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:

```
a = SharedArray(Int64,10)
@sync @parallel for i=1:10
   a[i] = i
end
fetch(a)
```

Another more complicated example :

```
@everywhere function myrange(q::SharedArray)
    idx = indexpids(q)
    if idx == 0
        # This worker is not assigned a piece
        return 1:0, 1:0
    end
    nchunks = length(procs(q))
    splits = [round(Int, s) for s in linspace(0,size(q,2),nchunks+1)]
    1:size(q,1), splits[idx]+1:splits[idx+1]
end
# Here's the kernel
@everywhere function advection_chunk!(q, u, irange, jrange, trange)
    @show (irange, jrange, trange)  # display so we can see what's happening
    for t in trange, j in jrange, i in irange
        q[i,j,t+1] = q[i,j,t] +  u[i,j,t]
    end
    q
end
# Here's a convenience wrapper for a SharedArray implementation
@everywhere advection_shared_chunk!(q, u) = advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

```
    function advection_parallel!(q, u)
        for t = 1:size(q,3)-1
            @sync @parallel for j = 1:size(q,2)
                for i = 1:size(q,1)
                    q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                end
            end
        end
        q
    end
    function advection_shared!(q, u)
@sync begin
    for p in procs(q)
        @async remotecall_wait(advection_shared_chunk!, p, q, u)
    end
end
q
    end
```

```
q = SharedArray(Float64, (500,500,500))
u = SharedArray(Float64, (500,500,500))
# Run once to JIT-compile
advection_parallel!(q, u)
advection_shared!(q,u)
```

```
@time advection_parallel!(q, u);
@time advection_shared!(q,u);
```

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:

Algorithm 1 Pseudocode of the Δ-stepping algorithm.

```
1  function Δ-Stepping(V,E,c,s,Δ):
2
3    for each vertex v in V:
4    | heavy[v] ← {(v,w) ∈ E : c(v,w) > Δ}
5    | light[v] ← {(v,w) ∈ E : c(v,w) <= Δ}
6    | tent[v] ← ∞
7    end for
8    relax(s,0)
9    i ← 0
10
11   while B ≠ ∅:
12   | S ← ∅
13   | while B[i] ≠ ∅:
14   | | Req ← {(w,tent(v)+c(v,w)) : v ∈ B[i] and
                                      (v,w) ∈ light[v]}
15   | | S ← S ∪ B[i]
16   | | B[i] ← ∅
17   | | for each (w,d) ∈ Req: relax(w,d)
18   | end while
19   | Req ← {(w,tent(v)+c(v,w)) : v ∈ S and
                                    (v,w) ∈ heavy[v]}
20   | for each (w,d) ∈ Req: relax(w,d)
21   | i ← i+1
22   end while
23   return tent[]
24 end function
```

Algorithm 2 Pseudocode of the auxiliary relax function.

```
1  function relax(w,d):
2  |  if d<tent[w]
3  |  |  tent[w] ← d
4  |  |  B[⌊tent[w]/Δ⌋] ← B[⌊tent[w]/Δ⌋] \ {w}
5  |  |  B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {w}
6  |  end if
7  end function
```

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:

```
include("deltastep.jl")
include("relax.jl")
(tent,pred)=Deltastep(graph,2,1)
```

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.

3.4.1 Parallel Delta-Stepping

The implementation provided for the Delta stepping algorithm is the following:

```
function Deltastep(list,s,delta)
    @everywhere rdc(d::Vector,i::Vector) = rdc(rdc(Dict(),d),i)
    @everywhere rdc(d::Dict,i::Vector) = begin d[i[1]] = []; d[i[2]] = [];d end
    @everywhere rdc(d::Dict,i::Dict) = merge!(d,i)
    @everywhere rdt(d::Vector,i::Vector) = rdt(rdt(Dict(),d),i)
    @everywhere rdt(d::Dict,i::Vector) = begin d[i[1]] = 1000000000000000;d[i[2]] = 1000000000000000;
    @everywhere rdt(d::Dict,i::Dict) = merge!(d,i)
    @everywhere rdcc(d::Vector,i::Vector) = rdcc(rdcc(Dict(),d),i)
    @everywhere rdcc(d::Dict,i::Vector) = begin d[i[1],i[2]] = i[3]; d end
    @everywhere rdcc(d::Dict,i::Dict) = merge!(d,i)
    @everywhere rds(d::Vector,i::Vector) = rds(rds(Set(),d),i)
    @everywhere rds(d::Set,i::Vector) = begin push!(d,i[1]);push!(d,i[2]); d end
    @everywhere rds(d::Set,i::Set) = union!(d,i)
    @everywhere rde(d::Vector,i::Vector) = rde(rde(Set(),d),i)
    @everywhere rde(d::Set,i::Vector) = begin push!(d,[i[1],i[2]]); d end
    @everywhere rde(d::Set,i::Set) = union!(d,i)
    heavy = @parallel (rdc) for item in list
      item;
    end
    light=@parallel (rdc) for item in list
        item;
    end
    tent= @parallel (rdt) for item in list
        item;
    end
    pred= @parallel (rdt) for item in list
        item;
    end
    C = @parallel (rdcc) for item in list
        item
    end
    V = @parallel (rds) for item in list
    item
    end
    E= @parallel (rde) for item in list
        item
    end
    B=Dict{Any,Set}()
    Req=[]
    # println("Heavy: $heavy ")
    # println("C=$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("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,tent,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
```

3.5 Some considerations

One of the problems I found when trying to implement the algorithm in parallel, was the possibility to implement Shared memory data types for different structures like dictionaries, sets and others.

One of the solutions the community provided me when I asked on Stackoverflow was to create a set everywhere, populate it and in the end merge it. Doing this was helpful, however inefficient for memory allocations.

Let us see an example that clarifies this concept: We want to create a dictionary of sets. We want to set every vertex of the graph as a key and initialize it in parallel.

The first idea was simply to initialize a dictionary and then using *@parallel* populate it. Something like this:

```
julia> C=Dict()
julia> @parallel for item in a
                C[item[1],item[2]]=item[3]
            end
julia> println("$C")
```

This was the result:

Again we face the problem of overwriting, from the race conditions. For this reason was necessary to implement the following way:

```
julia> @everywhere rdc(d::Dict,i::Vector)= begin d[i[1]] = length(a); d end
julia> @everywhere rdc(d::Vector,i::Vector) = rdc(rdc(Dict(),d),i)
julia> @everywhere rdc(d::Dict,i::Dict) = merge!(d,i)
julia> C = @parallel (rdc) for item in a
                item
             end
julia> println("$C")
```

Implementing this way lead me to this desired result:

Several solutions like this one were useful to parallel implementation.

CHAPTER

FOUR

IMAGES

The images in this project are very important. Specially because from them will be extracted the graphs that would allow us to run relevant examples of the algorihtm.

The Images package developed by Tim Holy, was extensively used. Let us provide some insights about how this work.

The first step is to load the image. The file should be on the same path you are executing Julia. You use the following command that would load the image as an array of pixels:

```
julia> using Images, Colors, FixedPointNumbers, ImageView
julia> img=load("rose.png")
```

Now we can play with and "see" the data:

If you want to see the image you can use the following command:

```
julia>  view(img)
```

That would open an extra window showing

4.1 GrayScale

Now we can convert this image to gray scale, so it is easy to handle for calculations.

|rosegray|

4.2 Gradient

Now we can calculate gradient using the function from Images

```
julia> xx,yy=imgradients(A)
julia> c=(sqrt(xx.^2+yy.^2))
julia> view(c)
```

Now we need to find a way to reduce the cost of the middle of the images but make larger the cost of the border. To so I ran the following code:

4.3 Cost

```
julia> cost=1./(c.+1)
julia> view(cost)
```

Now if you go and see the cost of the borders they will be smaller thant those outside (they will be around 1). Therefore we will have the way to form the shortest path as the border of the image.

Now we are ready to set up the graph.

CHAPTER

FIVE

SET UP THE GRAPH.

In the last section I explained how to extract information from a picture and more important how to transform that information in order to make it relevant for our purposes.

In this section I will explain how I build the graph out of the image.

The directed graph we are trying to set is described by a list of items where each item has the following items (V1,V2, C) where V1 is connected with V2 (in that direction) and C is the cost of going from V1 to V2.

5.1 Pixels, vertex and edges

The nodes of our graph would be given by each pixel of the image. In future releases we will reduce significatively the number of pixels to describe a picture. Right now for the rose example we will be dealing with around 200.000 nodes. Heavy enough to make our algorithm to go for around a minute before getting the answer. In the future the idea is reduce those nodes to the half using zooming or different staff.

The edges are therefore a bigger problem. Each pixel has eight neigbours (I considered up,down, left, right and the corners), so there are 8 edges defined for each pixel. Setting the set of edges from our graph to around a million edges! This is just a huge ammount of information. In the following releases this will be reduced significatively, because this causes a poor performance of the application.

5.2 Cost function

As explained before the reciprocus of the gradient would be used to calculate the cost. Actually the cost would be given by the gradient in the ending pixel itself.

5.3 Setup function

The following code is used to setup the graph. The input is an image and the output is a graph in the structure previously explained.

5.4 Example

The following example ran over the rose picture would show the bast number of edges created by this approach. This would change in future releases.

CHAPTER

SIX

PROFILING THE PERFORMANCE:

Julia has *@time*, *@elapsed* and *@allocated* profiling tools. The firs one, time the application and shows the memory allocations but shows the result as well. The second one would only output the time and the third one would only output the memory allocations.

This code was implemented to time and measure the memory allocation of the algorithm.

6.1 Code for serial

