Erlang

Assignment 2
High Performance Computing
UCD School of Computer Science and Informatics

Author: Robert O'Regan
Email: robertoregan@eircom.net
Student No: 08283974

Introduction

For this assignment I chose the Concurrent Programming language 'Erlang'. Along with this paper, I implemented two versions of the Mandelbrot algorithm - a sequential version and a concurrent version.
Section 1 of this paper contains a description of Erlang and discusses some of its fundamental aspects. Section 2 discusses Concurrency in Erlang. Section 3 contains details on the implementation of the Mandelbrot algorithms in Erlang. Section 4 contains a detailed comparison of the Mandelbrot algorithms and finally, Section 5 contains a brief conclusion.

1 Erlang

[1] "Erlang is a general-purpose concurrent programming language and runtime system. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications and with the aim of improving the development of telephony applications".

1.1 Erlang Environment

Shell

Erlang has its own shell where you can directly input and execute Erlang code. For example, when at the shell prompt, if you input '2 + 5.' and hit enter, the calculation will be processed and the result returned.
One point to note is that all commands input at the shell should end in a period – '.'.

Modules & Functions

Like most programming languages, Erlang provides a way of grouping code so it can be re-used. This is done through modules and functions. If coming from an Object perspective, Modules can be thought of as classes. Modules contain functions that can be called and which will generally return a result or complete an action.
In Erlang, you may specify multiple patterns against which calls to a function will be matched. This is similar to function overloading in procedural languages. One important thing to note is that Erlang functions are identified by the combination of their name and their arity. Arity refers to the number of parameters that a function accepts. If we define two functions with the same name but different arity, Erlang will consider them to be two separate functions.
The following is a basic Erlang module…

```
-module(tut).
-export([double/1]).
        double(X) ->
                2 * X.
```

The first line of this module is mandatory. This line contains the module name and must match the file name that the module was saved as (i.e. tut.erl). The second line lists functions that are available form this module. They could be considered 'public'. If a function is defined in the module but not listed in the export command, then it is considered 'private' to the module, i.e. it can only be called by other functions within the module. The '/1' part of the export denotes that the 'double' function requires one

parameter. Finally the third and fourth lines implement the function. The third line states that if the module receives a request to execute a function called 'double' and is passed one parameter 'X' then (->) the output of the function is '2*X'. This highlights another feature of Erlang. Like some other languages (Ruby), the return value of a function is the last value it touched, which in this case is the result of the calculation '2*X'.

Compiling & Executing

Once a module has been constructed, it can then be compiled and executed using the shell. The command to compile a module is c(module). So using the module from the previous section, it would be compiled by executing c(tut). If compilation was successful, an 'ok' message should be outputted.

Once a module has been compiled successfully, its exported functions can then be executed. When executing a function within a module, the module and function need to be specified. To execute the 'double' function in the 'tut' module, the following command would be input – 'tut:double(5).'. This runs the 'double' function in the 'tut' module and passes in the parameter '5'. The result of this command will be 10.

1.2 Fundamentals

This section briefly discusses some of the fundamental components and concepts of the Erlang language and is intended for those who have no previous knowledge of Erlang.

Variables

Variables can be assigned like normal using the '=' operator. So 'X=2' is valid. An important point to note is that variables cannot be re-assigned. Once bound a variable has to be un-bound before it can be re-assigned. Trying to assign an already bound variable another value will generate an error.

Global Variables

Variables are only scoped to the function they are declared in. Variables can be given a more 'global' scope using put() and get(). To declare a variable in this way you would issue the following command…

- put(x, 5).

To subsequently access this value you would use get()…

- get(x).

Variables created in this way can also be re-assigned, i.e. put(x,10) is valid.

Atoms

Atoms are another data type in Erlang. Atoms start with a small letter, for example: 'charles', 'centimeter', 'inch'. Atoms are simply names, nothing else. They are not like variables which can have a value.

Tuples
A tuple is a compound data type with a fixed number of terms. Tuples allow you to group data together. Tuples can also contain tuples. Tuples start with a '{' and end with a '}'. Items in a tuple are separated by ','. The following is an example of a tuple…

- {moscow, {celsius, -10}}

This tuple contains an atom (moscow) followed by another tuple which contains the temperature (-10) and another atom denoting the temperature type.
The data in a tuple doesn't have to be related, they can be two independent values. Tuples have a number of built in functions which allow you to access and manipulate the it.. Tuples can be used to pass data around functions.

