AME 60614: Numerical Methods
Fall 2021

Problem Set 0

Qihao Zhuo

1 Code Documentation

1. Download the document `sample_ps0.tex` and `sample_figure.eps` from the course Sakai page. Rename the file `lastname_firstname_ps0.tex`, and edit the report to add your name. Compile it using `pdflatex` or the LaTeX editor of your choice.
2. Document your response to Problem 2 in your report. Use appropriately highlighted code listings for codes requested in Problem 2. Print a copy and bring it to class on the due date.

2 Computing in Parallel

2.1 Assignment

2.1.1 a

The scripting language used is Matlab and the script is shown below. The sentences of plot or file can be commented out if there is no need for plot or output file.

```
clear
clc
tic
nx=10000;
ny=10000;
dx=2.5/(nx-1);
dy=4.0/(ny-1);
Re_plot = -2:dx:0.5;
Im_plot = -2:dy:2;

hold on

f1 = fopen('exp.txt','w');
for Re = Re_plot
    for Im = Im_plot
        c_p = Re + Im*1i;
        if(ms_check(c_p, 10))
            plot(Re, Im, 'r.');
            fprintf(f1,'%6.2f %6.2f\n',real(c_p),imag(c_p));
        end
    end
end
```

Table 1: Time cost of Matlab while only writing output file

niter	10	100	1000
time	524.60	501.68	927.91

```
fclose(f1);
toc

function [logi] = ms_check(c, niter)
z=0;
count=0;
while abs(z) < 2 && count < niter
    z = z^2 + c;
    count=count+1;
end
if count == niter
    logi=true;
else
    logi=false;
end
end
```

Fig. 1 is plotted by Matlab with a lower resolution.

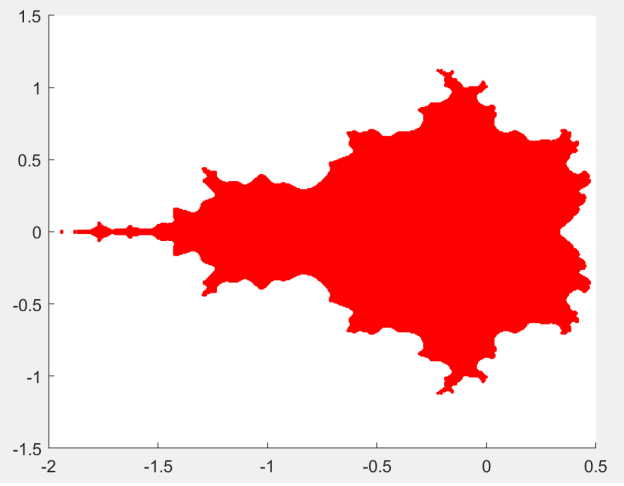

Figure 1: Figure of Mandelbrot Set by Matlab

Since compiled language usually could not plot, Tab. 1 shows the time cost of Matlab scripts under $niter = 10, 100, 1000$ and will be compared with results of compiled language in Sec. 2.1.2.

The decrease between $niter = 10$ and $niter = 100$ might because when $niter = 10$, the number of points written to the output file is much more than $niter = 100$. The I/O operation would take a lot of time.

Table 2: Comparison of time cost between Matlab and Fortran while only run iterations

niter	10	100	1000
Matlab	524.60	501.68	927.91
Fortran	44.21	53.61	235.97

2.1.2 b

Since in Fortran there is intrinsic *complex* data type, the compile language used is Fortran. To compare with results in Sec. 2.1.1, I also let the Fortran program write the output file directly. The comparison of time cost is shown in Tab. 2.

