High Performance Computing Assignment 1

Task 1.1

In [1]:
```
from timeit import default_timer as timer
import time
```

time.time( )

In [2]:
```
import numpy as np
def checktick1():
    M = 200
    timesfound = np.empty((M,))
    for i in range(M):
        t1 =  time.time() # get timestamp from timer
        t2 = time.time() # get timestamp from timer
        while (t2 - t1) < 1e-16: # if zero then we are below clock granul
            t2 = time.time() # get timestamp from timer
        t1 = t2 # this is outside the loop
        timesfound[i] = t1 # record the time stamp
    minDelta = 1000000
    Delta = np.diff(timesfound) # it should be cast to int only when need
    minDelta = Delta.min()
    return minDelta
```

In [3]:
```
checktick1()
```

Out[3]:
```
np.float64(7.152557373046875e-07)
```

timeit( )

In [4]:
```
import numpy as np
def checktick2():
    M = 200
    timesfound = np.empty((M,))
    for i in range(M):
        t1 =  timer() # get timestamp from timer
        t2 = timer() # get timestamp from timer
        while (t2 - t1) < 1e-16: # if zero then we are below clock granul
            t2 = timer() # get timestamp from timer
        t1 = t2 # this is outside the loop
        timesfound[i] = t1 # record the time stamp
    minDelta = 1000000
    Delta = np.diff(timesfound) # it should be cast to int only when need
    minDelta = Delta.min()
    return minDelta
```

In [5]:
```
checktick2()
```

Out[5]: np.float64(2.0797597244381905e-07)

time.time_ns( )

In [6]:
```
import numpy as np
def checktick3():
    M = 200
    timesfound = np.empty((M,))
    for i in range(M):
        t1 =  time.time_ns() # get timestamp from timer
        t2 = time.time_ns() # get timestamp from timer
        while (t2 - t1) < 1e-16: # if zero then we are below clock granul
            t2 = time.time_ns() # get timestamp from timer
        t1 = t2 # this is outside the loop
        timesfound[i] = t1 # record the time stamp
    minDelta = 1000000
    Delta = np.diff(timesfound) # it should be cast to int only when need
    minDelta = Delta.min()
    return minDelta
```

In [7]:
```
checktick3()*1e-9
```

Out[7]: np.float64(7.680000000000001e-07)

Task 1.2

Decorator

In [11]:
```
from functools import wraps
import statistics
```

In [13]:
```
"""Julia set generator without optional PIL-based image drawing"""
import time
from functools import wraps

# area of complex space to investigate
x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8
c_real, c_imag = -0.62772, -.42193

# decorator to time
def timefn(fn):
    times = []
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = timer()
        result = fn(*args, **kwargs)
        t2 = timer()
        time_elapsed = t2 - t1
        times.append(time_elapsed)
        print(f"@timefn: {fn.__name__} took {t2 - t1} seconds")
        return result


    def get_stats():
        if times:
```

```
            avg = statistics.mean(times)
            std_dev = statistics.stdev(times)
            return avg, std_dev
        else:
            return None, None

    measure_time.get_stats = get_stats
    return measure_time

@timefn
def calc_pure_python(desired_width, max_iterations):
    """Create a list of complex coordinates (zs) and complex parameters (
    build Julia set"""
    x_step = (x2 - x1) / desired_width
    y_step = (y1 - y2) / desired_width
    x = []
    y = []
    ycoord = y2
    while ycoord > y1:
        y.append(ycoord)
        ycoord += y_step
    xcoord = x1
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step
    # build a list of coordinates and the initial condition for each cell
    # Note that our initial condition is a constant and could easily be r
    # we use it to simulate a real-world scenario with several inputs to
    # function
    zs = []
    cs = []
    for ycoord in y:
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))

    print("Length of x:", len(x))
    print("Total elements:", len(zs))
    # start_time = timer()
    output = calculate_z_serial_purepython(max_iterations, zs, cs)
    # end_time = timer()
    # secs = end_time - start_time
    # print(calculate_z_serial_purepython.__name__ + " took", secs, "seco

    # This sum is expected for a 1000^2 grid with 300 iterations
    # It ensures that our code evolves exactly as we'd intended
    assert sum(output) == 33219980

@timefn
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
```

```
    return output

if __name__ == "__main__":
    # Calculate the Julia set using a pure Python solution with
    # reasonable defaults for a laptop
    num_runs = 10
    for _ in range(num_runs):
        calc_pure_python(desired_width=1000, max_iterations=300) # Reduce

    calc_avg, calc_std = calc_pure_python.get_stats()
    z_avg, z_std = calculate_z_serial_purepython.get_stats()

    print("\n--- Statistics ---")
    if calc_avg:
        print(f"calc_pure_python: Average = {calc_avg:.4f} s, Standard De
    if z_avg:
        print(f"calculate_z_serial_purepython: Average = {z_avg:.4f} s, S
```

```
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.6986198750091717 seconds
@timefn: calc_pure_python took 2.849360332998913 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.294878709013574 seconds
@timefn: calc_pure_python took 2.4328963329899125 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.240699832967948 seconds
@timefn: calc_pure_python took 2.3754435420269147 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.3812026670202613 seconds
@timefn: calc_pure_python took 2.515382374986075 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.2198940420057625 seconds
@timefn: calc_pure_python took 2.3566730829770677 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.2302856250316836 seconds
@timefn: calc_pure_python took 2.364553541992791 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.236042583012022 seconds
@timefn: calc_pure_python took 2.375761125003919 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.268640583031811 seconds
@timefn: calc_pure_python took 2.4037159999716096 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.338958750013262 seconds
@timefn: calc_pure_python took 2.475540583021939 seconds
Length of x: 1000
Total elements: 1000000
@timefn: calculate_z_serial_purepython took 2.2417154999566264 seconds
@timefn: calc_pure_python took 2.375050583970733 seconds

--- Statistics ---
calc_pure_python: Average = 2.4524 s, Standard Deviation = 0.1489 s
calculate_z_serial_purepython: Average = 2.3151 s, Standard Deviation = 0.
1445 s
```

