Fractals

Matthew Redoute
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester MI
e-mail: mredoute@oakland.edu

ABSTRACT
Fractals, such as the Mandelbrot and Julia Sets, are great applications to perform parallelization. This is because complex coordinates within the complex plane are inserted into an identical function to determine its belonging in the set. Two parallel approaches coming from Threading Building Blocks (TBB) were utilized: parallel_for and parallel_invoke. The results showed significant decrease in time and even halved sequential implemenations at the higher step sizes. The Julia set's elapsed time was a lot less, causing the parallel_invoke strategy almost useless. However, another time consuming fractal similar to the Mandelbrot would make a perfect example for this second parallel strategy.

EQUATION

$$f_c(z) = z^2 + c \text{ OR } z_{n+1} = z_n^2 + c, \qquad (1)$$

Ⅰ. INTRODUCTION

The contents of the report contain an introduction on the Mandelbrot and Julia fractals, a detailed methodology on the sequential and parallel implementations, an overview of the experimental setup, and a discussion of the gathered results.

A. Mandelbrot Set

The Mandelbrot set, represented by equation 1, is a 2D set defined in a complex plane where *c* is a complex number that does not diverge into infinity upon iteration of *z* at 0. Therefore, recursion is applied resulting in two output cases: unbounded (not in the set, meaning blowing up or going to infinity) or bounded (in the set). An example can be seen in Table 1 for complex numbers 1 and -1 indicating unbounded and bounded, respectively [1] [3].

B. Filled Julia Set

The filled Julia set is similar to the Mandelbrot set because it also uses equation 1 to determine if *z* blows up upon iteration. However, the complex number *z* can now start anywhere while *c* is fixed. Additionally, the filled Julia set's output behavior can determine if the starting point belongs in the Mandelbrot set. For example, the filled Julia set can result in one piece and disconnected pieces resulting in it belonging and not belonging to the Mandelbrot set, respectively [2] [3].

Mandelbrot Equation: (un)Bounded Complex Elements		
iteration	CASE 1: c = 1 (unbounded)	CASE 2: c = -1 (bounded)
0	1	-1
1	2	0
2	5	-1
3	26	0

Table 1: Example of Complex Elements Bound vs Unbound

C. Fractal Dimensions

Fractal dimensions determine the granularity of a fractal. The more complex coordinates there are between the minimum and maximum coordinates, the better looking the fractal becomes. For example as shown in Figure 1, a shorter stick reveals a more detailed fractal.

Figure 1: Example of Fractal Dimension

Ⅱ. METHODOLOGY

A. Sequential Implementation

As discussed in the introduction, the sequential implementation can be performed through equation 1 where recursion is applied until 'z' is greater than two or the interaction count has reached the maximum number of iterations; therefore, the complex number coordinate is not and is part of the Mandelbrot and Julia sets, respectively.

B. Pseudocode

The understanding of the algorithms were first achieved through Octave due to its simplistic matrix style programming language and as a proof of concept for the sequential implementations. The descriptive pseudocode below reveals the steps to generate Mandelbrot Figure 2 and Julia Figure 3.

1. Set up all complex numbers (Re, Im) to be called by functions *IsCpxNumInMandelbrotSet* and *IsCpxNumInJuliaSet.* Respectively from equation 1, Mandelbrot has 1 parameter in which the complex number represents 'c' while Julia has 2 parameters in which the complex number represents 'z' while 'c' is fixed at a complex coordinate number.
2. Within both functions: Create & set vars *z* (only for Mandelbrot) *& iter* to 0 b/c each complex number iterates from 0 to ∞ (i.e., max iters). Complex number(s) fed into the equation runs until *z > 2 or iter > max_iter*. If *iter == max_iter*, then complex is in the set, thus return TRUE, else FALSE.
3. If the function returns TRUE, then plot the cpx num (Re, Im) onto the complex plane.

