ECS 158 Final Project: An Attempt At Parallelizing R's Phylobase::ShortestPath()

Raymond S. Chan, Alicia Luu, Bryan Ng
997544611, 997306859, 999999999
raschan@ucdavis.edu, ajuu@ucdavis.edu, bng@ucdavis.edu

Abstract

This report attempts paralleize CRAN's phylobase package's shortestPath function in RSnow, OpenMP and CUDA. The function takes in a phylogeneic tree, two nodes in the tree and produces the shortest path of nodes inbetween them. The RSnow implementation built on top fo the existant code and paralleized the descendants function.The OpenMP implementation did a similar approach, except with **¡ ALICIA WHAT DID YOU DO¿**. The CUDA implementation took the brute force approach to the shortestPath problem. Overall, **¡ SENTENCE ABOUT TEST RESULTS ¿**. **¡ CLOSE WITH A MORE GENERAL STATEMENT BUILDING OFF RESULTS OF THE TESTS ¿**

1 Introduction and Motivation

Alongside the rise of "big data" in the recent years, bioinformatices has gained considerable momentum. But the a consistent issues remain: what do we do with all the data and how do we make sense of it at a reasonable rate? The R community has taken a stab at those issues. For this report, we are examining the R's phylobase package, which provides the base class and functions for phylogenetic or evolutionary structures and comparitive data. (CITE) We will be focusing our efforts in making the "treewalk" utility functions, such as finding descendants/ancestors and shortest pathes, fast through three different parallel programming models (RSnow, OpenMP and CUDA).

2 Approaches

Only pseudocode or key chunks of code of each implementation are shown or described, see Appendix A for code details.

2.1 Original

The original R implementation calculated the shortest path betweeen the two nodes of interest by first calculating their Most Recent Common Ancestor (MRCA). Then that MRCA's descendants are calculated and compared to the the two nodes of interest's ancestors. Any overlap is stored and that is the shortest path.
The C version of the descendants function, called Cdescendant(), works by first marking given node in a preordered list of edges. Then the direct descendants (or children) of the given node is marked. Cdescendant() then iterates through the other edges and marks each marked node's direct descendants (children).

The C version of the ancestors function works the same as Cdescendants, expect direct *ancestors* (or parents) are marked instead of direct *descendants* (children).

Pseudocode (source code in original phylobase package):

```
descendants(tree, given_node){
   #let x be the edges of tree listed in PREORDER,
   #with the older node occupying the first column
   # C function call
   isDescendant <- Cdescendants(x[,1], x[,2], given_node)
   retval <- getNode(tree, isDescendant)
}

ancestors(tree, node1){
   #let x be the edges of tree listed in POSTORDER,
   #with the older node occupying the first column
   # C function call
   isAncestor <- Cancestors(x[,1], x[,2], given_node)
   retval <- getNode(tree, isAncestor)
}

MRCA(tree, node1, node2 ... noden){
   nodes <- unique(node1, node2, ..., noden)
   ancests <- lapply(nodes, ancestors, phy=phy, type="ALL")
    retval <- getNode(phy, max(Reduce(intersect, ancests)))
}
shortestPath(tree, node1, node2){
     t1 <- getNode(tree, node1)
     t2 <- getNode(tree, node2)

     # most recent common ancestor
     comAnc <- MRCA(tree, t1, t2)
     desComAnc <- descendants(tree, comAnc)

     # path: common ancestor to t1
     ancT1 <- ancestors(x, t1)
     path1 <- intersect(desComAnc, ancT1)

     # path: common ancestor to t2
     ancT2 <- ancestors(x, t2)
     path2 <- intersect(desComAnc, ancT2)

     # union of the path above paths
     retval <- union(path1, path2)
}
```

2.2 RSnow

The RSnow implemenation builds on top of original R version by paralleizing the descendants function. In order to make independent subproblems, for a given node, every other node keeps marching upwards to its ancestors until they either reached the root or encountered the given node. We were unable to parallelize the ancestor function. The original R version seems to have taken the most efficent serial approach for calculating a given node's ancestors.

Pseudocode: (see code in Appendix A.1)

```
descendants(tree, given_node){
    #let x be the list of all nodes except the given_node
    for i from 1 to height of tree
        if x == given_node -> append node to retval
        update x with its correponding ancestor
    return retval
}
```

2.3 OpenMP

ALICIA WRITE STUFF HERE

See code in Appendix A.2.