Lists
Whereas tuples group things together, we also want to be able to represent lists of things. A list is a compound data type with a variable number of terms. Lists start with a '[' and end with a ']'. Items in a list are separated by ','. The following is an example of a list…

- X = [1,2,3,4,5,6,7]

Lists also have a number of built in functions and other means of access. For example, if a list were defined as follows…

- [E1, E2 | R] = [1,2,3,4,5,6,7].

…E1 would be bound to the first value in the list – 1, E2 would be bound to the second value – 2 and R would be bound to the remainder of the list – 3,4,5,6,7.

If & Case Statements
Conditional statements are supported in Erlang using 'if' and 'case'. These are similar in syntax and look as follows…

```
if
        Condition 1 ->
                Action 1;
        Condition 2 ->
                Action 2;
        Condition 3 ->
                Action 3;
        Condition 4 ->
                Action 4
end
```

When using an 'if' statement, Erlang starts at the top and checks each clause in sequence until it finds a matching one. If no matching clause is found a runtime error is generated.

Looping
Erlang doesn't implement loops in the way some people would be accustomed to. Looping in Erlang is achieved through recursion. The following is a looping example, implemented using recursion, that prints the numbers 1 to 10 to the screen…

```
Go() ->
        Go(1).
Go(11) ->
        io:fwrite("Done", []);
Go(X) ->
        io:fwrite("~p~n", [X]),
        Go(X+1).
```

There are three functions defined in this module. The first function (Go()) is used to kick off the loop by calling 'Go(1)'. 'Go(X)' picks up the 'Go(1)' command and assigns 1 to X. The value of X is then printed to the screen and 'Go(X)' is called again, giving it the value 'X+1'. This recursion will continue until X reaches 11. At this point, 'Go(11)' is executed first which prints 'Done' and exits.
There are other options to achieve a loop. Lists for example have a number of built in functions, such as 'for_each()' which allows you to traverse the list.

2 Concurrency in Erlang

Erlang's main strength is support for concurrency. Erlang processes are neither operating system processes nor operating system threads, but lightweight processes somewhat similar to Java's original "green threads".
While threads are considered a complicated and error-prone topic in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks.

Creating a Process
Process in Erlang are created using the spawn() built-in function. The spawn command is defined as 'spawn(Module, Exported Function, [List of Arguments]).'. The spawn function returns an id representing the newly created process.

Message Passing
As stated, Erlang process do not share state. Communication between processes is achieved using Message Passing. Lists, tuples etc… can be passed between processes. Processes receive messages using the 'receive' command…

```
receive
        pattern1 ->
                actions1;
        pattern2 ->
                actions2;
        ....
```

```
        patternN
                actionsN
    end.
```

Erlang processes send messages using the '!' operator…

- Pid ! Message

To send a message from one process to another the sending process must know the process id (PID) of the intended recipient. The Message can be a list, tuple, atom etc…

3 Assignment Summary

For this assignment I implemented both sequential and concurrent versions of the Mandelbrot algorithm in Erlang.
For reference, the source code for each part is located in the following appendices...

- Appendix A – Complex Module
- Appendix B – Sequential Mandelbrot Module
- Appendix C – Concurrent Mandelbrot Module

3.1 Sequential Implementation

The C implementation of Mandelbrot used structs to store data. In place of structs I implemented these data holders as tuples. Each of the 'complex' functions, for example, takes one or more tuples as parameters.
The first aspect to this implementation that I tackled was to implement the 'complex*' functions from the C version. The 'complex*' functions are…