The standard deviation of the 10 attempts is 0.1489s and 0.1445 respectively, which are significantly larger than the granualiry that we have caculated using the timeit module (2.079-07s). This suggests that our experiment is capturing real variations in execution time but not statistically affected by the physical limit of the timer (the granularity varaition).

Task 1.3

Using cProfile

In [15]:
```
!python -m cProfile -s cumulative JuliaSet.py
```

```
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.8358170986175537 seconds
         36221995 function calls in 4.067 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.067    4.067 {built-in method builtins.ex
ec}
        1    0.010    0.010    4.067    4.067 JuliaSet.py:1(<module>)
        1    0.177    0.177    4.057    4.057 JuliaSet.py:21(calc_pure_pyt
hon)
        1    3.057    3.057    3.836    3.836 JuliaSet.py:59(calculate_z_s
erial_purepython)
 34219980    0.778    0.000    0.778    0.000 {built-in method builtins.ab
s}
  2002000    0.042    0.000    0.042    0.000 {method 'append' of 'list' o
bjects}
        1    0.003    0.003    0.003    0.003 {built-in method builtins.su
m}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.pr
int}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lspro
f.Profiler' objects}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.le
n}
```

Generate a profile.stats file

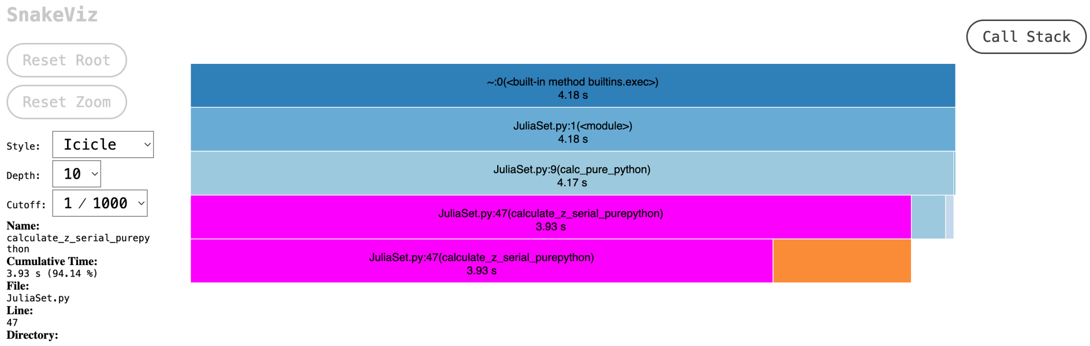
In [54]:
```
!python -m cProfile -o profile.stats JuliaSet.py
```

```
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.934892177581787 seconds
```

Visualize using SnakeViz

In [55]:
```
!python -m snakeviz profile.stats --server
```

```
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to exit
http://127.0.0.1:8080/snakeviz/%2FUsers%2Ffranklin%2FCodes%2FCOMP%2FHigh%2
0Performance%20Computing%20%28KTH%29%2Fprofile.stats
^C

Bye!
```



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	3.18	3.18	3.935	3.935	JuliaSet.py:47(calculate_z_serial_purepython)
34219980	0.7552	2.207e-08	0.7552	2.207e-08	~:0(<built-in method builtins.abs>)
1	0.1866	0.1866	4.17	4.17	JuliaSet.py:9(calc_pure_python)
2002000	0.04554	2.275e-08	0.04554	2.275e-08	~:0(<method 'append' of 'list' objects>)
1	0.009579	0.009579	4.18	4.18	JuliaSet.py:1(<module>)
1	0.002994	0.002994	0.002994	0.002994	~:0(<built-in method builtins.sum>)
3	0.0001175	3.915e-05	0.0001175	3.915e-05	~:0(<built-in method builtins.print>)
2	1.167e-06	5.835e-07	1.167e-06	5.835e-07	~:0(<built-in method time.time>)
1	1.001e-06	1.001e-06	4.18	4.18	~:0(<built-in method builtins.exec>)
4	8.33e-07	2.083e-07	8.33e-07	2.083e-07	~:0(<built-in method builtins.len>)
1	7.08e-07	7.08e-07	7.08e-07	7.08e-07	~:0(<method 'disable' of '_lsprof.Profiler' objects>)

