# COMP90025 Project 1A: OpenMP and Mandelbrot Set

**San Kho Lin (829463) — sanl1@student.unimelb.edu.au**

## 1 Introduction

In this project, I am going to experiment OpenMP programming on a given serial program which is counting the total sum of complex points that fall under a Mandelbrot Set.

## 2 Mandelbrot Set

To begin with, I first study on the Mandelbrot Set. The set is in complex plane and start with a given point $c$, then apply a certain number of iterations to the transformation in such that:

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

The iteration is continued until the magnitude of $z_{n+1}$ is greater than 2 (escape from set) or the number of iterations reaches the limit (bounded in set). The initial condition is $z_0 = 0$. The magnitude of $z = z_{n+1}$ is $\sqrt{a2 + b2}$ where $z = a + bi$.(Harwood, 2016)

The following are some of the significant characteristics of Mandelbrot Set for considering parallel programming.

- The set is mathematically '*self similar*' object — it is exactly or similar to a part of itself.
- Some starting points reach the escape (or bounded) condition fast, whereas others may take a large number of iterations to do so.

(Krieger, 2014)(Gang et al., 2009)

## 3 Serial Nature of the Set

The code in **Listing 1** depict the computation of finding a given point is in Mandelbrot Set. The loop in this computation is not suited for parallelization. It is related to the significant characteristics aforementioned in **Section 2** as follow.

- The loop can only be calculated in serial because the calculation of next iteration depends on the previous iteration.
- The loop terminal condition vary around either escapes a circle of radius 2 or hit the maximum iteration. The trip count is not known until the loop is executed and, consequently it is difficult to distribute work load to parallel threads.

(Gove, 2011)

## 4 Parallel Porting

The code in **Listing 2** contains two loops. Each iteration compute 4 points — the box region and, the trajectory of the region is getting smaller as it grows at the rate of resolution by *num* step and, until reaching the upper bound. This portion of code is parallelized as shown in **Listing 3**. This loop produce points in complex plane to find whether they are in Set. The step approach is similar to Moler (2011) Matlab toolbox tutorial.

## 5 Load Balancing

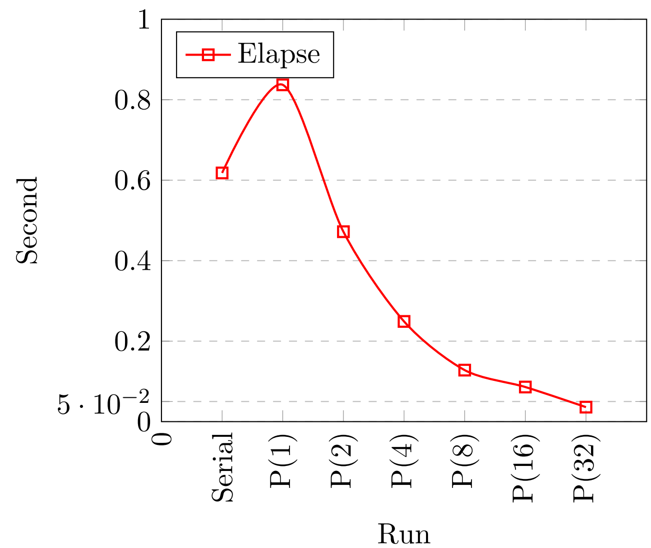
The *speedup* performance challenge with Mandelbrot Set is mitigating the load imbalance. Several heuristics approaches have proposed (Matloff, eBook)(Gove, 2011)(Gang et al., 2009). In my experiment, I am using the OpenMP worksharing constructs schedule *static*, *guided*, *dynamic* and, the speedup increase significant with *dynamic* schedule as in Gove example. However, Dr. Matloff (eBook) argues that the Set computation in Gove example is *Embarrassingly Parallel* in such that there is no communication between the two half-divided Set. He further states that the **static** (but possibly **random**) task assignment is typically better than *dynamic*. I also try this approach. Its load balancing is much better than schedule *static* and, comparable to *dynamic* and *guided*. I use *Intel VTune Amplifier*[1] to perform OpenMP hotspot analysis. **Figure 4** show the schedule *static* CPU timeline. There are 2-idle threads out of total 8 processors, whereas the other approaches **Figure 5, 6 and 7**, a better load balance observed.

