Delta Stepping Algorithm in Parallel : A beginner approach. Documentation

Release 0.0.1

Ivan Felipe Rodriguez Rodriguez

May 17, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

Julia language is a high-level, high-performance dynamic programming language for technical computing. Between several other features it provides a sophisticated compiler and distributed parallel execution, that allows the user to code sophisticated applications. Given the novelty of the language, is hard to find documentation that help the beginners and new learners to understand the core concepts and advantages of using Julia. For this reason, and mostly as a new learner, I provide this work that show by example some usages and advantages of using Julia .

The example I will be refering to is the Delta-stepping algorithm. Which is a clever proposition developed by Madduri et. all. In their paper Parallel Shortest Path Algorihtm for Solving Large Scale Instances; along with the one by M. Kranjcevic $Delta$-Stepping Algorithm for Shared Memories Architectures.

This algorithm will be specially used to find the contour of an image. Using the package Images I converted the image to gray scale and calculate the gradient to compute a function cost. Later I set a directed graph where the pixels where the nodes and the cost to go to a neighbor pixel with different color was high, then using the delta-stepping algorithm I can find the shortest path, which was also the contour.

A parallel implementation in Julia of this algorithm is propposed. For this implementation I ran several test to evaluate it's performance. In particular three images were selected.

There are many changes that still have to be done. In the next version, I will be updating the implementation changing the buckets from a dictionary of sets(not very efficient) to a shared list (to provide better parallelism). As well I will be working on the tutorials so that clear and nicer examples can be shown.

Note: All the information provided here is based on the Official Documentation of Julia, any other source will be cited explicitely.

1.1 A Little Bit of Julia.

Instalation might be very tricky. However for learning I suggest you to download the binary extension that can be found Here

When you have it go to the folder *julia/bin* and simply execute *Julia*. Like:

```
julia
```

This initialization would start julia by default with one process. If you want to start with more processes you need to specify it with the following command (It makes sense when -p # correlates with the number of processors on your machine). For example on mine two cores.

```
julia -p 2
```

In case you forget to start with this command, you can add more procs from the julia command line:

```
addprocs()
```

Which will add all the available processors by default, but you can specify too just puting inside the parenthesis as many process as cores has your machine.

You can get at any momment the number of available workers by running the following code:

```
nprocs()
```

CHAPTER

TWO

JULIA PARALLELISM

Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI [1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

2.1 Low-level Parallelism

Parallel programming in Julia is built on two primitives: remote references and remote calls. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

These references, however, are low-level. Therefore is not recomendable to use them unless is very necessary for specific things. Usually higher level functions would provide very efficient performance.

Some basic functions are:

- *remotecall()*: Call a function asynchronously on the given arguments on the specified process. Returns a Future. Keyword arguments, if any, are passed through to func.
- *fetch()*: Waits and fetches a value from x depending on the type of x. Does not remove the item fetched:
- *@spawn*: Creates a closure around an expression and runs it on an automatically-chosen process, returning a RemoteRef to the result.
- *@spawnat*: Accepts two arguments, p(process) and an expression. A closure is created around the expression and run asynchronously on process p. Returns a Future to the result.

2.2 Examples:

In the following example the remotecall function would be used to automatically run the function fill *fill* which basically fills an array of 2 by 2 with numbers 3. Then we can fetch the result of that remote reference and store in a variable *ref*. Then using *@spawnat* we can specify where we want to implement certain function, in this case we want to multiply the array declared by 2, giving as a result a 2 by 2 array with numbers 6.

```
julia>ref=remotecall(2,fill,2,2,2)
julia>fetch(ref)
julia>ref2= @spawnat 1 2 .* fetch(ref)
julia>fetch(ref2)
```

2.3 High Level Parallelism

- @*parallel* (reducer) : Like the openmp *#pragma parallel for* This one is often used in for loops. HOWEVER IS IMPORTANT TO DEFINE THE REDUCER PROPERLY!
- *pmaps()* : This is a parallel execution for more complicated parallel initializations.
- *SharedArray* : This structure is used for shared memory computations
- @*time* : This profile tool measure the time. Shown After the execution.
- @*elapsed* : This profile tool measure the time, but does show the output time rather than the function.
- @*Allocated*: This profiling tool measure the allocations in memory.

