A Report on the Feasibility of Using Elixir for Multiprocessing in the MMULT and TRAP Functions

By TJ Mundy, Tyler ___, and Daniel Forman

Introduction

For this report we will use Matrix Multiplication (MMULT) and the Trapezoidal Rule for Integration (TRAP) as benchmarks for evaluating the Elixir programming language's multiprocessing ability. We will also be comparing the Elixir language to several C implementations of the functions, the C implementations using PThreads and OpenMP for parallelization. The goal of this report is to establish whether Elixir can be used as a suitable language for parallelization, and further establish areas where it excels over a C implementation of the same functions.

Elixir is a dynamic, functional programing language, that runs on a BEAM VM, and implements the Erlang programming language. As a functional programming language, its focus is on performing function operations in a lightweight manner. Elixir uses lightweight threads to perform various operations, allowing systems to run many threads at a single time. These threads are also fault tolerant. For this reason, it is highly favored in web-development and used in embedded systems as well. Because of its predisposition towards multi-programming in its design, we expected Elixir to succeed very well in a multi-threaded design.

SYNTAX

Elixir is a functional programming language, which means it was difficult for us to grasp at first, coming from an imperative programming background (mainly Java and C).

Elixir does not use end of line statements such as semicolons, instead simply using the next line, or some pipeline operator to demonstrate that the data is meant to flow from one operation or another. Another aspect of Elixir that proved frustrating is that there are no loops, but recursion is used instead.

First let's discuss typing and some basic Elixir types. There are the standard basic primitives: numbers, chars, Strings, and there are several other basic types, namely atoms and tuples. Atoms are pieces of data whose name is the same as its type. It is best to think of them much like a string, but more primitive. Atoms are declared with a semicolon before the variable. For example :Pittsburgh, is an atom representing Pittsburgh. Atoms are useful for control checks such as switches and if statements. Tuples are another basic Elixir type, that are made up of other sets of primitive data. Tuples form the basis for most data storage in Elixir, such as Maps and Arrays. They have a constant time lookup, but a linear modification time because saved data is inherently immutable by default in Elixir.

First, let's look at a simple "Hello World" function:

```
IO.puts "Hello World"
```

This function will cause this statement to be written to the command line:

```
Hello World
:ok
```

This is a demonstration of Elixir's use of tuples in returning function calls. IO.puts calls the "puts" function in the "IO" module. Most elixir functions implicitly return functions as a tuple with the expected output and a status update from the function. The :ok atom is there to show that the function was successfully completed and encountered no errors.

Other aspects of Elixir that were crucial to completing our project were Data Streams and Tasks. Streams are a way of abstracting a series of transformations on a single piece of data. An example from our code is:

```
hold = Stream.map([i*1024+j], fn x -> (partial+Enum.at(inputArrayB, i*size+j)) end)
```

The call to Stream.map() shows what the final product of the function is meant to be. The value [i*1024+j] is how our code differentiated different locations in the matrix (Elixir has no nested array support, so we approximated to the best of our ability), and the other input "fn x -> (partial+Enum.at(inputArrayB, i*size+j)) end" is the transformation to be done on the matrix. The values "fn x ->" is a means for a function to be defined in place in Elixir. Since functional programming treats functions as a basic unit, this is necessary to avoid clutter when using a file.

Elixir has a variety of tools for implementing multiprogramming. The most simple is the spawn(fn x) function, which spawns a new task that runs the given function. We chose to use a Task abstraction of the spawn function. The Task allows for creating multiple threads of program and makes inter-thread communication easier for the calling process. Each Task stores the end value of the spawned process, which can be called when the main function needs it.