[1]Part of Intel Parallel Studio XE

## 6 Limitation

**Figure 1** show the outcome of 10 consecutive OpenMP jobs run on VLSCI[2] Snowy cluster through Slurm[3] queue. I find that the *Static-Random* approach do not outperform schedule *dynamic* as in Dr.Matloff book. I observe that the limitation may come from overhead of array construct and Fisher-Yates Shuffle[4] which has $O(n)$ linear complexity for randomization step. Beside, the array construct[5] has potential memory bound and serial aspect of program has expanded. Together with **Section 3**, this realize the *Amdahl's Law*; the fraction of the problem that cannot be parallelized a function of the problem size, $f(n)$ and the speedup is limited to $\frac{1}{f(n)}$ for a large number of processors (Harwood, 2016).

Plot 1: Linear Speedup with Dynamic Schedule




## 7 Conclusion

In regard to PRAM, the practical part of parallel computing has to consider communication activity, layer of caches and memory contention, local data load balancing and, all these constitute to wall-clock speedup time. On the other hand, PRAM is an abstract model for idealization of a parallel architecture, whereas it conveys to theoretical aspect of algorithmic step and time. In summary, the **Table 1** breakdown the program run of serial to parallel schedule *dynamic* with number of processor 1, 2, 8, 16 and 32. And the **Plot 1** visualize the linear speedup curve of the table.

$$S(p_{32}) = \frac{T_s}{T_p} = \frac{0.618}{0.036} \approx 17.1667$$

$T_s$ = Execution time using one processor
$T_p$ = Execution time using 32 processors

Table 1: Linear Speedup Breakdown

| Run | Elapse |
|---|---|
| Serial | 0.618 |
| P(1) | 0.837 |
| P(2) | 0.472 |
| P(4) | 0.249 |
| P(8) | 0.128 |
| P(16) | 0.086 |
| P(32) | 0.036 |

[2] Victorian Life Sciences Computation Initiative
[3] http://slurm.schedmd.com/
[4] https://en.wikipedia.org/wiki/Fisher-Yates_shuffle
[5] Code Listing 4 Line 49

## 8 References


Aaron Harwood [2016]. "COMP90025 Parallel and Multicore Computing". Lecture series — Introduction, Architectures, Classes of parallel algorithms. The University of Melbourne.

Dr. Norm Matloff [eBook]. "Programming on Parallel Machines". University of California, Davis. GPU, Multicore, Clusters and More.
http://heather.cs.ucdavis.edu/parprocbook

Darryl Gove [2011]. "Multicore Application Programming". For Windows, Linux, and Oracle Solaris. Developer's Library.

Isaac K. Gang, David Dobson, Jean Gourd and Dia Ali. "Parallel Implementation and Analysis of Mandelbrot Set Construction", University of Southern Mississippi.

Dr. Holly Krieger [2014] MIT. "The Mandelbrot Set - Numberphile".
https://www.youtube.com/watch?v=NGMRB4O922I

Beej [2010]. "The Mandelbrot Set".
http://beej.us/blog/data/mandelbrot-set/

Cleve Moler [2011]. "Experiments with MATLAB, Chapter 13 Mandelbrot Set"
http://au.mathworks.com/moler/exm/chapters.html

## 9 Appendix

```c
int inset(double real, double img, int maxiter) {
  double z_real = real;
  double z_img = img;
  for(int iters = 0; iters < maxiter; iters++) {
    double z2_real = z_real*z_real-z_img*z_img;
    double z2_img = 2.0*z_real*z_img;
    z_real = z2_real + real;
    z_img = z2_img + img;
    if(z_real*z_real + z_img*z_img > 4.0) return 0;
  }
  return 1;
}
```