- mult
- plus
- abs
- smult

I implemented each of these as Erlang functions within a module called 'complex' (see Appendix A) with the file name 'complex.erl'. So to execute 'mult', you would use the command 'complex:mult({I,J}, {X,Y}).'.
Once these functions were in place I could then concentrate on converting the existing C code for the algorithm to its equivalent Erlang syntax.
The source code for this algorithm is located in the 'mandelbrot.erl' file or within Appendix B.
The entry point to execute this application is the 'main()' function. The 'main()' function sets up most of the data that will be used in the various functions. The height and width are 'put' into variables so they are accessible from other functions. The CENTER, SPAN, and BEGIN tuples are setup with the BEGIN tuple being stored in a globally accessible spot using 'put()'. Finally the NUMPIXELS variable is created.
I created a recursive set of functions called 'for_loop' which take the NUMPIXELS variable and iterates from 0 to NUMPIXELS.
For every iteration of the 'for_loop' function, it calls the 'generate(PIX)' function. The 'generate(PIX)' function is responsible for the main number crunching. This function makes use of various other functions to get the number of iterations and then converts this to a character. The two main functions it calls are 'calculate()' and 'getchar()'.
The 'calculate()' function uses the 'complex' module and its functions to iterate its value until it is greater than or equal to 2.0. It then stores the number of iterations that were required to get to 2.0 and returns this value to the 'generate(PIX)' function which converts that value to a character and outputs it to the screen. The 'calculate()' function uses a number of recursive functions to achieve this 'loop' functionality. These functions are called 'go()' and basically keep looping until the value >= 2.0.

The final function that was implemented is the 'getchar()' function. This function takes a parameter (which comes from the number of iterations to reach 2.0) and does a modulo calculation on it to convert it to a character.
When executed, output similar to Figure 1 is produced…