Obviously Fortran performs better than Matlab. And in fact the Fortran can also write output file on my laptop, even for the 10000x10000 grid. Therefore compiled languages are much advantageous in such loops than scripting languages. The code used in this section is shown below.

```
program MSet
    implicit none
    COMPLEX(kind=8) :: c,z
    REAL(kind=8) :: xmin,xmax,ymin,ymax,dx,dy
    INTEGER(kind=8) :: nx,ny,niter,i
    INTEGER(kind=8) :: cx,cy
    REAL(kind=8) :: t_begin,t_end

    OPEN(unit=1,file='MSet.in')
    READ(1,*) xmin,xmax,nx,ymin,ymax,ny,niter
    dx = (xmax-xmin)/(nx-1)
    dy = (ymax-ymin)/(ny-1)
    c = complex(xmin,ymin)
    OPEN(unit=2,file='dyIntMSet.out')
    CALL cpu_time(t_begin)
    do cx=1,nx-1
        do cy=1,ny-1
            z = (0,0)
            i = 1
            do while (abs(z)<=2 .AND. i<=niter)
                z = z**2 + c
                i = i + 1
            end do
            if ( i >= niter ) then
              WRITE(2,*) real(c), AIMAG(c)
            end if
            c = complex(real(c),ymin+cy*dy)
        end do
        c = complex(xmin+cx*dx,ymin)
    end do
    CALL cpu_time(t_end)
    WRITE(*,*) "time", t_end-t_begin, "case", xmin,xmax,nx,ymin,ymax,ny,niter
end program MSet
```

With points from the Fortran program, Fig. 2 is also a figure of the Mandelbrot set.

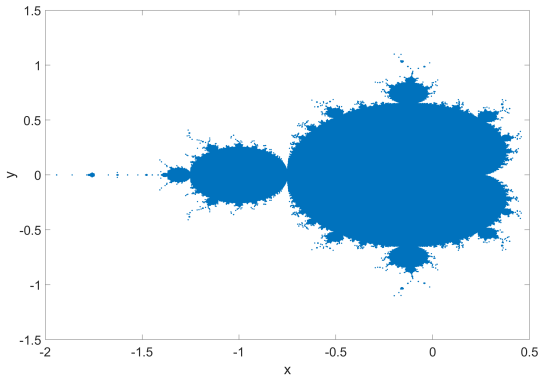

Figure 2: Figure of Mandelbrot Set by data from Fortran

Table 3: Comparison between different methods of output

Thread	Method 1	Method 2	Method 3
2	91.29	91.80	86.90
4	60.21	53.97	52.83

2.1.3 c

To plot the figure of Mandelbrot Set, there are three methods. The first is to write c to the output file directly in each loop, which was used in Sec. 2.1.2. But it is low efficient because there are also too much I/O operation on the output file. The second is to record c in an array and write all picked points to the output file at the end of code. The third is to record intergers indicating c and compute corresponding c when writing the outpu file. The comparison between timecost of them are shown in Tab. 3 with cases under 2 threads and 4 threads for $niter = 1000$.

So in that section, the third method was chosen. The code is shown below.

```
program MSet
    use omp_lib
    implicit none
    COMPLEX(kind=8) :: c,z
    REAL(kind=8) :: xmin,xmax,ymin,ymax,dx,dy
    INTEGER(kind=8) :: nx,ny,niter,i
    INTEGER(kind=8) :: cx,cy,j,k
    REAL(kind=8) :: t_begin,t_end,t_middle
    integer(kind=8),dimension(:,:),allocatable :: MS

    t_begin = omp_get_wtime()
    allocate (MS (10000,10000))
    MS = 0
    OPEN(unit=1,file='MSet.in')
    READ(1,*) xmin,xmax,nx,ymin,ymax,ny,niter
    dx = (xmax-xmin)/(nx-1)
    dy = (ymax-ymin)/(ny-1)
    c = complex(xmin,ymin)
    OPEN(unit=2,file='Int.out')
```

Table 4: Comparison between diffrent SCHEDULE under $niter = 1000$

Threads	2	4	8	16	32	64
Without O3	139.89	74.64	46.93	26.48	15.28	8.49
With O3	68.278	33.03	17.22	9.01	6.25	3.90

```
      !$OMP PARALLEL DO SCHEDULE(guided) SHARED(MS) PRIVATE(cx,cy,i,c,z)
      do cx=1,nx-1
          do cy=1,ny-1
              z = (0,0)
              i = 1
              do while (abs(z)<=2 .AND. i<=niter)
                  z = z**2 + c
                  i = i + 1
              end do
              if ( i >= niter ) then
                  MS(cx,cy) = 1
              end if
              c = complex(real(c),ymin+cy*dy)
          end do
          c = complex(xmin+cx*dx,ymin)
      end do
      !$OMP END PARALLEL DO

      t_middle = omp_get_wtime()
      do j = 1, 10000
          do k = 1, 10000
              if ( MS(j,k) /= 0 ) then
                  WRITE(2,*) xmin+j*dx, ymin+k*dy
              end if
          end do
      end do
      t_end = omp_get_wtime()
      WRITE(*,*) "time", t_end-t_begin, t_middle-t_begin, t_end-t_middle
      WRITE(*,*) "case", xmin,xmax,nx,ymin,ymax,ny,niter

  end program MSet
```

My personal machine only has two cores and much low performance, so the parallel part was taken on a work station of our research group. So the serial performance in this section was re-computed and results are shown in Tab. 6.