Listing 1: Find a given point is in Mandelbrot Set

```c
int mandelbrotSetCount(double real_lower, double real_upper, double
    img_lower, double img_upper, int num, int maxiter) {
  int count=0;
  double real_step = (real_upper-real_lower)/num;
  double img_step = (img_upper-img_lower)/num;
  for(int real=0; real<=num; real++) {
    for(int img=0; img<=num; img++) {
      count+=inset(real_lower+real*real_step,img_lower+img*img_step,
    maxiter);
    }
  }
  return count;
}
```

Listing 2: Serial — Count the number of points that are in Mandelbrot Set

```c
    #pragma omp parallel
    {
        double complex c;
        double real_step = (real_upper - real_lower) / num;
        double img_step = (img_upper - img_lower) / num;

        #ifdef STATIC
        #pragma omp for reduction(+:count) schedule(static)
        #elif defined GUIDED
        #pragma omp for reduction(+:count) schedule(guided)
        #else
        #pragma omp for reduction(+:count) schedule(dynamic)
        #endif

        for (int real = 0; real <= num; real++) {
            for (int img = 0; img <= num; img++) {
                c = (real_lower + real * real_step) + (img_lower + img
    * img_step) * I;
                count += inset(c, maxiter);
            }
```

```
        }
    }
```

Listing 3: Parallel — Count the number of points that are in Mandelbrot Set

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <complex.h>

#ifdef RANDOM
#include <time.h>
typedef struct {
  double real, img;
} Coord;

// Reference:
// Dr. Norm Matloff [Online eBook] "Programming on Parallel Machines"
// Chapter 2.2 Load Balancing
// Chapter 2.4 Static (But Possibly Random) Task Assignment Better Than Dynamic
// Chapter 4.4 Example: Mandelbrot Set
void findmyrange (int n, int nth, int me, int *myrange) {
    int chunksize = n / nth;
    myrange[0] = me * chunksize;
    if (me < nth-1) {
        myrange[1] = (me+1) * chunksize - 1;
    } else {
        myrange[1] = n - 1;
    }
}
#endif

// return 1 if in set, 0 otherwise
int inset(double complex c, int maxiter) {
    double z_real, z_img;
    double complex
    z = c;
    for (int iters = 0; iters < maxiter; iters++) {
        z = z * z + c;
        z_real = creal(z);
        z_img = cimag(z);
        if (z_real * z_real + z_img * z_img > 4.0) return 0;
    }
    return 1;
}

// count the number of points in the set, within the region
int mandelbrotSetCount(double real_lower, double real_upper, double img_lower, double img_upper, int num, int maxiter) {
    int count = 0;

```

```
#ifdef RANDOM

int idx_size = (num+1) * (num+1);
Coord points[idx_size];
double real_step = (real_upper-real_lower)/num;
double img_step = (img_upper-img_lower)/num;
int cnt = 0;
for (int real=0; real<=num; real++) {
    for (int img=0; img<=num; img++) {
        Coord p;
        p.real = real_lower+real*real_step;
        p.img = img_lower+img*img_step;
        points[cnt] = p;
        cnt++;
    }
}

// Reference:
// Fisher Yates shuffle
int n = sizeof(points) / sizeof(points[0]);
srand(time(NULL));
for (int i = n-1; i > 0; i--) {
    int j = rand() % (i+1);
    Coord temp = points[i];
    points[i] = points[j];
    points[j] = temp;
}

#pragma omp parallel reduction(+:count)
{
    double complex c;
    int myrange[2];
    int me = omp_get_thread_num();
    int nth = omp_get_num_threads();
    int i;
    findmyrange(idx_size, nth, me, myrange);
    for (i = myrange[0]; i <= myrange[1]; i++) {
        Coord p = points[i];
        c = p.real + p.img*I;
        count += inset(c, maxiter);
    }
}

#else

#pragma omp parallel
{
    double complex c;
    double real_step = (real_upper - real_lower) / num;
    double img_step = (img_upper - img_lower) / num;

```

```
        #ifdef STATIC
        #pragma omp for reduction(+:count) schedule(static)
        #elif defined GUIDED
        #pragma omp for reduction(+:count) schedule(guided)
        #else
        #pragma omp for reduction(+:count) schedule(dynamic)
        #endif

        for (int real = 0; real <= num; real++) {
            for (int img = 0; img <= num; img++) {
                c = (real_lower + real * real_step) + (img_lower + img * img_step) * I;
                count += inset(c, maxiter);
            }
        }
    }