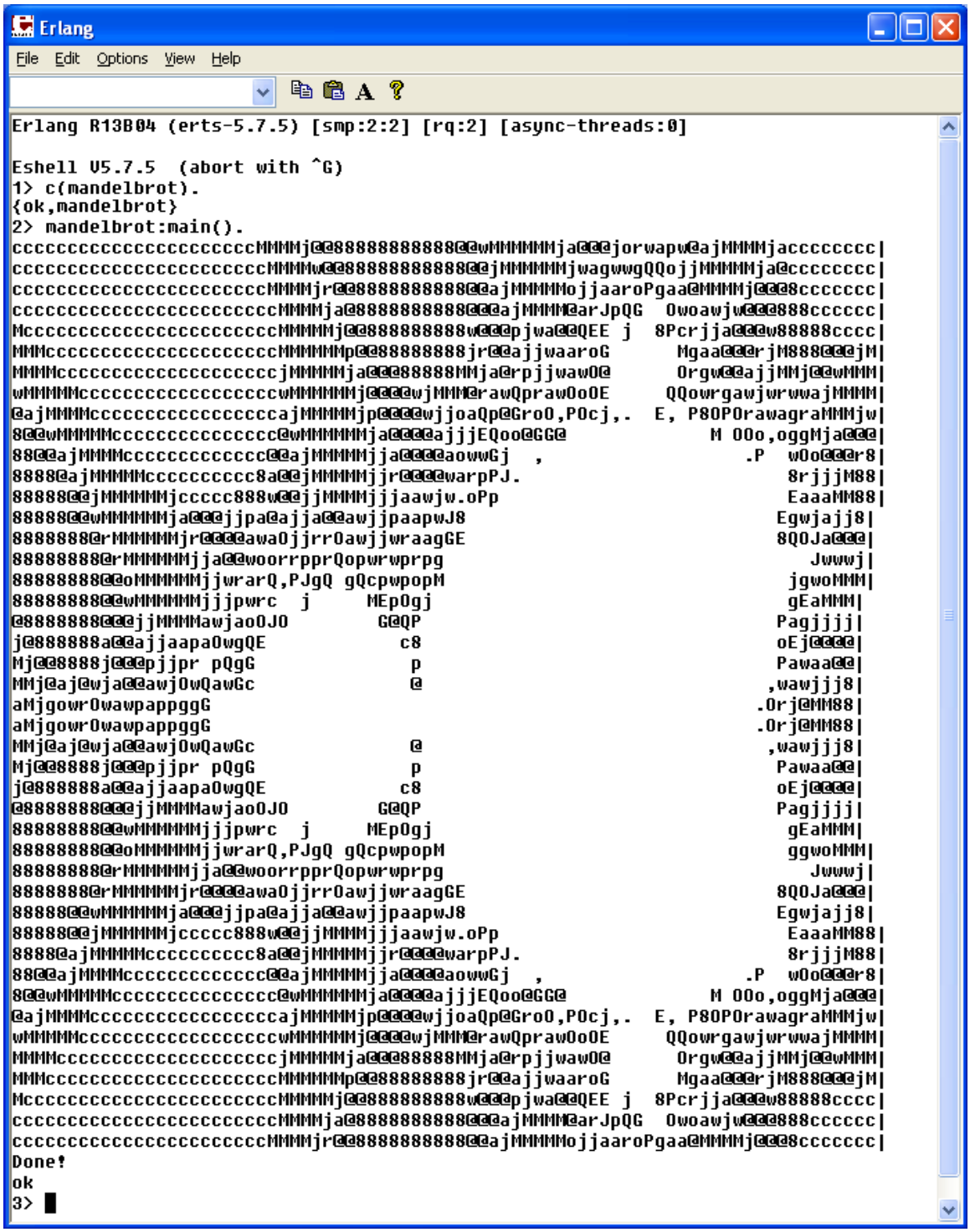

Figure 1: Erlang Mandelbrot Algorithm Output

3.2 Concurrent Implementation

Relatively minor changes were required to make the sequential version concurrent. The concurrent version can be viewed in file 'mandelbrot2.erl' or in Appendix C
The 'main()' function still sets up the initial data used in the algorithm. A new function called 'start()' was added which accepts Width, Height, Start and End as parameters. The 'main()' function kicks off each process and passes it the total width and height along with the start and end points. Each process is created using the 'spawn()' command, by specifying the module, function and arguments.
One point to note here is that, as stated, Erlang processes do not share state. So any data that a process needs should be passed to it either when it is spawned or while it is running. The main data that each process needs to know is the height, width, start and end points.
The start and end points are used to control the 'for_loop'. In the sequential version, the loop started at 0 and ended at NUMPIXELS. A simple implementation would be to split the work between 2 processes. So, for example, process 1 would have a start point of 0 and end point of NUMPIXELS/2, whereas process 2 would have a start point of (NUMPIXELS/2)+1 and an end point of NUMPIXELS. With this approach each of the processes is given half of the overall workload.

4 Comparison

This section compares the sequential implementations execution time to that of the concurrent implementation. Before presenting the results, there is some extra information that should be considered.

Erlang VM & SMP

The Erlang VM uses schedulers to pick runnable Erlang processes and IO jobs from the run-queue. On single-core machines and versions of Erlnag with no SMP support, the VM uses 1 scheduler which runs in the main process thread. This effectively means that multiple processes will not run efficiently as there is only one scheduler to handle these processes.
An Erlang VM with SMP support can have multiple schedulers, each of which runs in a separate thread. The schedulers are then able to pick jobs from a common queue.

Test Bed

All these tests were performed on a laptop with a dual-core processor. The version of Erlang used (5.7.5) includes support for SMP. By default, the Erlang VM recognised the availability of 2 cores and enabled 2 schedulers.

Execution Times

The execution times of both the sequential and concurrent algorithms were gathered using the built-in function 'timer:tc'. This function takes three parameters, the module, the function and any arguments/parameters for the function…

- timer:tc(mod, fun, args)

It is run from the command line. So instead of executing the Mandelbrot function directly (Mandelbrot:main().), it is executed using the timer function…

- timer:tc(Mandelbrot, main, []).

The timer:tc function returns a tuple which contains the execution time in microseconds and the output of the function.

Screen IO
Initial analysis of the concurrent execution times returned an average time that was greater than sequential version. This was unexpected, as the laptop these tests were being performed on is a dual-core laptop and the SMP Scheduler was set to 2.
I suspected that a cause of this anomaly might be that each process was generating its character and then fighting over access to output the character to the screen. To remedy this I modified both the sequential and concurrent algorithms to append the generated character to a list instead of outputting it to the screen as it was generated.

Sequential Algorithm
The sequential Mandelbrot algorithm, which outputted to the screen returned an average execution time of 2.4 seconds.
The non-IO version was found to have an average execution time of 1.9 seconds.

Concurrent Algorithm
The concurrent algorithm was tested with a varying number of processes.

- 2 Processes = 1.3 seconds
- 4 Processes = 1.4 seconds
- 6 Processes = 1.4 seconds

From the tests it is obvious that the limit for the number of processes is 2, when it is being run on a machine that has a dual-core processor and the VM is running 2 SMP Schedulers.

6 Conclusion

I found Erlang an interesting language to examine. Some of its fundamental concepts do require a little bit of patience to fully grasp, but overall it is relatively straightforward. Erlangs implementation of processes removes many of the headaches that are inherent in how other languages approach concurrency and, as can be seen from the execution times, deliver large increases in performance.