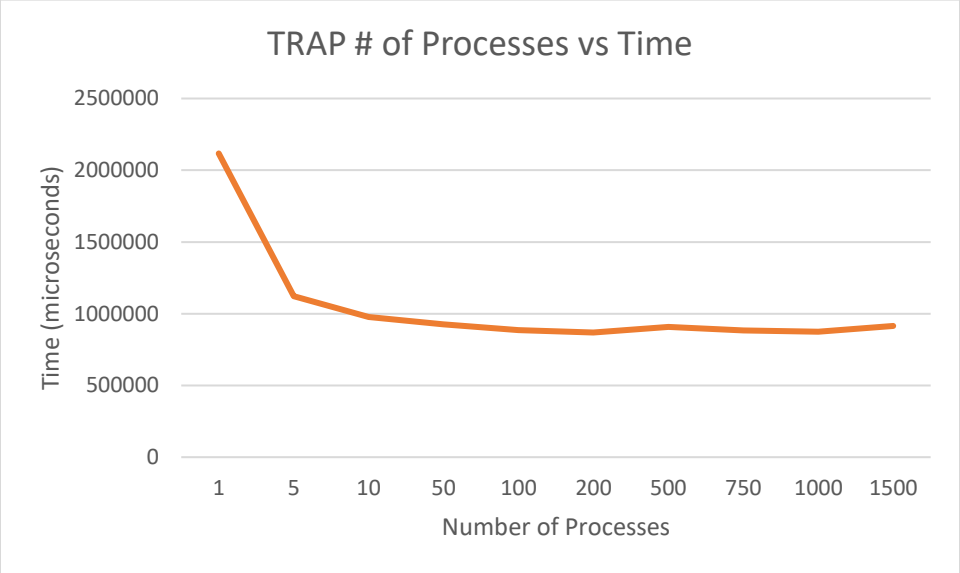
Comparison

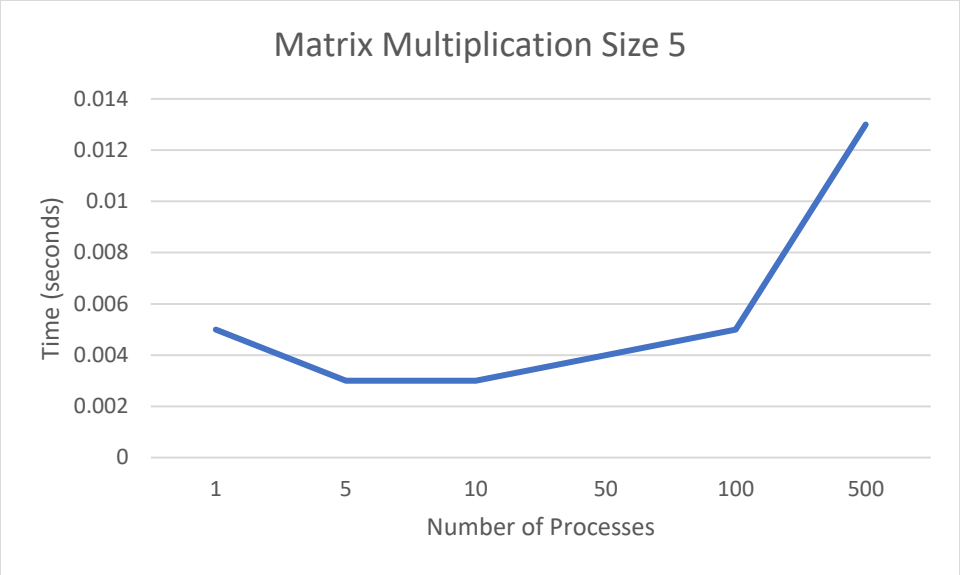
# of TRAP Processes	Time (microseconds)
1	2117000
5	1122000
10	978000
50	926000
100	885000
200	869000
500	907000
750	883000
1000	875000
1500	914000

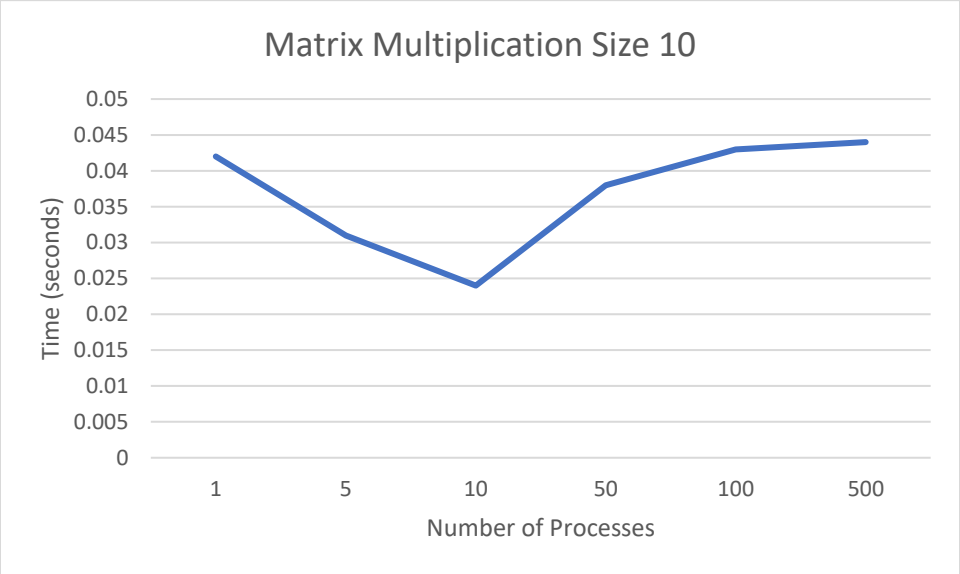
# MMULT Processes	Size	Time (seconds)
1	5	0.005
5	5	0.003
10	5	0.003
50	5	0.004
100	5	0.005
500	5	0.013