Showing 1 to 11 of 11 entries

SnakeViz Docs

Using line_profiler

In [26]:
```
!python -m kernprof -l JuliaSet_profiler.py
```

```
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 21.05274271965027 seconds
Wrote profile results to JuliaSet_profiler.py.lprof
Inspect results with:
python -m line_profiler -rmt "JuliaSet_profiler.py.lprof"
```

In [27]:
```
!python -m line_profiler JuliaSet_profiler.py.lprof
```

```
Timer unit: 1e-06 s

Total time: 21.454 s
File: JuliaSet_profiler.py
Function: calc_pure_python at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def calc_pure_python(desi
red_width, max_iterations):
    11                                               """Create a list of c
omplex coordinates (zs) and complex parameters (cs),
    12                                               build Julia set"""
    13         1          1.0      1.0      0.0      x_step = (x2 - x1) /
desired_width
    14         1          0.0      0.0      0.0      y_step = (y1 - y2) /
desired_width
    15         1          0.0      0.0      0.0      x = []
    16         1          0.0      0.0      0.0      y = []
    17         1          0.0      0.0      0.0      ycoord = y2
    18      1001        108.0      0.1      0.0      while ycoord > y1:
    19      1000        107.0      0.1      0.0          y.append(ycoord)
    20      1000         86.0      0.1      0.0          ycoord += y_step
    21         1          0.0      0.0      0.0      xcoord = x1
    22      1001        122.0      0.1      0.0      while xcoord < x2:
    23      1000        109.0      0.1      0.0          x.append(xcoord)
    24      1000         87.0      0.1      0.0          xcoord += x_step
    25                                               # build a list of coo
rdinates and the initial condition for each cell.
    26                                               # Note that our initi
al condition is a constant and could easily be removed,
    27                                               # we use it to simula
te a real-world scenario with several inputs to our
    28                                               # function
    29         1          0.0      0.0      0.0      zs = []
    30         1          0.0      0.0      0.0      cs = []
    31      1001         99.0      0.1      0.0      for ycoord in y:
    32   1001000      79277.0      0.1      0.4          for xcoord in x:
    33   1000000     149995.0      0.1      0.7              zs.append(com
plex(xcoord, ycoord))
    34   1000000     168225.0      0.2      0.8              cs.append(com
plex(c_real, c_imag))
    35
    36         1         33.0     33.0      0.0      print("Length of x:",
len(x))
    37         1          2.0      2.0      0.0      print("Total element
s:", len(zs))
    38         1          2.0      2.0      0.0      start_time = time.tim
e()
    39         1   21052741.0    2e+07     98.1      output = calculate_z_
serial_purepython(max_iterations, zs, cs)
    40         1          1.0      1.0      0.0      end_time = time.time
()
    41         1          1.0      1.0      0.0      secs = end_time - sta
rt_time
    42         1         16.0     16.0      0.0      print(calculate_z_ser
ial_purepython.__name__ + " took", secs, "seconds")
    43
    44                                               # This sum is expecte
```

```
d for a 1000^2 grid with 300 iterations
    45                                                  # It ensures that our
code evolves exactly as we'd intended
    46          1       3007.0    3007.0      0.0      assert sum(output) ==
33219980

Total time: 11.5627 s
File: JuliaSet_profiler.py
Function: calculate_z_serial_purepython at line 48

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48                                           @profile
    49                                           def calculate_z_serial_pu
repython(maxiter, zs, cs):
    50                                               """Calculate output l
ist using Julia update rule"""
    51          1        500.0    500.0      0.0      output = [0] * len(z
s)
    52    1000001      98083.0      0.1      0.8      for i in range(len(z
s)):
    53    1000000      70791.0      0.1      0.6          n = 0
    54    1000000      96073.0      0.1      0.8          z = zs[i]
    55    1000000      75024.0      0.1      0.6          c = cs[i]
    56   34219980    5293090.0      0.2     45.8          while abs(z) < 2
and n < maxiter:
    57   33219980    3074613.0      0.1     26.6              z = z * z + c
    58   33219980    2746507.0      0.1     23.8              n += 1
    59    1000000     108001.0      0.1      0.9          output[i] = n
    60          1          1.0      1.0      0.0      return output
```

Without the Profiler

In [24]:
```
!python JuliaSet.py
```

```
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.179577112197876 seconds
```

With the cProfiler, calculate_z_serial_purepython took 3.83581s, with line_profiler, it took 11.9712 s. Without any profiler, it took 2.1796s, which is faster than both. It shows that two profiler have added a significant overhead to the function, while the overhead for line_profiler is more significant.

