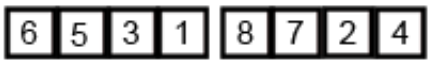
Parallel programming in Elixir

Introduction

Elixir is a functional programming language built on top of the Erlang VM. It's supposed to be great for developing highly concurrent, distributed and fault tolerant applications. We are masters student with emphasis on distributed systems so we were very excited to have some fun with Elixir. We decided to start out with something simple, doing parallel mergesort in Elixir. Doing the same in Erlang had shown that it was possible to get a very good speedup on mergesort. As such, we expected to see a similar speedup in Elixir, since both languages run on the same virtual machine.

Mergesort

For those who are not familiar with the merge sort algorithm, it´s an algorithm that sorts a list of integers by splitting up the list in half, then performs merge sort on both of the sub-lists. Once the lists have been split up until they only contain a single element. Then they are merged back together in sorted order. A visual representation of this can be seen in the following gif.



Sequential Mergesort

Here we have an implementation of a sequential mergesort. It consists of two functions merge and seq_merge_sort. The merge function is used to merge two ordered lists, it will be used in all of the different merge sort implementations in this tutorial. The seq_merge_sort function takes care of splitting up the lists, then recursively calling seq*merge*sort on both of the lists before merging them back together.

```elixir
defmodule SeqMergeSort do
    def merge([], y),           do: y
    def merge(x, []),           do: x
    def merge([x|xs], [y|ys])   do
        if x < y do
            [x|merge(xs, [y|ys])]
        else
            [y|merge(ys, [x|xs])]
        end
    end

    def seq_merge_sort([]), do: []
    def seq_merge_sort(x) do
        len = length(x)
        if len > 1 do
            {l1, l2} = Enum.split(x, round(len/2))
            merge(seq_merge_sort(l1), seq_merge_sort(l2))
        else
            x
        end
    end
end
```

Running work in parallel in Elixir

There are few different ways in Elixir to run things in parallel. We will be looking at the most simplest ways to do it, which are spawn_link and tasks.

Spawn link

Spawn link in Elixir works just like the Erlang spawn link function, it creates a new background process that can perform some work. Once the work is done it can be passed back as a message to the parent process.

```elixir
pid = spawn(Spawn2, :greet, [])
send pid, {self, "Hello World!"}

receive do
  {sender, msg} ->
    IO.puts message
end
```

In the example above it can be seen in line 1 how a new process is spawned. Line two demonstrates how to do message passing between processes and lines 4-7 show how to receive a message.

Tasks

Elixir tasks is simply an abstraction of the spawn function in Elixir. It allows you to add parallelism to your program in a very simple way. We can run the following code

```elixir
worker = Task.async(func)
results = Task.await(worker)
```

and this will result in a new process being created in the background which will run the function func that we pass into it. The result of that will be the id/pid of the process that got created, which we can then pass into Task.await() function that returns the results when the computation is finished.

Naive parallel mergesort using tasks

We created this parallel version of merge sort using tasks. It is almost exactly the same as the sequential version except that we use tasks to spawn a new process to perform the computations.

```elixir
defmodule NaiveParMergeSort do
    alias SeqMergeSort, as: Merge

    def naive_par_merge_sort([]), do: []
    def naive_par_merge_sort(x) do
        len = length(x)
        if len > 1 do
            {l1, l2} = Enum.split(x, round(len/2))
            w1 = Task.async(NaiveParMergeSort, :naive_par_merge_sort, [l1])
            w2 = Task.async(NaiveParMergeSort, :naive_par_merge_sort, [l2])
            Merge.merge(
                Task.await(w1),
                Task.await(w2))
        else
            x
        end
    end
end
```