Appendix A – The Complex Module

```
-module(complex).
-export([mult/2, plus/2, abs/1, smult/2]).

mult({I, J}, {X, Y}) ->
        {I*X - J*Y, I*Y + J*X}.

plus({I, J}, {X, Y}) ->
        {I+X, J+Y}.

abs({X, Y}) ->
        TEMP = (X*X-Y*Y),
        if
                TEMP > 0 ->
                        math:sqrt(X*X - Y*Y);
                TEMP < 0 ->
                        math:sqrt(1)
        end.

smult(K, {X, Y}) ->
        {K*X, K*Y}.
```

Appendix B – Sequential Module

```
-module(mandelbrot).
-export([calculate/2, main/0]).

calculate({X, Y}, K) ->
        put(c, {X, Y}),
        put(totaliter, K),
        go({X, Y}).

go({X, Y}) ->
        go(0, {X, Y}).

go(1000, {X, Y}) ->
        "";

go(I, {X, Y}) ->
        % keep updating totaliter
        put(totaliter, I),

        % get complex:abs
        K = complex:abs({X, Y}),

        % if complex:abs >= 2.0 then break else keep looping
        if
                K >= 2.0 ->
                        % end loop
                        go(1000, {X, Y});
                K < 2.0 ->
                        Z = complex:plus(complex:mult({X, Y}, {X, Y}), get(c)),
                        go(I+1, Z)
        end.

for_loop(N, Length, _Inc, Acc) when N >= Length ->
   io:format("Done!~n");

for_loop(N, Length, Inc, Acc) ->
   for_loop(N+Inc, Length, Inc, [generate(N)]).

generate(PIX) ->
        % setup vars
        WIDTH = get(width),
        X = trunc(PIX rem WIDTH),
        Y = trunc(PIX/WIDTH),
        C = get(gbegin),
        {REAL, IMAG} = C,
        {SPANREAL, SPANIMAG} = get(gspan),
```

```
        G = {REAL+X*SPANREAL/(get(width)+1.0),IMAG+Y*
                SPANIMAG/(get(height)+1.0)},

        % get num iterations
        MAXITER = get(maxiter),
        calculate(G, MAXITER),
        N = get(totaliter),
        if
                N == MAXITER ->
                        N = 0;
                N < MAXITER ->
                        N = N;
                N > MAXITER ->
                        N = 0
        end,

        % get the character to output to the screen based on num iterations
        if
                N > 0 ->
                        I = getchar(N+1);
                N =< 0 ->
                        I = ''
        end,

        % output the character to the screen
        io:fwrite("~s", [I]),

        % if we are at width then add in a line break
        T = X+1,
        if
                (T) >= WIDTH ->
                        io:format("| ~n");
                (T) < WIDTH ->
                        io:format("")
        end.

% this function returns the character to display
getchar(N) ->
        L = N rem 20,
        if
                L < 1 -> ' ';
                L == 1 -> '.';
                L == 2 -> ',';
                L == 3 -> 'c';
                L == 4 -> '8';
                L == 5 -> 'M';
                L == 6 -> '@';
```

```
                L == 7 -> 'j';
                L == 8 -> 'a';
                L == 9 -> 'w';
                L == 10 -> 'r';
                L == 11 -> 'p';
                L == 12 -> 'o';
                L == 13 -> 'g';
                L == 14 -> 'O';
                L == 15 -> 'Q';
                L == 16 -> 'E';
                L == 17 -> 'P';
                L == 18 -> 'G';
                L == 19 -> 'J';
                L > 19 -> ' '
        end.

% start point
main() ->
        put(width, 78),
        put(height, 44),
        NUMPIXELS = get(width) * get(height),
        CENTER = {-0.7, 0},
        SPAN = {2.7, -(4/3.0)*2.7*get(height)/get(width)},
        put(gspan, SPAN),
        put(maxiter, 100000),

        BEGIN = complex:plus(CENTER, complex:smult(-1/2, SPAN)),
        put(gbegin, BEGIN),

        % loop for NUMPIXELS
        for_loop(0, NUMPIXELS, 1, 0).
```