```
using Images, Colors, FixedPointNumbers, ImageView
img=load("rose.png")
include("setup.jl")
include("deltastep.jl")
include("relax.jl")
include("path.jl")
graph=setup(img)
f=open("output.txt","w")
for i in [10,50,100,500,1000]
Time=@elapsed Deltastep(graph,5000,i)
Alloc=@allocated Deltastep(graph,5000,i)
write(f,"  Deltastep : $i Time Elapsed $Time Memory Allocations : $Alloc \n")
end
close(f)
```

6.2 Code for parallel

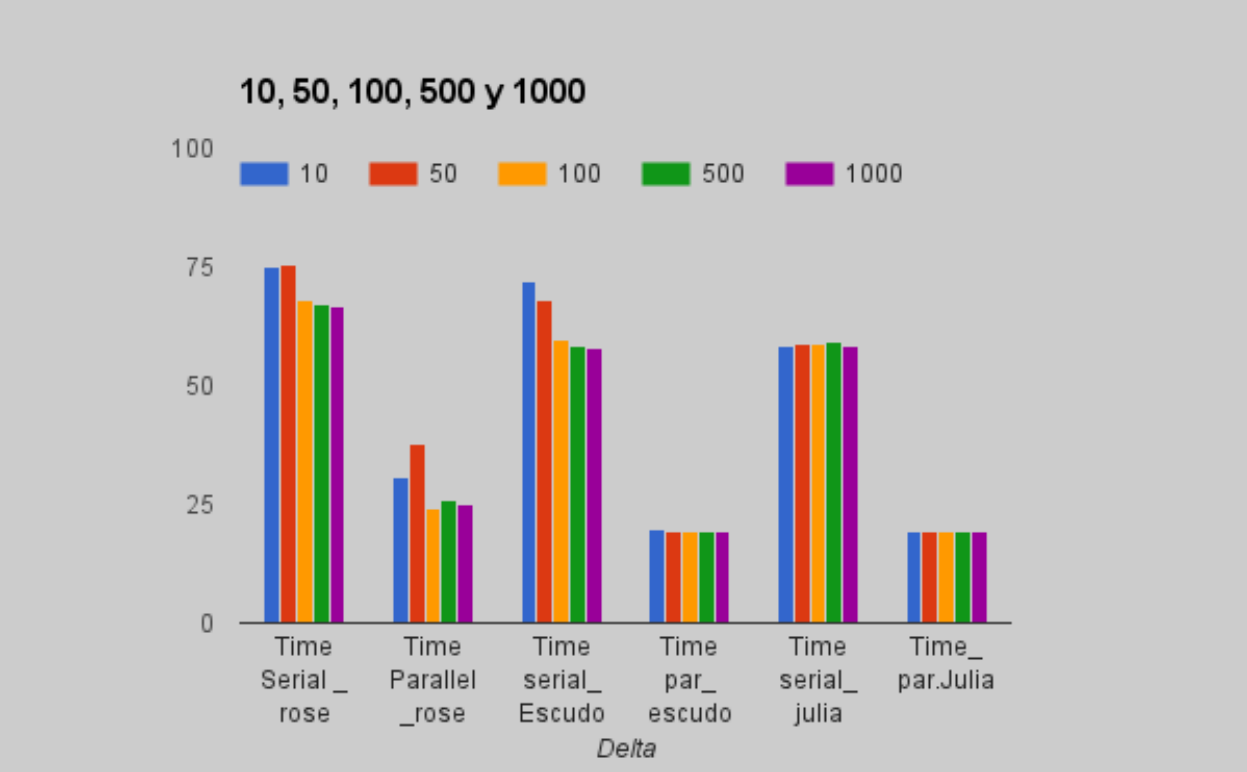
```
using Images, Colors, FixedPointNumbers, ImageView
addprocs()
img=load("Imagenes/rose.png")
include("setup.jl")
include("relax.jl")
include("path.jl")
include("deltastepar.jl")
graph=setup(img)
f=open("outputpar.txt","w")
for i in [10,50,100,500,1000]
Time=@elapsed Deltastep(graph,5000,i)
Alloc=@allocated Deltastep(graph,5000,i)
write(f,"Image :$size  Deltastep : $i Time Elapsed $Time Memory Allocations : $Alloc \n ")
end
close(f)
```

CHAPTER

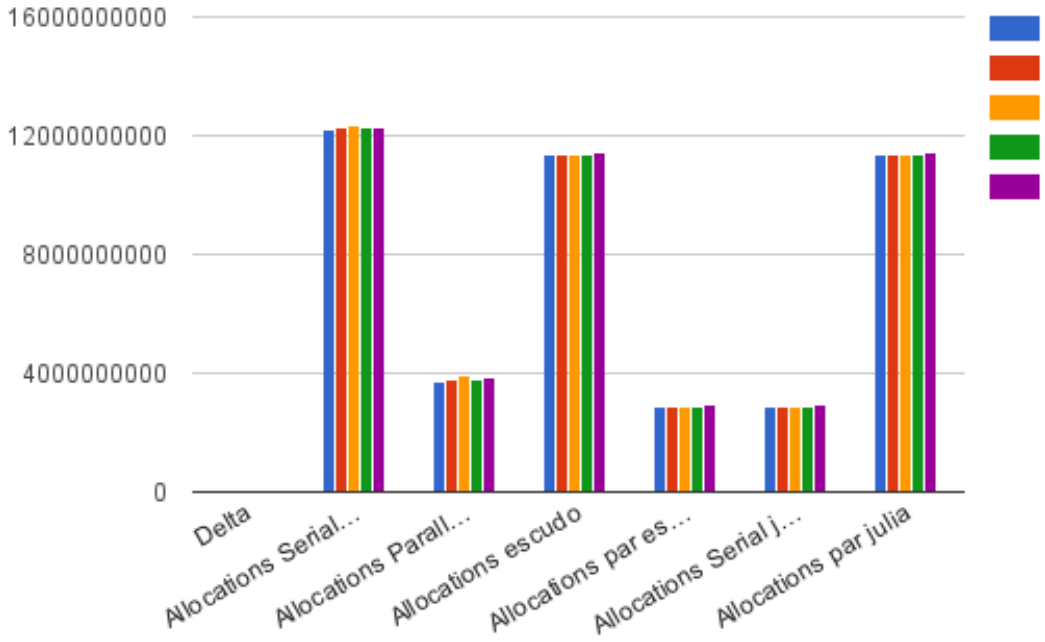
SEVEN

RESULTS

Running the profiling tools on the super computer Stampede I obtained the following results:



The parallel performance increased significatively. Even getting to a speed up of 2.9 which was closer to a linear performance.



The results of the memory were kind of high in parallel. This is due to the used of the memory when I am creating the arrays in every node, and then merging the branches into the principal one.

Since julia is still on development (but now is reaching 0.5 dev version) is not available yet on Stampede. I use the binaries available to run it on stampede. But I could not run it with more than 3 process. Everytime I try to run it with more cores I got “Could not allocate pools”.

CHAPTER

EIGHT

ANALYSIS