2.4 Examples:

The following example would show why is important to be carefull when using @*parallel* using it without declaring a shared array might cause the following problem:

```
a = zeros(10)
@parallel for i=1:10
        a[i] = i
end
fetch(a)
```

This code did not work as expected because every process tried to write at the same time, causing an overwriting. To fix the problem you can create a Shared Array in the following way:

Another more complicated example :

Look at the jupyter notebook with the final presentation for more details.

CHAPTER

THREE

DELTA STEPPING ALGORITHM

During this work I implemented on Julia The delta stepping algorithm proppossed by [Meyer and Sanders] in 1998. This implementation split the Dijikstra algorithm in phases so that each phase can be implemented on parallel as well I followed [Pintarelli et.all] In order to get the idea of each part of the algorithm and the propper way to implemented.

The purpouse of this algorithm is to solve the so called SSSP problem (Single Source Shortest Path Problem). That can be stated as follows:

Given a weighted graph G=(V,E,c) where:

- V is a Set of vertices or nodes
- E is a set of edges, i.e., ordered pairs of nodes
- c cost or weight function $c: E rightarrow mathbb{N}$,

Find a minimal weight path from one chose node $s in V$ called the **source node**, to all other nodes in V. We say that the nodes $v,w in V$ are neighbours if $(v,w) in E$, i.e., if there exists and edge between them.

3.1 The pseudocode:

3.2 The implementation:

Here is the code propposed for the first algorithm:

```
    function Deltastep(list,s,delta)
    C=Dict()
    V=Set()
    tent=Dict()
    B=Dict{Any,Set}()
    E=Set()
    heavy=Dict()
    light=Dict()
    Req=[]
    pred=Dict()
    for item in list
        C[item[1],item[2]]=item[3]
        push!(V,item[1])
        push!(V,item[2])
        push!(E,[item[1],item[2]])
        heavy[item[1]]=[]
        heavy[item[2]]=[]
        light[item[1]]=[]
        light[item[2]]=[]
```

```
        tent[item[1]]=10000000000000
        tent[item[2]]=10000000000000
        pred[item[2]]=10000000000000
    end
    #println("C=$C")
    for e in E
        v=e[1]
        w=e[2]
        if C[v,w]>delta
            push!(heavy[v],e)
        else
            push!(light[v],e)
        end
    end
    #println("Heavy: $heavy")
    #println("Light: $light")
    #println("E=$E")
    relax(s,0,delta,B,tent,pred,1000000000)
    i=0
    while isempty(B)==false
        println("B=$B")
        println("tent=$tent")
        #Processing vertices from B[i]
        if haskey(B,i)==true
            S=Set()
            #Relax  recursively all light edges while they stay in B[i]
            while isempty(B[i])==false
                #Push to Req all light      edges from      vertices in B[i]
                Req=[]
                for v in B[i]
                    println("v=$v")
                    for e in light[v]
                        println("e  =$e")
                        push!(Req,[e[2],tent[v]+C[e[1]      ,e[2]],v])
                    end
                end
                #Update S
                union!(S,B[i])
                B[i]=Set()
                #Relax all  Req edges
                for r in Req
                    println("r=$r")
                    relax(r[1],r[2],delta,B,ten     t,pred,r[       3])
                end
            end
            #Relax all heavy edges of vertices in S
            Req=[]
            delete!(B,i)
            for v in S
                for e in heavy[v]
                    push!(Req,[e[2],tent[v]+C[e[1],e[2]],v  ])
                end
            end
            for r in Req
                relax(r[1],r[2],delta,B,tent,pred,r[3])
            end
        end
        i=i+1
```

```
    end
println("This is the answer: $tent")
return tent,pred
    end
```