2.4 CUDA

Our CUDA implementation of the shortestPath function utilizes the GPU to find all ancestors of a given pair of nodes and then construct the shortest path between them. Our implementation assumes CSIF's pc43's resources, which are 1024 threads per block and 1 GB of global memory. We assume the given data can fit in our GPUs global memory. This assumption may limit the test we will be able to perform. Our solution utilizes the fact that the shortest path between two nodes in a tree must converge at the lowest common ancestor of both nodes. In cases, where one node is an ancestor of another, then the shortest path is then found by traversing the parents of the child node. We parallelized our code by finding both sets of ancestors of the given nodes at the same time. Since neither node needs to know about the other to find its own ancestors, this problem can be done independently of each other. Both sets of ancestors are then traversed to find the shortest path. We were unable to parallelize this part of the solution since each list of ancestors must be checked to find overlapping elements.

See code in Appendix A.3.

3 Experiment Results

These are the results of running the above scripts with a simplified internet (i.e. n = 6).

```
>> i = [ 2 6 3 4 4 5 6 1 1];
>> j = [ 1 1 2 2 3 3 3 4 6];
>> n = 6;
>> G = sparse(i,j,1,n,n);
>> Finaltimetest
pagerank1          pagerank2          pagerank3A         pagerank3B         pagerankpow
-----------------------------------------------------------------------------------------

  0.0002            0.0001             0.0002             0.0003             0.0004
```

These are the results of running the above scripts with the Harvard500 dataset (i.e. n = 500).

```
>> load Harvard500
>> Finaltimetest
```

```
pagerank1          pagerank2          pagerank3A          pagerank3B         pagerankpow
-----------------------------------------------------------------------------------------
  0.0024             0.0329             0.0226              0.0011             0.0255
```

These results are consistent with the dicussion above.

4 Discussion

4.1 RSnow

TALK ABOUT BIG O's.

4.2 OpenMP

TALK ABOUT BIG O's.

4.3 CUDA

THIS IS JUST A GUESS.
Compared to the serial version, the cuda implementation performed slower in most test cases. While the cuda version can compute both given nodes ancestors at the same time, it must also load the entire tree into the GPUs memory.

5 Conclusion

The PageRank algorithm was the starting point of Google's rise to fame. It was able to numerically quantify the "quality" on links/web pages of the internet. Pagerank is a Markov chain for which we solve for the dominant eigenvector of its transition probability matrix. There are two main methods of solving such a system of linear equations. The Power method is shown here to be the best method because of its efficiency in run time and memory usage. The run times are decent, as Hopcraft stated, it varies logarithmically with the size of input (n web paes). For the current day, the sheer amount of data that needs to processed is daunting. Further studies on PageRank could be done in further optimizing its space usage.

6 Acknowledgements

We would like to thank Professor Norman Matloff for his guidance and knowledge presented during lectures. This work is the result of a final project for ECS 158 Winter Quarter 2015. His open-source textbook, blog and various tutorial were an essential part of our learning. We would like to thank the teaching assistance Shengren Li for offering invaluble advice and feedback on our codes (especically our CUDA) throughout the quarter.