Pseudocode: Mandelbrot & Julia Set

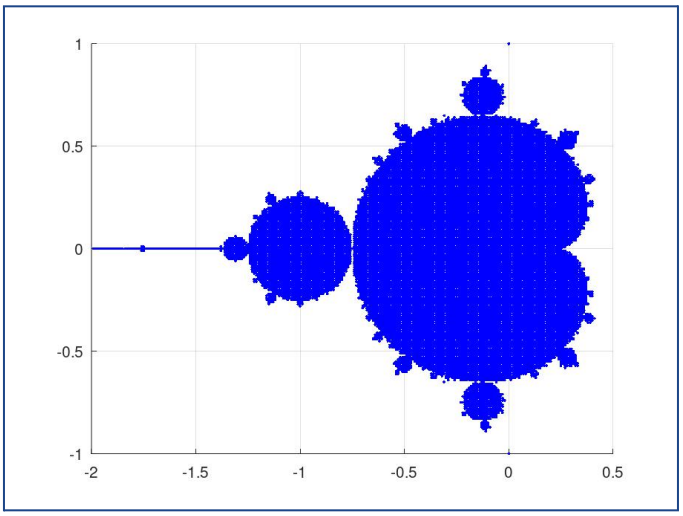

Figure 2: Mandelbrot Set Proof of Concept

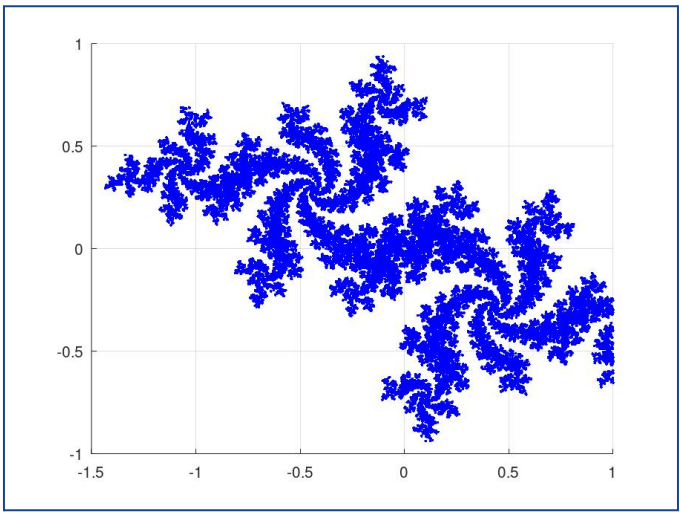

Figure 3: Julia Set Proof of Concept (*c* at -0.54 + 0.54i)

C. Parallel Implementation

Since both fractals are utilizing equation 1 differently, i.e. the Mandelbrot changes *c* while it is constant for Julia, TBB parallel_invoke was utilized. Two different functions (one for Mandelbrot and one for Julia) were created representing the tasks performed in parallel. Additionally, due to each complex plane coordinate needing to be fed into the functions to determine if it belongs in the set, TBB parallel_for was applied as well. Therefore, two parallelization strategies were implemented: parallel_for and parallel_invoke followed by parallel_for. In either approach the complex numbers being sent to these functions are each done in parallel instead of the sequential approach where the next complex number is fed once the current one is finished.

1. parallel_for

The first approach implements a map pattern allowing a collection of elements to be iterated in parallel. A parallel_for function typically replaces a sequential for loop where each iteration calls the body statements in parallel. The iterations are broken into tasks and sent to the operating system scheduler, where available threads execute iterations (non-deterministically). Below represents the syntax: *parallel_for(blocked_range<int>(0, n), func_obj);*. The blocked_range represents the iterations as [0, n). However, compact lambda expressions were used which modified the syntax to hide the blocked_range and anonymously create the function object. The new syntax became: *parallel_for(int(0), int(n),[capture-list] (int k);* where capture-list captures the variables by value or reference and k is the current iteration index. As shown in Figure 4, the flowchart reveals that the complex coordinates (Re, Im) are sent both to the Mandelbrot and Julia functions through one parallel_for function.

2. parallel_invoke

The second approach also implements a map pattern but on different tasks. Below represents the syntax: *parallel_invoke (const Func0& f0, … , Func9& f9);* Similarly, compact lambda expressions were used for the parameters to replace the need of manually creating function objects (functors). As shown in Figure 5, the flowchart reveals the Mandelbrot and Julia functions as the two tasks in which parallelization is performed. With this approach, if the two functions take the same time sequentially, then a time elapsed calculation should be half of the first parallel approach [4].