Benchmarking and test data

Now in order to see if we are actually getting some speedup by running things in parallel we will have to benchmark both of the implementations. We created the Helpers module that contains two functions random_list and benchmark. random_list takes takes care of generating a random list of integers as test data, we pass into the function an upper bound of an interval that the numbers in the list should be within and n which specifies the size of the list. benchmark takes care of running the benchmark n number of times and returns the average run time of those n runs.

```elixir
defmodule Helpers do
    def random_list(interval, n) do
        Enum.map(1..n, fn _ -> :rand.uniform(interval) end)
    end

    def benchmark(n, module, fun, fun_input) do
        runs = for i <- 0..n, i > 0, do:
            :timer.tc(module, fun, fun_input)
            |> elem(0)
        Enum.sum(runs) / length(runs)
    end
end
```

Now let's run the benchmark in iex.

```
iex(1)> c("helpers.exs")
[Helpers]
iex(2)> c("seq_msort.exs")
[SeqMergeSort]
iex(3)> c("naive_par_msort.exs")
[NaiveParMergeSort]
iex(4)> test_data = Helpers.random_list(1000000, 200000)
[381705, 231607, 570899, 273191, 596392, 938350, 138406, 224800, 675115, 653381,
 932585, 944682, 651997, 557857, 64047, 181843, 468684, 284822, 14658, 975491,
 457776, 959020, 317133, 980863, 684949, 754667, 706702, 25833, 753176, 350774,
 985197, 502568, 601405, 224066, 858791, 465992, 79250, 397259, 210487, 214706,
 362727, 895811, 206303, 502644, 962127, 939182, 612425, 666950, 575519, 197109,
 ...]
iex(5)> Helpers.benchmark(10, SeqMergeSort, :seq_merge_sort, [test_data])
173464.4
iex(6)> Helpers.benchmark(10, NaiveParMergeSort, :naive_par_merge_sort, [test_data])
1159144.7
```

Bash

By looking at the results we can see that something is wrong with our parallel implementation, since it takes much longer time than the sequential version. For those who have done some parallel programming before this is probably not suprising. Even though the overhead of spawning new processes in very low in the Erlang virtual machine it still does not make sense to spawn a process for every single sub list that needs to be sorted. We will fix this by adding granularity.

Parallel mergesort with granularity using tasks

We add granularity to our code by introducing detph parameter to our sorting function. Once a certain depth is reached we stop spawning new processes and simply run it sequentially from that point.

Elixir

```elixir
defmodule GranParMergeSort do
    alias SeqMergeSort, as: Merge

    def gran_par_merge_sort([], _), do: []
    def gran_par_merge_sort(x, depth) do
        len = length(x)
        if len > 1 do
            {l1, l2} = Enum.split(x, round(len/2))
            if depth > 0 do
                w1 = Task.async(GranParMergeSort, :gran_par_merge_sort, [l1, depth - 1])
                w2 = Task.async(GranParMergeSort, :gran_par_merge_sort, [l2, depth - 1])
                Merge.merge(
                    Task.await(w1),
                    Task.await(w2))
            else
                    Merge.merge(gran_par_merge_sort(l1, 0), gran_par_merge_sort(l2, 0))
            end
        else
            x
        end
    end
end
```

Now if we run the benchmarks once again we will get the following results.

```
➜  erlang-tutorial git:(master) ✗ iex
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe]

Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("helpers.exs")
[Helpers]
iex(2)> c("seq_msort.exs")
[SeqMergeSort]
iex(3)> c("naive_par_msort.exs")
[NaiveParMergeSort]
iex(4)> c("gran_par_msort.exs")
[GranParMergeSort]
iex(5)> test_data = Helpers.random_list(1000000, 200000)
[381705, 231607, 570899, 273191, 596392, 938350, 138406, 224800, 675115, 653381,
 932585, 944682, 651997, 557857, 64047, 181843, 468684, 284822, 14658, 975491,
 457776, 959020, 317133, 980863, 684949, 754667, 706702, 25833, 753176, 350774,
 985197, 502568, 601405, 224066, 858791, 465992, 79250, 397259, 210487, 214706,
 362727, 895811, 206303, 502644, 962127, 939182, 612425, 666950, 575519, 197109,
 ...]
iex(6)> Helpers.benchmark(10, SeqMergeSort, :seq_merge_sort, [test_data])
173464.4
iex(7)> Helpers.benchmark(10, NaiveParMergeSort, :naive_par_merge_sort, [test_data])
1159144.7
iex(8)> Helpers.benchmark(10, GranParMergeSort, :gran_par_merge_sort, [test_data, 4])
71399.3
```