7 Appendix

A Codes

INSERT ALL CODES HERE ALONG WITH A PARAGRAPH EXPLAINING IT

A.1 RSnow Code

```
SNOW <- function(x,size,root,type=c("descendants")){
    ans <- rep(0,size)
    mystart <- (myid-1)*length(x)+1
    myend <- myid*length(x)

    type <- match.arg(type)
    if (type == "descendants"){
        v1 <- descendant
        v2 <- ancestor
        #initalization
        temp <- v1[mystart:myend]

        #second and beyond iteration
        for (j in 1:size){
            if (node %in% temp){
                setthese <- which(temp == node) + mystart-1
                ans[setthese] <- 1
            }
            blah <- rep(-1,length(temp))
            for (i in (1:length(temp))){
                matched_pos <- which(v1 == temp[i])
                if (length(matched_pos) != 0){
                    blah[which(temp == temp[i])] <- matched_pos
                }
                else{#matched_pos == 0
                    ## R is 1 INDEXED!
                    if (type == "descendants"){
                        blah[i] <- 1
                    }
                }
            }#for i
            #"go to your parents set"
            difference <- length(temp) - length(v2[blah])
            temp <- v2[blah]
            if (difference > 0){
                temp <- c(rep(0,difference),temp)
            }
            if (node %in% temp){
                setthese <- which(temp == node) + mystart-1
                ans[setthese] <- 1
            }
        }#j loop
    }#new endif for type==descendants
    return(ans)
}# end SNOW

setmyid <- function(i){
    myid <<- i
}
```

```r

## get descendants with RSnow
RSnowdescendants <- function (phy, node, type=c("tips","children","all"),cls) {
    type <- match.arg(type)

    ## look up nodes, warning about and excluding invalid nodes
    oNode <- node
    node <- getNode(phy, node, missing="warn")
    isValid <- !is.na(node)
    node <- as.integer(node[isValid])

    if (type == "children") {
        res <- lapply(node, function(x) children(phy, x))
        ## if just a single node, return as a single vector
        if (length(res)==1) res <- res[[1]]
    } else {
        ## edge matrix must be in preorder for the C function!
        #if (phy@order=="preorder") {
            edge <- phy@edge
        #} else {
        #    edge <- reorder(phy, order="postorder")@edge
        #}
        ## extract edge columns
        ancestor <- as.integer(edge[, 1])
        descendant <- as.integer(edge[, 2])

        ## return indicator matrix of ALL descendants (including self)
        #isDes <- .Call("descendants", node, ancestor, descendant)
        clusterExport(cls,c("node", "ancestor", "descendant","setmyid","SNOW"),
            envir=environment())
        dexgrps <- splitIndices(length(ancestor),length(cls))
        rootdex <- which(phy@edge[,1] == 0)
        clusterApply(cls,1:length(cls),setmyid)
        newisDes <- clusterApply(cls,dexgrps,SNOW,length(ancestor),rootdex,      "
            descendants")
        isDes <- (matrix(Reduce('+',newisDes),nrow=length(ancestor),ncol=1))
        storage.mode(isDes) <- "logical"
        ## for internal nodes only, drop self (not sure why this rule?)
        int.node <- intersect(node, nodeId(phy, "internal"))
        isDes[cbind(match(int.node, descendant),
            match(int.node, node))] <- FALSE

        ## if only tips desired, drop internal nodes
        if (type=="tips") {
            isDes[descendant %in% nodeId(phy, "internal"),] <- FALSE
        }
        ## res <- lapply(seq_along(node), function(n) getNode(phy,
        ##     descendant[isDes[,n]]))
        res <- getNode(phy, descendant[isDes[, seq_along(node)]])
    }
    ## names(res) <- as.character(oNode[isValid])

    res
}

###############
# shortestPath
###############

```

```
RSnowshortestPath <- function(phy, node1, node2,cls){

    ## conversion from phylo, phylo4 and phylo4d
    if (class(phy) == "phylo4d") {
        x <- extractTree(phy)
    }
    else if (class(phy) != "phylo4"){
        x <- as(phy, "phylo4")
    }
    ## some checks
    t1 <- getNode(x, node1)
    t2 <- getNode(x, node2)
    if(any(is.na(c(t1,t2)))) stop("wrong node specified")
    if(t1==t2) return(NULL)

    ## main computations
    comAnc <- MRCA(x, t1, t2) # common ancestor
    desComAnc <- RSnowdescendants(x, comAnc, type="all",cls)
    ancT1 <- ancestors(x, t1, type="all")
    path1 <- intersect(desComAnc, ancT1) # path: common anc -> t1

    ancT2 <- ancestors(x, t2, type="all")
    path2 <- intersect(desComAnc, ancT2) # path: common anc -> t2

    res <- union(path1, path2) # union of the path
    ## add the common ancestor if it differs from t1 or t2
    if(!comAnc %in% c(t1,t2)){
        res <- c(comAnc,res)
    }

    res <- getNode(x, res)

    return(res)
} # end shortestPath
```

A.2 OpenMP Code

```
ALICIA CODE GOES HERE
```

A.3 CUDA Code

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

//nvcc shortestPath.cu

struct node
{
    int nodeID, ancestor;
    char label[20]; //max size of label is 20
};//node

void setNode(node &phy, int numNodes, int id, int aID, char * label)
{
    phy.nodeID = id;
    phy.ancestor = aID;
    memset(phy.label, '\0', sizeof(label));
    strcpy(phy.label, label);
}//setNode

__global__ void kernel(node * array, int numNodes, int id1, int id2,
                       int * ancestorID1, int * ancestorID2)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < numNodes)
    {
        if (array[idx].nodeID == id1) //if found target node
        {
            int ancestorIndex = 0;
            node temp = array[idx]; //start from current node
            while (temp.ancestor != 0)
            {
                ancestorID1[ancestorIndex++] = temp.ancestor; //add all ancestors
                for (int i=0; i<numNodes; i++)
                {
                    if (array[i].nodeID == temp.ancestor)
                    {
                        temp = array[i];
                        break;
                    }//if
                }//for
            }//while
        }//if
        else if (array[idx].nodeID == id2) //if found target node
        {
            int ancestorIndex = 0;
            node temp = array[idx]; //start from current node
            while (temp.ancestor != 0)
            {
                ancestorID2[ancestorIndex++] = temp.ancestor; //add all ancestors
                for (int i=0; i<numNodes; i++)
                {
                    if (array[i].nodeID == temp.ancestor)
                    {
                        temp = array[i];
                        break;
                    }//if
                }//for
            }//while
        }//if