The second algorithm was implemented in the following way:

```
function relax(w,d,delta,B,tent,pred,v)
    # println("w : $w")
    if d<tent[w]
        old_i=floor(tent[w]/delta)
        tent[w]=d
        if (haskey(B,old_i)==true)
            delete!(B[old_i],w)
        else
            # println("Warning: In relax (w=$w,d=$d,delta=$delta), B[old_i=$old_i] not found")
        end
        #Add w to new bin and update its tent
        new_i=floor(d/delta)
        if (haskey(B,new_i)==false)
            B[new_i]=Set()
            # println("Warning: In relax (w=$w,d=$d,delta=$delta), allocated B[fld=$new_i].")
        end
        push!(B[new_i],w)
        tent[w]=d
        if (haskey(pred,w)==false)
            pred[w]=()
            pred[w]=v
            # println("Warning: In relax (w=$w,d=$d,delta=$delta), allocated B[fld=$new_i].")
        else
            pred[w]=v
        end
        # println("The previous node visited was: $pred")
    end
end
```

3.3 Path Function

Some other functions were implemented. The following one, was implemented to retrieve the Path

```
function path(pred,w)
  v=w
  path = []
  while v<1000000000
        push!(path,v)
        #println("$path")
        v = pred[v]
  end
  return reverse(path)
end
```

3.4 Example:

The following is an example of how to use the algorithm.

Let us start by setting up the graph. To do so let us define the following array:

```
graph=Array{Int64}[]
push!(graph,[1,2,1])
push!(graph,[2,3,1])
push!(graph,[3,1,4])
push!(graph,[3,4,2])
push!(graph,[4,2,5])
```

Now we have a directed graph. Let us say we want to know the shortest path to each node from the vertice 2, we will do something like:

This is a simple code that runs over the graph, starting from the node 2 and with a delta of 1.

As is shown. It is obvious that going from node 2 to 2 has to have cost 0. The other nodes can be extracted very easy too.

CHAPTER

FOUR

IMAGES

Julia language is a high-level, high-performance dynamic programming language for technical computing. Between several other features it provides a sophisticated compiler and distributed parallel execution, that allows the user to code sophisticated applications. Given the novelty of the language, is hard to find documentation that help the beginners and new learners to understand the core concepts and advantages of using Julia. For this reason, and mostly as a new learner, I provide this work that show by example some usages and advantages of using Julia .

The example I will be refering to is the Delta-stepping algorithm. Which is a clever proposition developed by Madduri et. all. In their paper Parallel Shortest Path Algorihtm for Solving Large Scale Instances; along with the one by M. Kranjcevic $Delta$-Stepping Algorithm for Shared Memories Architectures.

This algorithm will be specially used to find the contour of an image. Using the package Images I converted the image to gray scale and calculate the gradient to compute a function cost. Later I set a directed graph where the pixels where the nodes and the cost to go to a neighbor pixel with different color was high, then using the delta-stepping algorithm I can find the shortest path, which was also the contour.

A parallel implementation in Julia of this algorithm is propposed. For this implementation I ran several test to evaluate it's performance. In particular three images were selected.

There are many changes that still have to be done. In the next version, I will be updating the implementation changing the buckets from a dictionary of sets(not very efficient) to a shared list (to provide better parallelism). As well I will be working on the tutorials so that clear and nicer examples can be shown.

Note: All the information provided here is based on the Official Documentation of Julia, any other source will be cited explicitely.

4.1 A Little Bit of Julia.

Instalation might be very tricky. However for learning I suggest you to download the binary extension that can be found Here

When you have it go to the folder *julia/bin* and simply execute *Julia*. Like:

```
julia
```

This initialization would start julia by default with one process. If you want to start with more processes you need to specify it with the following command (It makes sense when -p # correlates with the number of processors on your machine). For example on mine two cores.

```
julia -p 2
```

In case you forget to start with this command, you can add more procs from the julia command line:

```
addprocs()
```

Which will add all the available processors by default, but you can specify too just puting inside the parenthesis as many process as cores has your machine.

You can get at any momment the number of available workers by running the following code:

```
nprocs()
```