Bash

Now we can see that we get a 2.4 times speed up on the sequential version. We played around a little bit with the granularity but ended up getting the best performance by using depth 4.

Parallel mergesort with granularity using spawn link

We wanted to try to implement parallel mergesort with granularity using spawn link, so we could see the benefits of what the tasks abstraction actually provides. We ended up with the following code, which has few extra lines of code. Our opinion is that the abstraction gives us a more readable code.

Elixir

```elixir
defmodule ErlPsort do
    alias SeqMergeSort, as: Merge

    def start(x,depth) do
      erl_psort(self(),x,depth)
      receive do
        x -> x
      end
    end

    def erl_psort(pid,[], _), do: send pid, []
    def erl_psort(pid, x, depth) do
        len = length(x)
        if len > 1 do
            {l1, l2} = Enum.split(x, round(len/2))
            if depth > 0 do
                spawn_link(ErlPsort, :erl_psort, [self(),l1,depth-1])
                spawn_link(ErlPsort, :erl_psort, [self(),l2,depth-1])
                receive do
                  x ->  receive do
                            y -> send pid, Merge.merge(y,x)
                        end
                end
            else
                send pid, Merge.merge(Merge.seq_merge_sort(l1), Merge.seq_merge_sort(l2
            end
        else
            send pid, x
        end
    end
end
```

Now if we run the final benchmark

```
➜  erlang-tutorial git:(master) ✗ iex
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe]

Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("helpers.exs")
[Helpers]
iex(2)> c("seq_msort.exs")
[SeqMergeSort]
iex(3)> c("naive_par_msort.exs")
[NaiveParMergeSort]
iex(4)> c("gran_par_msort.exs")
[GranParMergeSort]
iex(5)> c("erl_psort.exs")
[ErlPsort]
iex(6)> test_data = Helpers.random_list(1000000, 200000)
[381705, 231607, 570899, 273191, 596392, 938350, 138406, 224800, 675115, 653381,
 932585, 944682, 651997, 557857, 64047, 181843, 468684, 284822, 14658, 975491,
 457776, 959020, 317133, 980863, 684949, 754667, 706702, 25833, 753176, 350774,
 985197, 502568, 601405, 224066, 858791, 465992, 79250, 397259, 210487, 214706,
 362727, 895811, 206303, 502644, 962127, 939182, 612425, 666950, 575519, 197109,
 ...]
iex(7)> Helpers.benchmark(10, SeqMergeSort, :seq_merge_sort, [test_data])
173464.4
iex(8)> Helpers.benchmark(10, NaiveParMergeSort, :naive_par_merge_sort, [test_data])
1159144.7
iex(9)> Helpers.benchmark(10, GranParMergeSort, :gran_par_merge_sort, [test_data, 4])
71399.3
iex(10)> Helpers.benchmark(10, ErlPsort, :start, [test_data, 4])
66794.7
```

Bash

we can see that skipping the abstraction gives us only a very small speed up and it is probably not worth it to sacrifice code readability for such a small gain.

In conclusion

Elixir looks like a great language for doing parallel programming and having that tasks abstraction is very nice. Finally we reccommend to always benchmark parallel code to see that it is working properly.

References

Mergesort gif: https://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif

Elixir logo: https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT2Kncd*AWN2DfPoSvv6sQieZtLlKG4YeLu5FuJn-8g*_fZt5uX

Elixir homepage: https://elixir-lang.org/

Book by Dave Thomas: Programming Elixir