    }//if
}//kernel

void shortestPath(node * phy, int numNodes, char * label1, char * label2)
{
    node * deviceArray;
    int * deviceID1;
    int * deviceID2;
    int * ancestorID1 = new int[numNodes]; //initialize max size to number of nodes
    int * ancestorID2 = new int[numNodes];
    float blockSize = 1024; //num threads per block

    //check if invalid query
    node temp1, temp2;
    for (int i=0; i<numNodes; i++)
    {
        ancestorID1[i] = 0;
        ancestorID2[i] = 0;
        if (strcmp(label1, phy[i].label) == 0)
            temp1 = phy[i];
        else if (strcmp(label2, phy[i].label) == 0)
```

```
            temp2 = phy[i];
    }//for

    if ((temp1.ancestor == temp2.nodeID) || (temp2.ancestor == temp1.nodeID))
    {
        printf("named integer(0)\n");
        return;
    }//if

    //allocate device memory
    cudaMalloc(&deviceArray, sizeof(node) * numNodes);
    cudaMalloc(&deviceID1, sizeof(int) * numNodes);
    cudaMalloc(&deviceID2, sizeof(int) * numNodes);
    cudaMemcpy(deviceArray, phy, sizeof(node) * numNodes, cudaMemcpyHostToDevice);
    cudaMemcpy(deviceID1, ancestorID1, sizeof(int) * numNodes, cudaMemcpyHostToDevice);
    cudaMemcpy(deviceID2, ancestorID2, sizeof(int) * numNodes, cudaMemcpyHostToDevice);

    dim3 dimBlock(blockSize);
    dim3 dimGrid(ceil(numNodes/blockSize));

    //compute ancestors
    kernel <<< dimGrid, dimBlock >>> (deviceArray, numNodes, temp1.nodeID, temp2.nodeID, ↩
        deviceID1, deviceID2);
    cudaMemcpy(ancestorID1, deviceID1, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
    cudaMemcpy(ancestorID2, deviceID2, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
    cudaFree(deviceArray);
    cudaFree(deviceID1);
    cudaFree(deviceID2);

    //find shortest path
    int * path = new int[numNodes];
    int currentPath = ancestorID1[0];
    int pathIndex = 0;
    bool isLCAPath = false;
    //check if path converges at LCA
    for (int i=0; i<numNodes; i++)
    {
        path[i] = 0;
        if (temp1.nodeID == ancestorID2[i])
        {
            for (int j=0; j<i; j++)
                path[j] = ancestorID2[j];
            isLCAPath = true;
            break;
        }//if
        else if (temp2.nodeID == ancestorID1[i])
        {
            for (int j=0; j<i; j++)
                path[j] = ancestorID1[j];
            isLCAPath = true;
            break;
        }//else if
    }//for

    //if one node is the ancestor of another
    if (!isLCAPath)
    {
        for(int i=0; i<numNodes; i++)
        {
            for (int j=0; j<numNodes; j++)
            {
                if (currentPath == ancestorID2[j])
                    break;
                if ((ancestorID2[j] == 0) || (j == numNodes-1))
                {
                    path[pathIndex++] = ancestorID1[i];
                    currentPath = ancestorID1[i];
                    break;
                }//if
            }//for
        }//for

        if (pathIndex == 0)
            path[pathIndex++] = currentPath;

        for (int i=0; i<numNodes; i++)
        {
            if (ancestorID2[i] == currentPath)
                break;
            path[pathIndex++] = ancestorID2[i];
        }//for
    }//if
    for (int i=0; i<numNodes; i++)
    {
        if (path[i] == 0)
            break;
        for (int j=0; j<numNodes; j++)
        {
            if (path[i] == phy[j].nodeID)
            {
                printf("%s ", phy[j].label);
                break;
            }//if
        }//for
    }//for

    printf("\n");
    for (int i=0; i<numNodes; i++)
    {
        if (path[i] == 0)
            break;
        printf("%d ", path[i]);
    }//for
    printf("\n");

    delete [] ancestorID1;
    delete [] ancestorID2;
    delete [] path;
}//shortestPath
```

B Who Did What

Alicia wrote the OpenMP implementation. Bryan wrote the CUDA implemenation. Raymond wrote the RSnow implementation. We worked on running tests and writing the report in LaTeX.

References

[1] C. Moler, *Numerical Computing with MATLAB Revised Reprint* 2004.

[2] L. Page, S. Brin, R. Motwani, and T. Wingograd, *The PageRank Citation Ranking: Bringing Order to the Web*, avaiable at http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf, 1998

[3] J. Hopcraft and R. Kannan, *Foundations of Data Science*, available at http://www.cs.cornell.edu/jeh/NOSOLUTIONS90413.pdf, 2011