/5,cmap=cm.gray,dpi=dpi,origin='lower')

```



In [25]:

```
from numpy import *
```

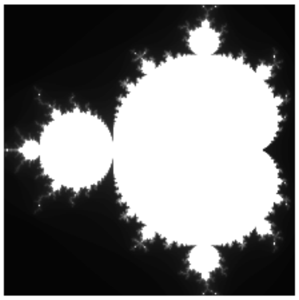
In [26]:

```
%%time
data=[]
arr = zeros((n,n),dtype=np.uint8)
for col in range(n):
    for row in range(n):
        x0 = xc - (sizze / 2) + (sizze * col / n)
        y0 = yc - (sizze / 2) + (sizze * row / n)
        data.append((x0,y0,MAX))

with mp.Pool(processes=12) as pool:
    result=pool.starmap(mandel_set,data)
    pool.close()
    pool.join()
result=array(result)
arr=result.reshape((n,n)).T

```

```
CPU times: user 15.4 s, sys: 3.5 s, total: 18.9 s
Wall time: 1min 30s
```

In [27]:
```
%%time
imshow(arr,vmin=0,vmax=MAX/5,origin='lower',cmap=cm.gray)
gca().axis('off')


imsave("Problem5_parrallel.png",arr,vmin=0,vmax=MAX/5,cmap=cm.gray,dpi=dpi,origin='lower')
```

```
CPU times: user 1.44 s, sys: 196 ms, total: 1.64 s
Wall time: 1.64 s
```



DISCUSSION

There is a close relationship between the mandelbrot set and the Julia set. Each point in the Mandelbrot set corresponds to a Julia set. The Mandelbrot set is a map of the Julia sets. Both sets are fractals, meaning that they have an infinite level of detail and can be zoomed in on infinitely.

In [ ]:
```

```