Task 1.4

In [30]:
```
!python -m memory_profiler JuliaSet_memory.py
```

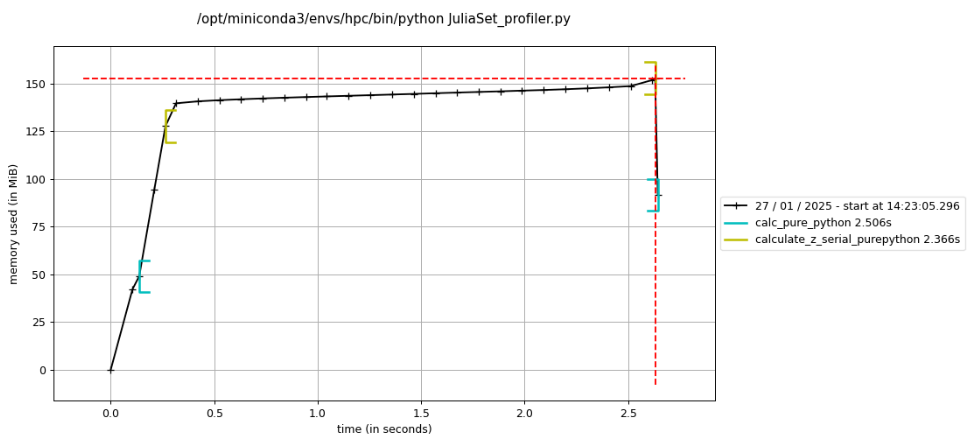
```
Length of x: 100
Total elements: 10000
calculate_z_serial_purepython took 10.960868120193481 seconds
Filename: JuliaSet_memory.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9   48.859 MiB   48.859 MiB           1   @profile
    10                                         def calc_pure_python(desire
d_width, max_iterations):
    11                                             """Create a list of com
plex coordinates (zs) and complex parameters (cs),
    12                                             build Julia set"""
    13   48.859 MiB    0.000 MiB           1       x_step = (x2 - x1) / de
sired_width
    14   48.859 MiB    0.000 MiB           1       y_step = (y1 - y2) / de
sired_width
    15   48.859 MiB    0.000 MiB           1       x = []
    16   48.859 MiB    0.000 MiB           1       y = []
    17   48.859 MiB    0.000 MiB           1       ycoord = y2
    18   48.859 MiB    0.000 MiB         101       while ycoord > y1:
    19   48.859 MiB    0.000 MiB         100           y.append(ycoord)
    20   48.859 MiB    0.000 MiB         100           ycoord += y_step
    21   48.859 MiB    0.000 MiB           1       xcoord = x1
    22   48.859 MiB    0.000 MiB         101       while xcoord < x2:
    23   48.859 MiB    0.000 MiB         100           x.append(xcoord)
    24   48.859 MiB    0.000 MiB         100           xcoord += x_step
    25                                             # build a list of coord
inates and the initial condition for each cell.
    26                                             # Note that our initial
condition is a constant and could easily be removed,
    27                                             # we use it to simulate
a real-world scenario with several inputs to our
    28                                             # function
    29   48.859 MiB    0.000 MiB           1       zs = []
    30   48.859 MiB    0.000 MiB           1       cs = []
    31   49.547 MiB    0.000 MiB         101       for ycoord in y:
    32   49.547 MiB    0.000 MiB       10100           for xcoord in x:
    33   49.547 MiB    0.047 MiB       10000               zs.append(compl
ex(xcoord, ycoord))
    34   49.547 MiB    0.641 MiB       10000               cs.append(compl
ex(c_real, c_imag))
    35
    36   49.547 MiB    0.000 MiB           1       print("Length of x:", l
en(x))
    37   49.547 MiB    0.000 MiB           1       print("Total element
s:", len(zs))
    38   49.547 MiB    0.000 MiB           1       start_time = time.time
()
    39   49.781 MiB   49.781 MiB           1       output = calculate_z_se
rial_purepython(max_iterations, zs, cs)
    40   49.781 MiB    0.000 MiB           1       end_time = time.time()
    41   49.781 MiB    0.000 MiB           1       secs = end_time - start
_time
    42   49.781 MiB    0.000 MiB           1       print(calculate_z_seria
l_purepython.__name__ + " took", secs, "seconds")
    43
    44                                             # This sum is expected
for a 1000^2 grid with 300 iterations
    45                                             # It ensures that our c
```

```
ode evolves exactly as we'd intended
    46                                                   # assert sum(output) ==
33219980


Filename: JuliaSet_memory.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
    48   49.547 MiB   49.547 MiB           1   @profile
    49                                         def calculate_z_serial_pure
python(maxiter, zs, cs):
    50                                             """Calculate output lis
t using Julia update rule"""
    51   49.547 MiB    0.000 MiB           1       output = [0] * len(zs)
    52   49.781 MiB    0.000 MiB       10001       for i in range(len(z
s)):
    53   49.781 MiB    0.000 MiB       10000           n = 0
    54   49.781 MiB    0.000 MiB       10000           z = zs[i]
    55   49.781 MiB    0.000 MiB       10000           c = cs[i]
    56   49.781 MiB    0.031 MiB      344236           while abs(z) < 2 an
d n < maxiter:
    57   49.781 MiB    0.078 MiB      334236               z = z * z + c
    58   49.781 MiB    0.125 MiB      334236               n += 1
    59   49.781 MiB    0.000 MiB       10000           output[i] = n
    60   49.781 MiB    0.000 MiB           1       return output
```

In [39]:
```
!python -m mprof run JuliaSet_profiler.py
```

```
mprof.py: Sampling memory every 0.1s
running new process
running as a Python program...
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.3665859699249268 seconds
```

In [40]:
```
!python -m mprof plot -o memory_profile.png mprofile_20250127142305.dat
```



Overhead by memory_profiler and mprof

For memory profiler, it takes 10.96s for the calculate_z_serial_purepython function (100x100), the prof takes 2.37s for (1000x1000 grid), while the one without profiler takes only 2.17s (for 1000x1000 grid). It shows that mprof samples by time but not by line and has a significantly low overhead that barely impacts the runtime of the code.