    #endif

    return count;
}

// main
int main(int argc, char *argv[]) {
    double real_lower;
    double real_upper;
    double img_lower;
    double img_upper;
    int num;
    int maxiter;
    int num_regions = (argc - 1) / 6;
    for (int region = 0; region < num_regions; region++) {
        // scan the arguments
        sscanf(argv[region * 6 + 1], "%lf", &real_lower);
        sscanf(argv[region * 6 + 2], "%lf", &real_upper);
        sscanf(argv[region * 6 + 3], "%lf", &img_lower);
        sscanf(argv[region * 6 + 4], "%lf", &img_upper);
        sscanf(argv[region * 6 + 5], "%i", &num);
        sscanf(argv[region * 6 + 6], "%i", &maxiter);
        printf("%d\n", mandelbrotSetCount(real_lower, real_upper, img_lower, img_upper, num, maxiter));
    }
    return EXIT_SUCCESS;
}
```

Listing 4: Complete OpenMP Parallel Program

Figure 1: Four approaches over 10 runs (values are in second)

| | 32 Processors | | | | | 1 Processor |
|---|---|---|---|---|---|---|
| Run | Dynamic | Random | Guided | Static | | Serial |
| 1 | 0.049 | 0.055 | 0.069 | 0.091 | | 0.712 |
| 2 | 0.037 | 0.05 | 0.052 | 0.072 | | 0.691 |
| 3 | 0.039 | 0.051 | 0.054 | 0.072 | | 0.691 |
| 4 | 0.035 | 0.044 | 0.054 | 0.073 | | 0.691 |
| 5 | 0.035 | 0.043 | 0.054 | 0.073 | | 0.692 |
| 6 | 0.039 | 0.044 | 0.054 | 0.073 | | 0.692 |
| 7 | 0.037 | 0.043 | 0.054 | 0.073 | | 0.691 |
| 8 | 0.038 | 0.051 | 0.054 | 0.073 | | 0.693 |
| 9 | 0.039 | 0.054 | 0.054 | 0.072 | | 0.692 |