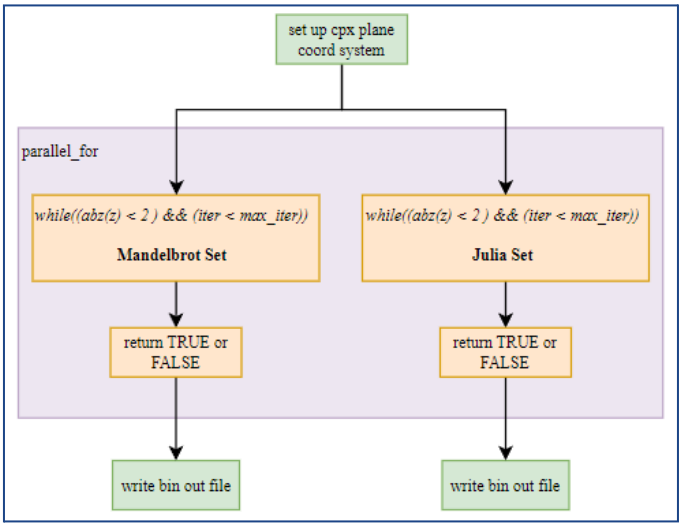

Figure 4: P1 Strategy - `parallel_for`

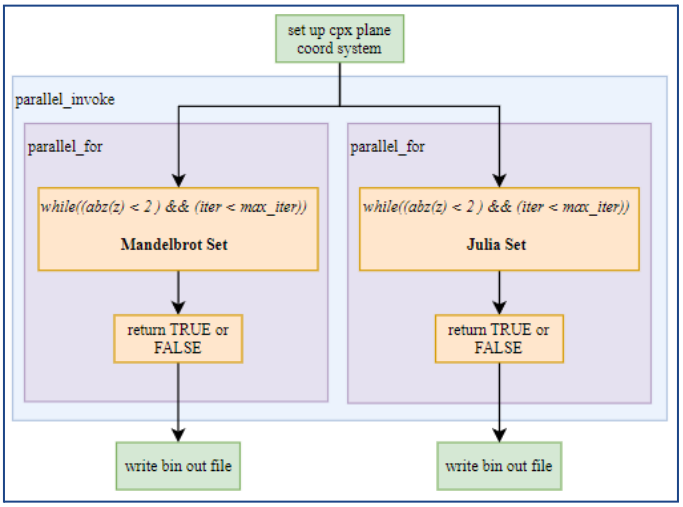

Figure 5: P2 Strategy - `parallel_invoke` & `parallel_for`

III. EXPERIMENTAL SETUP

A. Hardware & Software

The code was written in Ubuntu on a Terasic DE2I-150 board consisting of an Intel Atom processor running at 1.6 GHz containing two cores with two execution threads per core (4 logical processors). Additionally, the code was tested on a windows laptop (through a VM) consisting of an Intel i7-7700HQ running at 2.8 GHz with 2 cores with 2 execution threads per core (4 logical processors).

B. Test Cases

Ten step sizes were utilized to obtain timing results from small to large vectors. A smaller step size meant more complex numbers resulting in a large vector size. For example, as shown in Figure 6, a step size of one creates a vector size of twelve.

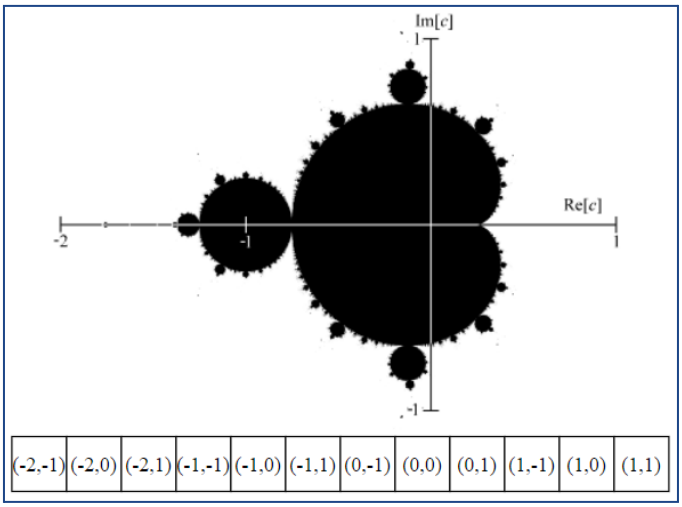

Figure 6: Mandelbrot cpx plane & Vector size (step of 1) [5]

C. Verification

The verification of the created fractals in C++ were compared with the Octave results. This was achieved by returning a one or zero for each complex number (answering the question if the complex number is in the fractal set) and storing the result in a large vector. The vector is then written to a binary output file which is then compared with a vector generated from an Octave implementation. Finally, a sum of differences is calculated between the two vectors where a result of zero means the C++ implementation was successful.