Task 2.1

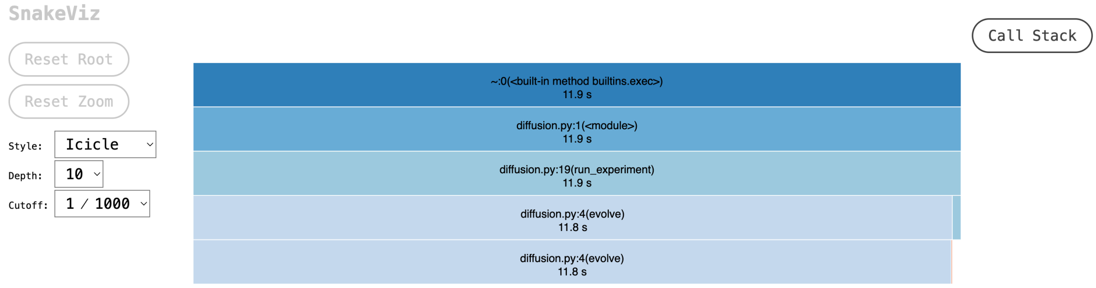
In [43]:
```
!python -m cProfile -s cumulative diffusion.py
```

```
         205 function calls in 11.833 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   11.833   11.833 {built-in method builtins.ex
ec}
        1    0.002    0.002   11.833   11.833 diffusion.py:1(<module>)
        1    0.131    0.131   11.831   11.831 diffusion.py:19(run_experime
nt)
      100   11.683    0.117   11.700    0.117 diffusion.py:4(evolve)
      100    0.016    0.000    0.016    0.000 diffusion.py:6(<listcomp>)
        1    0.000    0.000    0.000    0.000 diffusion.py:22(<listcomp>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lspro
f.Profiler' objects}
```

In [45]:
```
!python -m cProfile -o profile_task2.stats diffusion.py
```

In [56]:
```
!python -m snakeviz profile_task2.stats --server
```

```
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to exit
http://127.0.0.1:8080/snakeviz/%2FUsers%2Ffranklin%2FCodes%2FCOMP%2FHigh%2
0Performance%20Computing%20%28KTH%29%2Fprofile_task2.stats
^C

Bye!
```



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
100	11.74	0.1174	11.76	0.1176	diffusion.py:4(evolve)
1	0.1351	0.1351	11.89	11.89	diffusion.py:19(run_experiment)
100	0.0168	0.000168	0.0168	0.000168	diffusion.py:6(<listcomp>)
1	0.001636	0.001636	11.9	11.9	diffusion.py:1(<module>)
1	0.0003993	0.0003993	0.0003993	0.0003993	diffusion.py:22(<listcomp>)
1	1.833e-06	1.833e-06	11.9	11.9	~:0(<built-in method builtins.exec>)
1	8.34e-07	8.34e-07	8.34e-07	8.34e-07	~:0(<method 'disable' of '_lsprof.Profiler' objects>)

Showing 1 to 7 of 7 entries

SnakeViz Docs

line_profiler

In [48]:
```
!python -m kernprof -l diffusion_profile.py
```

```
Wrote profile results to diffusion_profile.py.lprof
Inspect results with:
python -m line_profiler -rmt "diffusion_profile.py.lprof"
```

In [49]:
```
!python -m line_profiler diffusion_profile.py.lprof
```

```
Timer unit: 1e-06 s

Total time: 41.1199 s
File: diffusion_profile.py
Function: evolve at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def evolve(grid, dt, D=1.
0):
     5       100         66.0      0.7      0.0      xmax, ymax = grid_sha
pe
     6       100      18153.0    181.5      0.0      new_grid = [[0.0] * y
max for x in range(xmax)]
     7     64100       6608.0      0.1      0.0      for i in range(xmax):
     8  41024000    3728457.0      0.1      9.1          for j in range(ym
ax):
     9  40960000    2771096.0      0.1      6.7              grid_xx = (
    10  40960000   11196951.0      0.3     27.2                  grid[(i +
1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
    11                                                       )
    12  40960000    2765853.0      0.1      6.7              grid_yy = (
    13  40960000   11774423.0      0.3     28.6                  grid[i]
[(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
    14                                                       )
    15  40960000    8858266.0      0.2     21.5              new_grid[i]
[j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
    16       100         33.0      0.3      0.0      return new_grid

Total time: 63.7211 s
File: diffusion_profile.py
Function: run_experiment at line 18

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    18                                           @profile
    19                                           def run_experiment(num_it
erations):
    20                                               # Setting up initial
conditions
    21         1          0.0      0.0      0.0      xmax, ymax = grid_sha
pe
    22         1        607.0    607.0      0.0      grid = [[0.0] * ymax
for x in range(xmax)]
    23
    24                                               # These initial condi
tions are simulating a drop of dye in the middle of our
    25                                               # simulated region
    26         1          0.0      0.0      0.0      block_low = int(grid_
shape[0] * 0.4)
    27         1          0.0      0.0      0.0      block_high = int(grid
_shape[0] * 0.5)
    28        65          5.0      0.1      0.0      for i in range(block_
low, block_high):
    29      4160        337.0      0.1      0.0          for j in range(bl
ock_low, block_high):
    30      4096        390.0      0.1      0.0              grid[i][j] =
0.005
    31
```

```
    32                                                   # Evolve the initial
conditions
    33       101         17.0      0.2      0.0      for i in range(num_it
erations):
    34       100   63719787.0 637197.9    100.0          grid = evolve(gri
d, 0.1)
```