The performance observed on three processors of Stampede was about three times higher than in the serial code. This let us make conclusions on the weak scalling property of the code. However, due to the large size of the images, is important to notice that the performance was impacted great. In future releases this problem would be addressed. Specially using image compressing tools. that would let me describe the image very accurate with less pixels.

On the other hand, the problems I experienced running the code on Stampede shows that Julia is still on development and there are some compability problems in large scale computers. As well is necessary to increase the documentation coverage, specially for large scientific applications that requires more introduction for beginners.

However from the implementation is clear that Julia is a very dynamic language for programming. I have worked before in python, and it is easy to understand the new semantics given the proximity to that technical language in particular. In Images for instance, there is a comfortable and good proximity to matlab syntax.

CHAPTER

NINE

DEMO

As a demo I ran the code, using the node img[11637] as the beginning node and the node img[154455]. Then following the guide provided in the *quick reference* section I obtained the following image. I choose the view function of the gradient cost function so it is more visible the path found:

|rose1|

CHAPTER

TEN

QUICK REFERENCE

To launch the program right now you must follow the next steps.

1. Start julia.

```
$ Julia -p n
```

2. Load the packages needed:

```
julia> using Images, Colors, FixedPointNumbers, ImageView
```

3. Include the functions needed.

```
julia> Include("setup.jl")
julia> Include("deltastep.jl")
julia> Include("relax.jl")
julia> Include ("path.jl")
julia> Include("color.jl")
```

4. Load the image and set it up.

```
julia> img=load("Imagenes/rose.png")
julia> Graph=setup(img)
julia> tent,pred= deltastep(Graph,node,delta)
julia> path (pred, node)
julia> color(path,img)
```