Adding the optimization option "-O3" when compiling the code, the computation will be speeded up, especially for higher threads. The comparison under $niter = 1000$ is shown in Tab. 4.

By the way, it was also noticed that when parallel computing, the threads are not always working fully. From the Fig. 2, the distribution of points in the Mandelbrot set is not uniform. So for different c, the actual number of iterations will be totally different. That might decrease the computation efficiency if the points of s are allocated to threads equally and statically. It seems efficient enough that each thread handles same amount of c, but the actual workload will be different.

Table 5: Comparison between diffrent SCHEDULE under $niter = 1000$

Threads	2	4	8	16	32	64
Defaut SCHEDULE	117.37	61.08	39.24	22.18	12.97	6.85
Guided SCHEDULE	68.278	33.03	17.22	9.01	6.25	3.90

Table 6: Results of serial performances of Fortran

niter	10	100	200	250	400	500	600	750	800	1000
t_{total}	30.26	35.16	47.23	53.49	71.78	83.94	96.16	114.45	120.66	145.45
t_{loop}	3.79	16.13	28.42	34.60	53.07	65.32	77.54	95.89	102.06	126.48
t_{output}	26.47	19.03	18.81	18.89	18.72	18.63	18.63	18.56	18.60	18.97

A handle named SHCEDULE could be added to the PARALLEL sentence so that the distribution of tasks for each thread can be dynamic to keep all threads at work fully. For $niter = 1000$, the time cost comparison between static scheduel and guided scheduel is shown in Tab. 5.

2.1.4 d

The comparison of serial performances between the compiled language Fortran and the scripting language Matlab is shown in Tab. 2 and Sec. 2.1.2. And it shows that for such huge loops, compiled languages have much higer efficiency than scripting languages. By the way, the serial performance of Fortran can be discussed more. Furthermore, I tested its performance under $niter = 10, 100, 200, 250, 500, 400, 600, 750, 800, 1000$. The results are shown in Tab. 6.

$t_{total} = t_{end} - t_{begin}$ means the total time cost of the program. $t_{loop} = t_{middle} - t_{begin}$ means the time cost of the loop part, e.g. the parallel part in the code. $t_{output} = t_{end} - t_{middle}$ means the time cost of writing the output file.

The t_{output} decreases because when $niter$ is low, the points picked are much more. Then the time cost on output would be more. When $niter > 200$, t_{output} becomes almost stable, which means the amount of picked points becomes almost stable.

So t_{loop} will describe the performance more accurately than t_{total}. As for serial performances, Fig. 3 shows the relation between $niter$ and t_{loop}. Since it is for single thread, the it shows a simple linear relation, the more work load, the more time cost. In fact, when parallelizing the output part, which is also a DO LOOP, the time cost will increase much. I have not found the reason yet.

I tested parallel performances with different number of threads to see the strong scaling speedup. Results of strong scaling speedup is shown in Fig. 4. It shows that at the beginning of increasing threads, the performance increases linearly, but after about 10 threads, the increase becomes slower and slower.

Then I also tested the weak scaling efficiency. Results of weak scaling efficiency is shown in Fig. 5. It shows that at the beginning of increasing threads, the efficiencies are close. I guess it is because the computation process is simple, then the efficiency can keep. But the efficieny decreases rapidly with increasing number of threads.

2.1.5 e

Include in your tar archive your commented source code and a graphical rendering of your results.