Task 2.2

Memory_Profiler

In [50]:
```
!python -m memory_profiler diffusion_profile.py
```

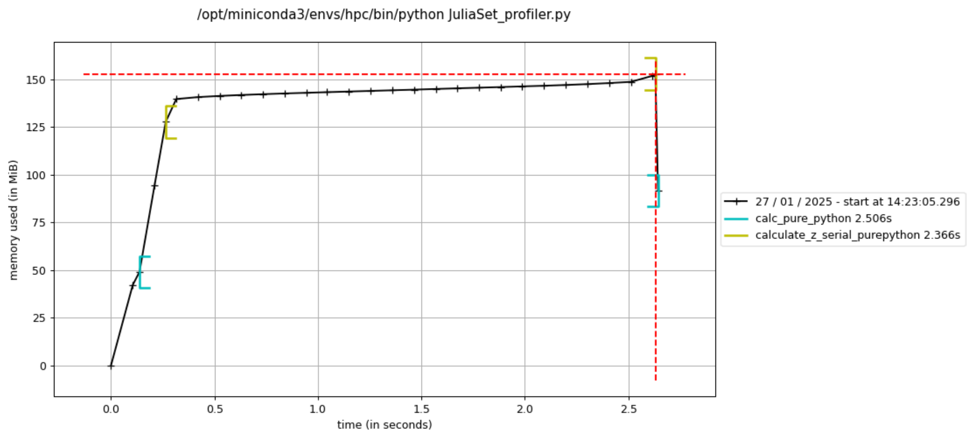
```
Filename: diffusion_profile.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     3   91.344 MiB 5253.969 MiB         100   @profile
     4                                         def evolve(grid, dt, D=1.
0):
     5   91.344 MiB -3736.500 MiB         100       xmax, ymax = grid_shap
e
     6   91.344 MiB -2340014.578 MiB       64300       new_grid = [[0.0] *
ymax for x in range(xmax)]
     7   96.344 MiB -2741546.859 MiB       64100       for i in range(xma
x):
     8   96.344 MiB -1754952840.828 MiB    41024000           for j in ran
ge(ymax):
     9   96.344 MiB -1752214609.781 MiB    40960000               grid_xx
= (
    10   96.344 MiB -1752214725.219 MiB    40960000                   grid
[(i + 1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
    11                                                             )
    12   96.344 MiB -1752214894.812 MiB    40960000               grid_yy
= (
    13   96.344 MiB -1752214772.141 MiB    40960000                   grid
[i][(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
    14                                                             )
    15   96.344 MiB -1752215011.578 MiB    40960000               new_grid
[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
    16   96.344 MiB -4354.141 MiB         100       return new_grid


Filename: diffusion_profile.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
    18   49.234 MiB   49.234 MiB           1   @profile
    19                                         def run_experiment(num_iter
ations):
    20                                             # Setting up initial co
nditions
    21   49.234 MiB    0.000 MiB           1       xmax, ymax = grid_shape
    22   52.328 MiB    3.094 MiB         643       grid = [[0.0] * ymax fo
r x in range(xmax)]
    23
    24                                             # These initial conditi
ons are simulating a drop of dye in the middle of our
    25                                             # simulated region
    26   52.328 MiB    0.000 MiB           1       block_low = int(grid_sh
ape[0] * 0.4)
    27   52.328 MiB    0.000 MiB           1       block_high = int(grid_s
hape[0] * 0.5)
    28   52.328 MiB    0.000 MiB          65       for i in range(block_lo
w, block_high):
    29   52.328 MiB    0.000 MiB        4160           for j in range(bloc
k_low, block_high):
    30   52.328 MiB    0.000 MiB        4096               grid[i][j] = 0.
005
    31
    32                                             # Evolve the initial co
nditions
    33   91.344 MiB -3787.469 MiB         101       for i in range(num_ite
```

```
rations):
    34   91.344 MiB 5241.969 MiB         100           grid = evolve(grid,
0.1)
```

In [51]:
```
!python -m mprof run diffusion_profile.py
```

```
mprof.py: Sampling memory every 0.1s
running new process
running as a Python program...
```