D. Valgrind

Valgrind is a memory programming debugging tool for discovering memory leaks and (de)allocation issues. The tool specifically helped solve a segmentation fault due to accessing unavailable allocated memory space. Therefore, the allocated vector size storing the results was fixed by increasing its size to the correct amount.

IV. RESULTS

The execution of the program is performed by the bash command: *$ ./fractals <step_size>*. Figure 7 displays a terminal output where the step size equals 1 (vector size, range of 12) and with the macro DEBUG set to TRUE. The DEBUG

macro reveals the total number of complex numbers that belong to the fractal set, making it easier to identify if the sequential and parallel implementations were identical.

```
buddy@vbox:~/SW/Final_Project/Fractals_V2$ ./fractals 1
step = 1
vector size = 12

DEBUG STATEMENTS
total_seq_m = 5
total_seq_j = 0
total_par_1_m = 5
total_par_1_j = 0
total_par_2_m = 5
total_par_2_j = 0

RESULTS
        SM      P1M     P2M     SJ      P1J     P2J
z[0]:   0       0       0       0       0       0
z[1]:   1       1       1       0       0       0
z[2]:   0       0       0       0       0       0
z[3]:   0       0       0       0       0       0
z[4]:   1       1       1       0       0       0
z[5]:   0       0       0       0       0       0
z[6]:   1       1       1       0       0       0
z[7]:   1       1       1       0       0       0
z[8]:   1       1       1       0       0       0
z[9]:   0       0       0       0       0       0
z[10]:  0       0       0       0       0       0
z[11]:  0       0       0       0       0       0
```

Figure 7: Terminal output with step size as 1 & DEBUG macro set to TRUE

A. Comparing Time Measurements

Time measurements were gathered which can be shown in the Appendix section. The main columns to look at are the sequential Mandelbrot and Julia (SMJ) and the parallel ones (P1 and P2). Additionally, individual fractal implementations were measured (first four columns) to see one's performance without the other fractal present.

The first table describes time measurements from the board with a maximum iteration set to 512. The higher step sizes (0.1, 0.5, and 0.25) result in slower times for parallelization, primarily because of the TBB library and API overhead that is needed. However, reaching a step size between 0.25 and 0.1 transitions faster times for the parallel strategies with each halving the sequential time. Additionally, it is important to note that the P1 and P2 strategies were almost identical except for the first couple test cases. This makes sense because the individual Julia implementations were significantly faster than the Mandelbrots ones. If the SM & SJ were similar, then the P2 strategy would halve the P1 strategy. Table 2 describes the time measurements from the board at a maximum iteration set to 1024, resulting in the doubling for all the SM, PM, and SMJ implementations. SJ and PJ stayed the same while P1 and P2 doubled at the lower step sizes. The third and fourth tables describe the time measurements from the laptop. Due to a faster processor, the parallel implementations showed faster times at almost all the test cases except the first couple.

B. Challenges

1. Missing Complex Coordinates

The first challenge encountered was the missing of complex coordinates that were being sent to the fractal functions. It is still unclear the reasoning behind this, but this phenomenon occurred during lower step sizes. The solution involved dynamically allocating an array (based on the user inputted step size) where each element consisted of a struct with data members for the real and imaginary coordinates.

2. Complex Compatibility Issues

The second challenge encountered was complex compatibility issues from compiling with C vs C++. With online research the solution became clear that the programming languages used different syntaxes for the creation of complex numbers: *double complex z; z = Re + Im * I* (<complex.h> and *complex<double> z: z = (Re, Im) (<complex>)*, respectively.

3. Complex Coordinates as 0

The third challenge encountered was the complex plane coordinates for 0 would instead be a very small number. This phenomenon would only happen in my virtual machine environment. In fact, the proof of concept Octave implementations on my native windows OS worked as expected, but running the same file in Octave within the VM showed this happening. The solution became clear that this issue was machine dependent.

4. Race Conditions for TBB

The fourth and final challenge was experiencing race conditions for the debug variables while dealing with the TBB functions. Since the debug variables were outside the scope of the functions, multiple threads could access them at a time yielding the *total_** variables to produce different results when running the application.

V. CONCLUSIONS

In conclusion, TBB's performance led to the decrease in computation time for both Mandelbrot and Julia Sets. This fractal application showcased the power of TBBs and why the use of multiple cores and threads can almost always improve one's performance. Some further improvements would be to create more fractals and to return the iteration of each complex number which can be used for coloring the fractals.