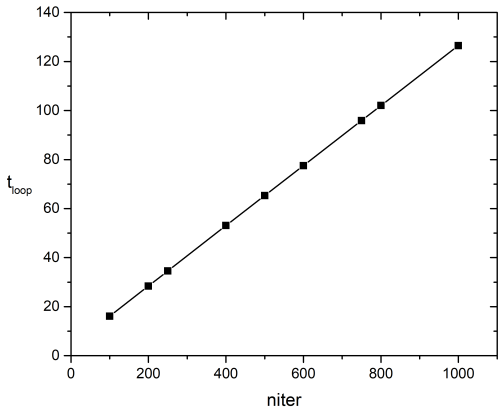

Figure 3: Figure of t_{loop} with niter

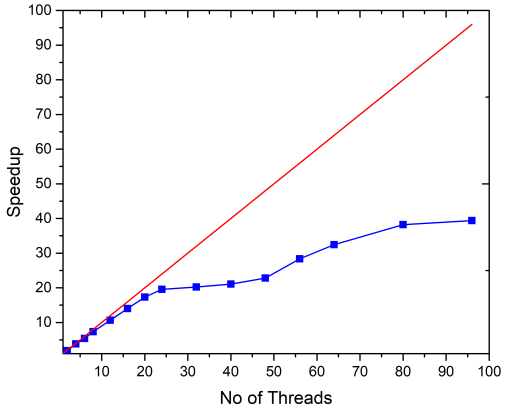

Figure 4: Figure of Strong Scaling

2.2 Tutorial: Programming in C

2.2.1 Setting up your programming environment

The gcc version of my local laptop is shown in Figure 6.

2.2.2 Writing a "hello world" program in C

The output of **/.hello** in terminal is shown below.

```
Hello world 0 0.000
Hello world 1 2.500
Hello world 2 5.000
```

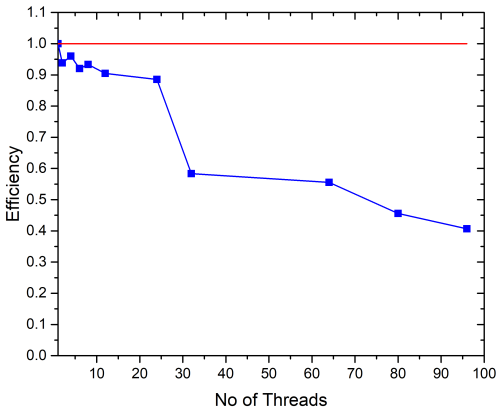

Figure 5: Figure of Weak Scaling

```
silverz@DESKTOP-4UP5RP6:/mnt/c/Users/Qihao Zhuo/OneDrive/Homework/Numerical Methods$ cd Problem\ Set\ 0/
silverz@DESKTOP-4UP5RP6:/mnt/c/Users/Qihao Zhuo/OneDrive/Homework/Numerical Methods/Problem Set 0$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

silverz@DESKTOP-4UP5RP6:/mnt/c/Users/Qihao Zhuo/OneDrive/Homework/Numerical Methods/Problem Set 0$
```

Figure 6: GCC Vsersion

```
Hello world 3 7.500
Hello world 4 10.000
Hello world 5 12.500
Hello world 6 15.000
Hello world 7 17.500
Hello world 8 20.000
Hello world 9 22.500
10 22.500
9 20.000
8 17.500
7 15.000
6 12.500
5 10.000
4 7.500
3 5.000
2 2.500
1 0.000
```

2.3 Parrallelizing your code using OpenMP

The output of **/.hello_omp** in terminal is shown below.

```
Hello from thread 2, i=6, data=15.000
Hello from thread 2, i=7, data=17.500
Number of threads is 4
Hello from thread 0, i=0, data=0.000
Hello from thread 0, i=1, data=2.500
Hello from thread 0, i=2, data=5.000
Hello from thread 1, i=3, data=7.500
Hello from thread 1, i=4, data=10.000
Hello from thread 1, i=5, data=12.500
Hello from thread 3, i=8, data=20.000
Hello from thread 3, i=9, data=22.500
End of parallel region
tid=32685, 0 0.000
tid=32685, 1 2.500
tid=32685, 2 5.000
tid=32685, 3 7.500
tid=32685, 4 10.000
tid=32685, 5 12.500
tid=32685, 6 15.000
tid=32685, 7 17.500
tid=32685, 8 20.000
tid=32685, 9 22.500
```