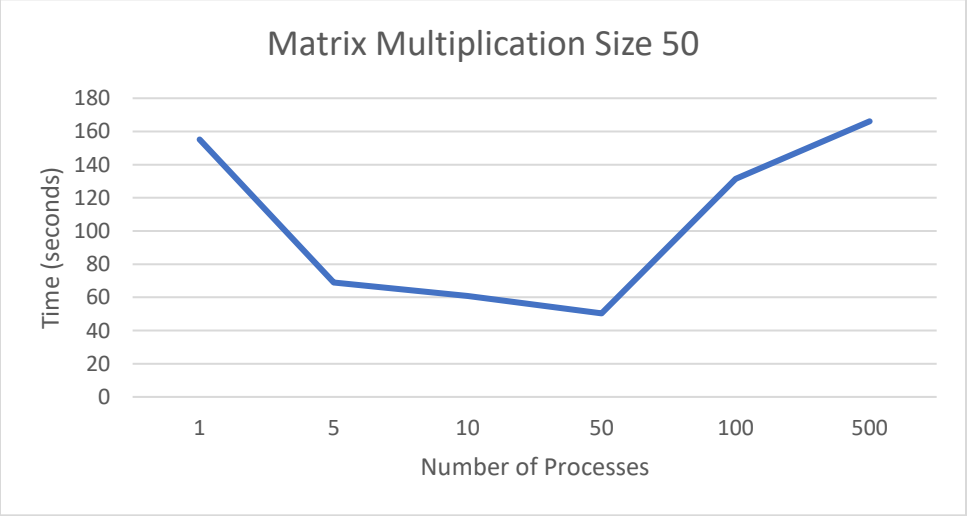
# MMULT Processes	Size	Time (seconds)
1	50	155.066
5	50	69.01
10	50	60.932
50	50	50.349
100	50	131.399
500	50	166.094

# MMULT Processes	Size	Time (seconds)
1	10	0.042
5	10	0.031
10	10	0.024
50	10	0.038
100	10	0.043
500	10	0.044




Matrix Multiplication Size 5
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
Time (seconds)
1
5
10
50
100
500
Number of Processes


Matrix Multiplication Size 10
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
Time (seconds)
1
5
10
50
100
500
Number of Processes


Matrix Multiplication Size 50
180
160
140
120
100
80
60
40
20
0
Time (seconds)
1
5
10
50
100
500
Number of Processes

Discussion

As mentioned previously, one of the hurdles with Elixir was that collectively, we had little experience with functional programming, so it made implementing the solution more difficult. The other major hurdle we encountered was that Elixir has no native support for nested arrays to implement the Matrix's data. We discussed a lot about the best way to implement the Matrix such that a) it would work with the best runtime, and b) that we could understand the implementation of it. Because of our lack of expertise with the language, that means that it could certainly be possible that our implementation is not the fastest. And as we can see from the data, speed certainly was a problem.

From the MMULT tests, we see that Elixir was extensively behind even the unparalleled C code, which we attribute to Elixir's immutable stored data. The problem we get with Elixir is that by transforming the data, each operation becomes extremely costly as we try to write a new matrix. C allows for transformations within the array, as each modification only affect that location in the array. Elixir's immutable data means that the entire object must be changed with each addition. This makes for extremely long runtimes with the stored data. We do see improvements from parallelization when comparing Elixir to itself, but the runtimes for the Elixir tests reach unsustainable levels at trivial problem sizes for C code.

Because we attribute the slowdown to Elixir's data storage, that means that we haven't truly evaluated Elixir's parallelization. From research we know that Elixir is known for its super lightweight threads, and that it is ill-suited to handle the storage of data that undergoes frequent changes.

//Add more once we see the TRAP Fucntion

Despite the shortcomings that we have found with Elixir, it definitely has its place within the realm of parallel programming. Creating new threads is easy and communicating between threads is also simple. Elixir provides a number of tools in the base language to make concurrent programming simple and easy to understand to the implementor. It is less simple than OpenMP and does require the implementor to understand the basics of parallel programming, but that also means that you can do more with the threads that are spawned. With that in mind, we would not recommend Elixir to be taught for this class. While it definitely has its place within a technology stack, with the types of examples we are running for this class, Elixir falls woefully short.

Conclusions

Elixir's major flaw as a language is its inability to be flexible with its weaknesses. Its strengths are its simplicity and lightweight threading, but its major weakness is in the immutability of its data. Some other functional languages, like Python, are able to offload extensive data operations into built in C code. While there are a number of Elixir libraries, we found to add this feature, we chose not to implement them since we felt it would defeat the purpose of the project. We don't expect that Elixir will change to better support mutable data storage, since the immutability has advantages for certain multithreading applications, but that remains one of our key worries