VI. REFERENCES

[1] https://en.wikipedia.org/wiki/Mandelbrot_set
[2] https://en.wikipedia.org/wiki/Julia_set
[3] https://www.youtube.com/watch?v=NGMRB4O922I
[4] https://www.secs.oakland.edu/~llamocca/index.html
[5] https://ncatlab.org/nlab/show/Mandelbrot+se

VII. APPENDIX - DATA COLLECTION

BOARD (MAX_ITER = 512)							
step (size)	SM (us)	PM (us)	SJ (us)	PJ (us)	SMJ (us)	P1 (us)	P2 (us)
1 (12)	994	2774	55	70	865	3042	1006
0.5 (35)	1811	3620	143	250	1903	4714	3016
0.25 (117)	6616	7405	808	1873	6132	8148	9077
0.1 (651)	26726	17808	5956	4005	37110	22138	19428
0.05 (2501)	106913	67822	24720	15188	125805	77740	65549
0.01 (60501)	2387769	1246708	592183	345369	3053782	1585526	1587576
0.003 (668334)	26783295	11259956	5786148	2939285	34901261	14442918	14318074
0.002 (1502501)	54910291	25633611	13566255	6510495	69535060	30641099	30070675
0.0015 (2670000)	98125648	45182934	24216932	17286905	136193852	83631824	83616507
0.001 (6005001)	268878332	98561471	54764069	25421260	274942426	120579766	145909507

BOARD (MAX_ITER = 1024)							
step (size)	SM (us)	PM (us)	SJ (us)	PJ (us)	SMJ (us)	P1 (us)	P2 (us)
1 (12)	1559	3542	53	125	1669	3184	2378
0.5 (35)	4308	6189	129	193	3846	5311	3173
0.25 (117)	11260	16882	805	3271	12158	10470	6752
0.1 (651)	51243	36194	6711	7583	64097	45250	43305
0.05 (2501)	204560	141977	22909	22423	233678	145240	135662
0.01 (60501)	4716694	2082056	611544	255156	5413252	2171016	2180614
0.003 (668334)	45625752	22102791	6039269	3163874	52690876	24381534	24313403
0.002 (1502501)	102824710	49832446	13415408	6977120	117312358	79193337	83965207
0.0015 (2670000)	217062838	133191937	31088184	17298713	216773118	142251986	146668254
0.001 (6005001)	513128798	267296673	65402022	36764013	533373583	300023789	301700639

LAPTOP (MAX_ITER = 512)							
step (size)	SM (us)	PM (us)	SJ (us)	PJ (us)	SMJ (us)	P1 (us)	P2 (us)
1 (12)	164	348	11	16	180	329	116
0.5 (35)	338	465	24	20	384	490	199
0.25 (117)	964	731	151	77	1099	711	473
0.1 (651)	5164	2888	1399	731	6607	3503	3681
0.05 (2501)	18912	9182	4109	1897	27740	11267	11355
0.01 (60501)	424705	233144	105539	57103	529670	272700	277526
0.003 (668334)	4531646	2417346	1112228	572522	5659827	2951749	3002682
0.002 (1502501)	10237756	5617855	2531764	1336452	12782056	6750351	6848396
0.0015 (2670000)	18245001	9831741	4545176	2353583	22795041	12046066	12092886
0.001 (6005001)	43404259	22176000	10614034	5316964	53555897	26935598	27224907

LAPTOP (MAX_ITER = 1024)							
step (size)	SM (us)	PM (us)	SJ (us)	PJ (us)	SMJ (us)	P1 (us)	P2 (us)
1 (12)	291	453	10	11	310	407	198
0.5 (35)	713	556	25	19	712	666	437
0.25 (117)	1992	1170	150	135	2076	1427	1279
0.1 (651)	10214	6515	1360	786	12197	6999	9675
0.05 (2501)	35564	17329	3956	2090	47771	20029	19598
0.01 (60501)	815100	436605	11007	54505	924907	478501	487119
0.003 (668334)	8911883	4841782	1162173	626614	10159241	5556781	5529584
0.002 (1502501)	20901921	10792177	2857091	1439762	24455671	12501607	12609205
0.0015 (2670000)	35990715	19154444	4686936	2447890	40564312	21331152	21662382
0.001 (6005001)	82276160	42938549	10547304	5506977	94221344	47369804	48379850