In [52]:
```
!python -m mprof plot -o memory_profile_task2.png mprofile_20250127142305
```



Bonus Exercise

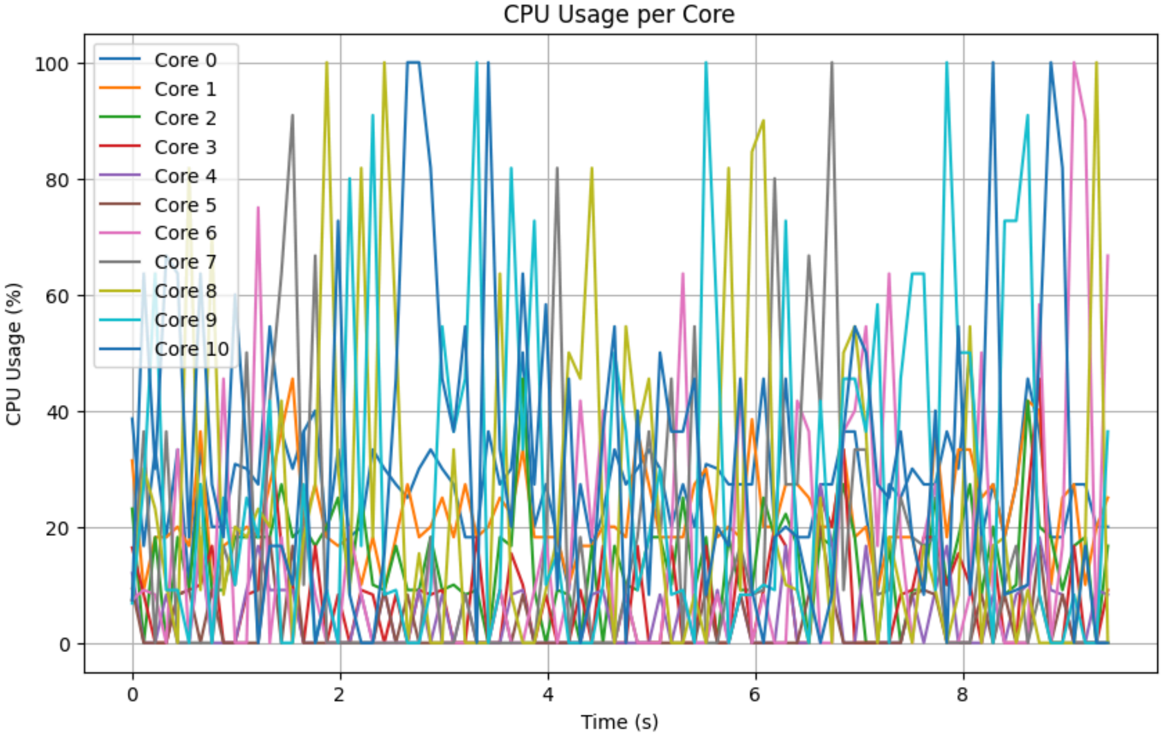
In [61]:
```python
import psutil
import time
import threading
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

class CPUProfiler:
    def __init__(self):
        self.cpu_data = []
        self.start_time = None
        self._stop_event = threading.Event()
        self._thread = None

    def start(self):
        self.cpu_data = []
        self.start_time = time.time()
        self._stop_event.clear()
        self._thread = threading.Thread(target=self._record_cpu_usage)
        self._thread.start()

    def stop(self):
        self._stop_event.set()
        self._thread.join()
        end_time = time.time()
        elapsed_time = end_time - self.start_time
        print(f"Profiling took {elapsed_time:.4f} seconds.")
```

```
    def _record_cpu_usage(self):
        while not self._stop_event.is_set():
            self.record()
            time.sleep(0.1)

    def record(self):
        cpu_percent = psutil.cpu_percent(interval=None, percpu=True)
        current_time = time.time() - self.start_time
        self.cpu_data.append((current_time, cpu_percent))

    def profile(self, func, *args, **kwargs):
        self.start()
        start_exec = time.time()
        result = func(*args, **kwargs)
        end_exec = time.time()
        self.stop()
        exec_time = end_exec - start_exec
        print(f"Execution took {exec_time:.4f} seconds.")
        return result

    def plot(self, filename="cpu_profile.png"):
        if not self.cpu_data:
            print("No profiling data recorded.")
            return

        df = pd.DataFrame(self.cpu_data, columns=["Time", "CPU Usage"])
        num_cores = len(df["CPU Usage"][0])
        plt.figure(figsize=(10, 6))
        for core in range(num_cores):
            core_usage = [usage[core] for usage in df["CPU Usage"]]
            plt.plot(df["Time"], core_usage, label=f"Core {core}")

        plt.xlabel("Time (s)")
        plt.ylabel("CPU Usage (%)")
        plt.title("CPU Usage per Core")
        plt.legend()
        plt.grid(True)
        plt.savefig(filename)
        plt.show()

    def summary(self):
        if not self.cpu_data:
            print("No profiling data recorded.")
            return
        df = pd.DataFrame(self.cpu_data, columns=["Time", "CPU Usage"])
        num_cores = len(df["CPU Usage"][0])
        summary_data = []
        for core in range(num_cores):
            core_usage = [usage[core] for usage in df["CPU Usage"]]
            summary_data.append({"Core": core, "Average Usage": np.mean(c
        summary_df = pd.DataFrame(summary_data)
        print(summary_df)

# Example usage:
profiler = CPUProfiler()
```