| 10 | 0.037 | 0.043 | 0.052 | 0.072 | | 0.692 |
| AVG | 0.0385 | 0.0478 | 0.0551 | 0.0744 | | 0.6937 |

Figure 2: Four approaches over 10 runs with 32 processors

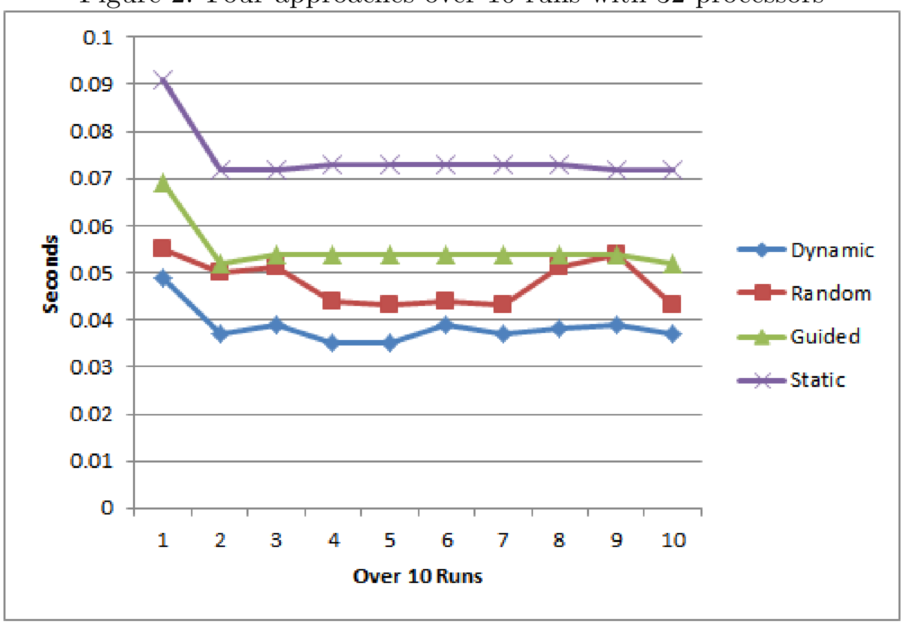



Figure 3: Average wall-clock elapse of Four approaches over 10 runs with 32 processors

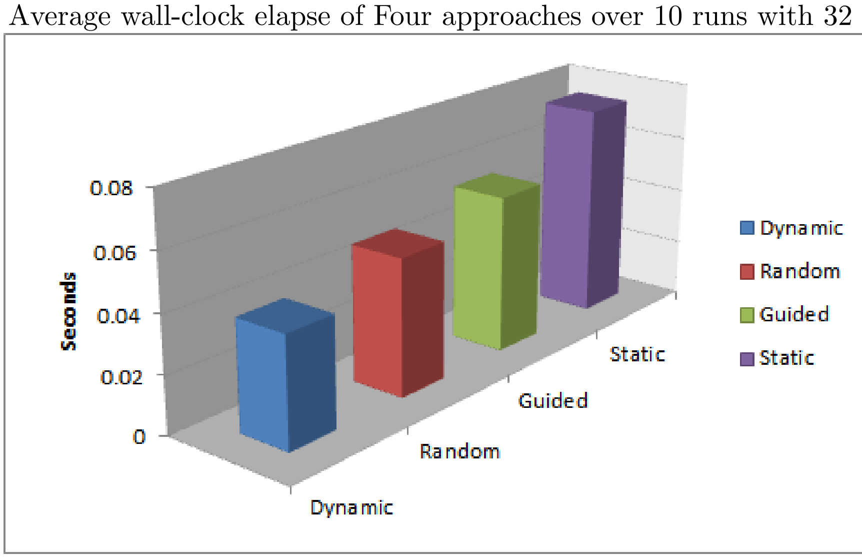


Figure 4: Intel VTune CPU Usage Timeline — schedule(static)

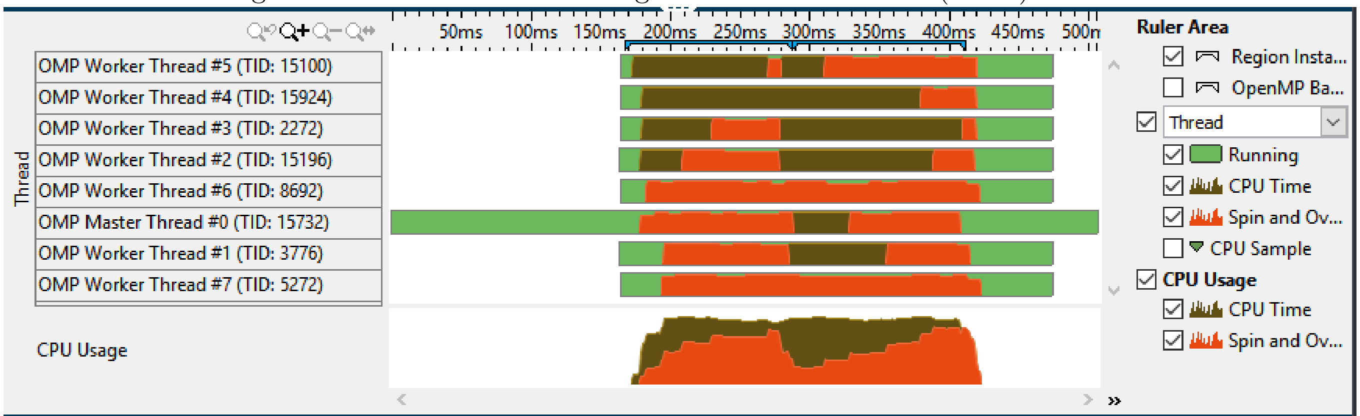



Figure 5: Intel VTune CPU Usage Timeline — schedule(guided)

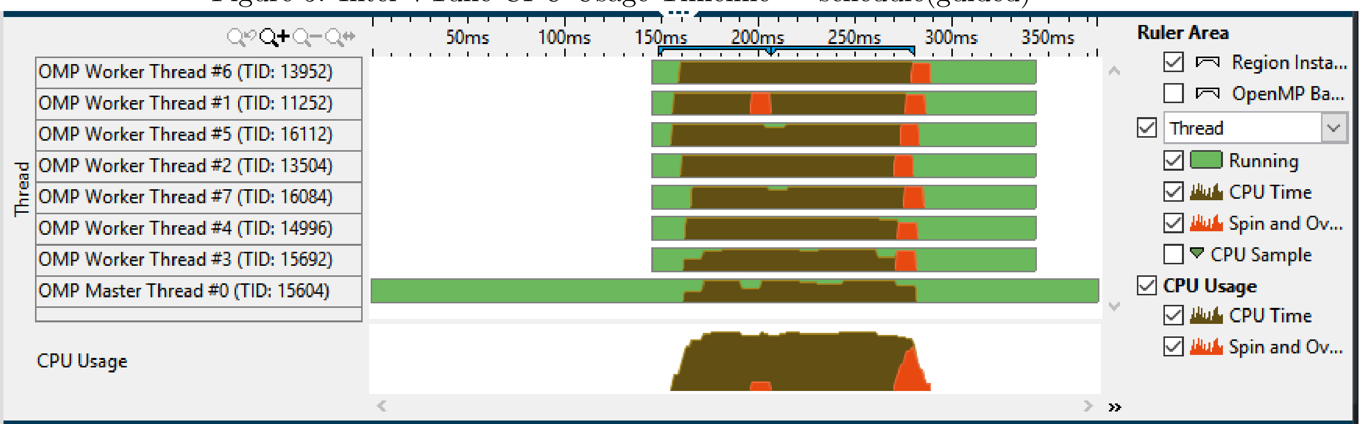



Figure 6: Intel VTune CPU Usage Timeline — schedule(dynamic)

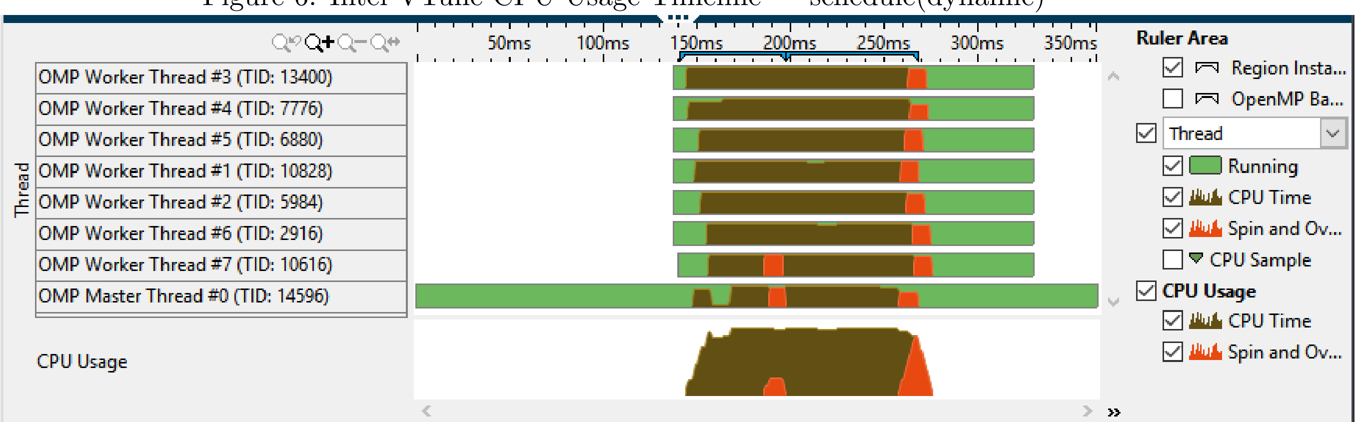



Figure 7: Intel VTune CPU Usage Timeline — Static-Random