Appendix C – Concurrent Module

```
-module(mandelbrot2).
-export([calculate/2, main/0, start/4]).

calculate({X, Y}, K) ->
        put(c, {X, Y}),
        put(totaliter, K),
        go({X, Y}).

go({X, Y}) ->
        go(0, {X, Y}).

go(1000, {X, Y}) ->
        "";

go(I, {X, Y}) ->
        % keep updating totaliter
        put(totaliter, I),

        % get complex:abs
        K = complex:abs({X, Y}),

        % if complex:abs >= 2.0 then break else keep looping
        if
                K >= 2.0 ->
                        % end loop
                        go(1000, {X, Y});
                K < 2.0 ->
                        Z = complex:plus(complex:mult({X, Y}, {X, Y}), get(c)),
                        go(I+1, Z)
        end.

for_loop(N, Length, _Inc, Acc) when N >= Length ->
   io:format("Done!~n");

for_loop(N, Length, Inc, Acc) ->
   for_loop(N+Inc, Length, Inc, [generate(N)]).

generate(PIX) ->
        WIDTH = get(width),
        X = trunc(PIX rem WIDTH),
        Y = trunc(PIX/WIDTH),
        C = get(gbegin),
        {REAL, IMAG} = C,
        {SPANREAL, SPANIMAG} = get(gspan),
        G = {REAL+X*SPANREAL/(get(width)+1.0),IMAG+Y*
```

```
                SPANIMAG/(get(height)+1.0)},

        % get maxiter
        MAXITER = get(maxiter),
        calculate(G, MAXITER),
        N = get(totaliter),
        if
                N == MAXITER ->
                        N = 0;
                N < MAXITER ->
                        N = N;
                N > MAXITER ->
                        N = 0
        end,

        % get character to display
        if
                N > 0 ->
                        I = getchar(N+1);
                N =< 0 ->
                        I = ''
        end,

        % get list
        L = get(list),

        % add character to list
        NEWLIST = lists:append(L, I),

        % put list
        put(list, [NEWLIST]).

getchar(N) ->
        L = N rem 20,
        if
                L < 1 -> ' ';
                L == 1 -> '.';
                L == 2 -> ',';
                L == 3 -> 'c';
                L == 4 -> '8';
                L == 5 -> 'M';
                L == 6 -> '@';
                L == 7 -> 'j';
                L == 8 -> 'a';
                L == 9 -> 'w';
                L == 10 -> 'r';
                L == 11 -> 'p';
                L == 12 -> 'o';
```

```
                L == 13 -> 'g';
                L == 14 -> 'O';
                L == 15 -> 'Q';
                L == 16 -> 'E';
                L == 17 -> 'P';
                L == 18 -> 'G';
                L == 19 -> 'J';
                L > 19 -> ' '
        end.

main() ->
        WIDTH = 78,
        HEIGHT = 44,
        NUMPIXELS = WIDTH * HEIGHT,
        J = NUMPIXELS/2,

        % spawn processes giving each process a portion of the work
        spawn(mandelbrot2, start, [WIDTH, HEIGHT, 0, NUMPIXELS/2]),
        spawn(mandelbrot2, start, [WIDTH, HEIGHT, 1717, NUMPIXELS]).

start(Width, Height, Start, End) ->
        put(width, Width),
        put(height, Height),
        CENTER = {-0.7, 0},
        SPAN = {2.7, -(4/3.0)*2.7*get(height)/get(width)},
        put(gspan, SPAN),
        put(maxiter, 100000),

        % setup list for characters
        put(list, []),

        BEGIN = complex:plus(CENTER, complex:smult(-1/2, SPAN)),
        put(gbegin, BEGIN),

        % loop for NUMPIXELS
        for_loop(Start, End, 1, 0).
```

References

The following sources were used during the course of this assignment.

1. Wikipedia
   http://en.wikipedia.org/wiki/Erlang_(programming_language)

2. Needless Complexity
   http://musings.memescraper.com/?p=16

3. Erlang Getting Started Guides
   http://www.erlang.org/starting.html

4. Erlang Manual
   http://www.erlang.org/download/getting_started-5.4.pdf

5. Scribd
   http://www.scribd.com/doc/44221/Thinking-in-Erlang

6. Inside the Erlang VM
   http://www.erlang.org/euc/08/euc_smp.pdf

7. Google Group
   http://groups.google.com/group/erlang-programming-book