This creates a CPUProfiler class to measure and visualize CPU usage. It uses psutil to sample CPU usage over time, storing the time and per-core usage in cpu_data. The profile method runs a given function while recording CPU usage. The plot method then creates a graph showing CPU usage per core over time using matplotlib, and the summary method calculates and prints average and maximum CPU usage per core using pandas. Essentially, it observe how much CPU a function uses and how that usage is distributed across CPU cores.

In [62]:
```
from diffusion import run_experiment

num_iterations = 100
grid = profiler.profile(run_experiment, num_iterations)
profiler.plot()
profiler.summary()
```

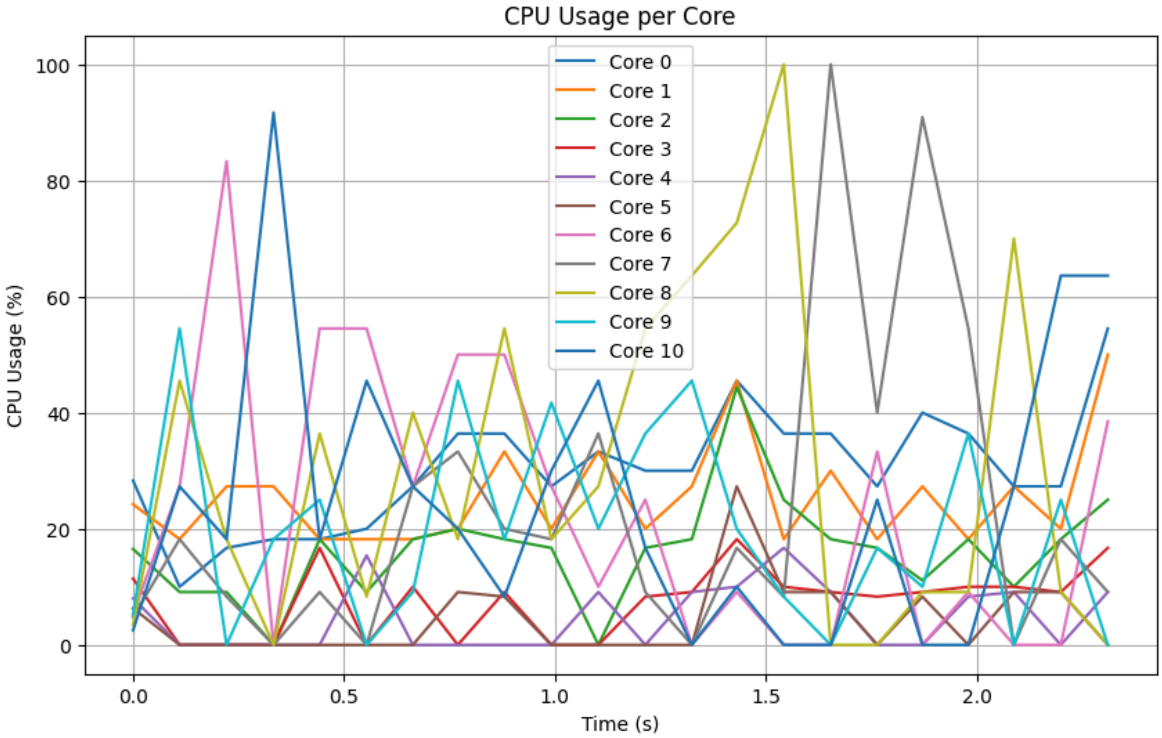
```
Profiling took 9.5005 seconds.
Execution took 9.4527 seconds.
```



	Core	Average Usage	Max Usage
0	0	27.706977	54.5
1	1	22.110465	45.5
2	2	14.155814	45.5
3	3	8.108140	45.5
4	4	4.667442	27.3
5	5	2.733721	20.0
6	6	15.543023	100.0
7	7	15.884884	100.0
8	8	24.202326	100.0
9	9	27.280233	100.0
10	10	31.543023	100.0

In [63]:
```
from JuliaSet import calc_pure_python

profiler2 = CPUProfiler()
desired_width = 1000
max_iterations = 300
grid = profiler2.profile(calc_pure_python, desired_width, max_iterations)
profiler2.plot()
profiler2.summary()
```

```
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 2.189697027206421 seconds
Profiling took 2.4161 seconds.
Execution took 2.3394 seconds.
```



	Core	Average Usage	Max Usage
0	0	30.145455	54.5
1	1	25.463636	50.0
2	2	16.218182	44.4
3	3	7.504545	18.2
4	4	4.722727	16.7
5	5	4.350000	27.3
6	6	22.931818	83.3
7	7	23.745455	100.0
8	8	29.918182	100.0
9	9	19.809091	54.5
10	10	24.577273	91.7