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

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

Abstract

This report attempts to parallelize CRAN's phylobase package's shortestPath function in RSnow, OpenMP and CUDA. The function takes in a phylogeneic tree and produces the shortest path between two nodes. The RSnow implementation is built on top of the existing code and parallelizes the descendants function. The OpenMP implementation took a similar approach by parallelizing both the ancestors and descendants C functions. The CUDA implementation took the brute force approach to the shortestPath problem. Overall, our attempt to parallelize the shortest path function was partially successful. The original version has shown that it is quite fast and efficient in finding the shortest path.

1 Introduction and Motivation

Alongside the rise of "big data" in the recent years, bioinformatics has gained considerable momentum. But a consistent issue remains: What do we do with all the data and how do we make sense of it at a reasonable rate? The R community has attempted to resolute these 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 comparative data. We will be focusing our efforts in making the "treewalk" utility functions, such as finding descendants/ancestors and shortest paths quickly 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). The MRCA's descendants are then calculated and compared to the the two nodes'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 a given node in a preordered list of edges. The direct descendants (or children) of the given node are then 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, but 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 implementaion builds on top of original R version by parallelizing 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

OpenMP allows parallelism to be added without changing a significant amount of existing code. And that's exactly what we did implement to implement an OpenMP version of the original shortestPath program. We decided to parallelize the two serial C functions called and implied by the shortestPath function in treewalk.R. Those two C functions are ancestors() in ancestors.c and descendants() in descendants.c. Because serial for loops comprised the bulk of the ancestor and descendant C functions, the OpenMP implementation is embarrassingly parallel. The functions are also small so there isn't a need for explicit barriers.

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, 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 unprocessed results from using R's system.time() on the RSnow and OMP shortestPath functions on testXXX.tre test cases, where XXX denotes the numbers of leaves the tree has. The serial label denotes the original R implementation. We did not manage to get the CUDA implemetation to correctly interface with R.

```
#msdir/set1/test10.tre
#serial
c(0.015, 0.011, 0.012, 0.012, 0.012, 0.011, 0.012, 0.008, 0.011, 0.012)
#RSnow
c(0.068, 0.017, 0.018, 0.017, 0.017, 0.017, 0.017, 0.018, 0.017, 0.019)
#OMP
c(0.016, 0.011, 0.012, 0.011, 0.011, 0.012, 0.011, 0.009, 0.012, 0.011)

#msdir/set1/test100.tre
#serial
c(0.027, 0.030, 0.022, 0.020, 0.020, 0.021, 0.021, 0.020, 0.021, 0.021)
#RSnow
c(0.155, 0.132, 0.117, 0.118, 0.130, 0.125, 0.136, 0.111, 0.115, 0.114)
#OMP
c(0.046, 0.025, 0.021, 0.021, 0.021, 0.020, 0.021, 0.021, 0.020, 0.021)

#msdir/set1/test1000.tre
#serial
#c(0.131, 0.128, 0.125, 0.122, 0.124, 0.136, 0.129, 0.123, 0.123, 0.126)
#RSnow
c(16.454, 17.481, 16.778, 17,686, 12.808, 16.032, 15.914, 16.816, 13.582, 14.389)
#OMP
#c(0.132, 0.130, 0.151, 0.124, 0.133, 0.134, 0.125, 0.124, 0.123, 0.126)

#msdir/set1/test10000.tre
#serial
c(1.819, 1.851, 1.850, 1.789, 1.754, 1.840, 1.809, 1.699, 1.735, 1.787)
#RSnow
c(0,0,0,0,0,0,0,0,0,0) #program crashed on this case
#OMP
c(1.783, 1.841, 1.820, 1.751, 1.805, 1.830, 1.796, 1.705, 1.733, 1.784)
```

4 Discussion

4.1 RSnow

The RSnow version was tested with four localhost worker nodes. The large majority of the runs were slower than the serial version. Looking back at our appraoch to the problem, it was quite demanding for resources, specifically memory hungry. It was not a good idea for have four localhost nodes tackle the issue.

4.2 OpenMP

The OpenMP version works marginally faster than the original code on large phylogeneic trees (approximately 10000 leaves). With small trees, serial "for" loops seem to be efficient in finding ancestor and descendant nodes.

4.3 CUDA

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

In the attempt to parallelize the shortest path function in the phylobase package, the RSnow implementation illustrated the efficiency of the original code and its serial search algorithm. The OpenMP implementation showed the utility of multithreading for large data. The CUDA implementation could not be timed, because of R/C interafacing issues.

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 also 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

The RSnow descendant functions calculates decesdant in reverse by looking at ancestors.

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
}
```

```
50
51 ## get descendants with RSnow
52 RSnowdescendants <- function (phy, node, type=c("tips","children","all"),cls) {
53     type <- match.arg(type)
54
55     ## look up nodes, warning about and excluding invalid nodes
56     oNode <- node
57     node <- getNode(phy, node, missing="warn")
58     isValid <- !is.na(node)
59     node <- as.integer(node[isValid])
60
61     if (type == "children") {
62         res <- lapply(node, function(x) children(phy, x))
63         ## if just a single node, return as a single vector
64         if (length(res)==1) res <- res[[1]]
65     } else {
66         ## edge matrix must be in preorder for the C function!
67         #if (phy@order=="preorder") {
68             edge <- phy@edge
69         #} else {
70         #    edge <- reorder(phy, order="postorder")@edge
71         #}
72         ## extract edge columns
73         ancestor <- as.integer(edge[, 1])
74         descendant <- as.integer(edge[, 2])
75
76         ## return indicator matrix of ALL descendants (including self)
77         #isDes <- .Call("descendants", node, ancestor, descendant)
78         clusterExport(cls,c("node", "ancestor", "descendant","setmyid","SNOW"),
               envir=environment())
79         dexgrps <- splitIndices(length(ancestor),length(cls))
80         rootdex <- which(phy@edge[,1] == 0)
81         clusterApply(cls,1:length(cls),setmyid)
82         newisDes <- clusterApply(cls,dexgrps,SNOW,length(ancestor),rootdex,      "
               descendants")
83         isDes <- (matrix(Reduce('+',newisDes),nrow=length(ancestor),ncol=1))
84         storage.mode(isDes) <- "logical"
85         ## for internal nodes only, drop self (not sure why this rule?)
86         int.node <- intersect(node, nodeId(phy, "internal"))
87         isDes[cbind(match(int.node, descendant),
88             match(int.node, node))] <- FALSE
89
90         ## if only tips desired, drop internal nodes
91         if (type=="tips") {
92             isDes[descendant %in% nodeId(phy, "internal"),] <- FALSE
93         }
94         ## res <- lapply(seq_along(node), function(n) getNode(phy,
95         ##     descendant[isDes[,n]]))
96         res <- getNode(phy, descendant[isDes[, seq_along(node)]])
97     }
98     ## names(res) <- as.character(oNode[isValid])
99
100    res
101 }
102
103 ################
104 # shortestPath
105 ################
106
```

```r
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

As stated approve in the approach section, very few lines were changed. OMP for's were added everywhere. The R code was modified so it would not interfere with the original code syntactically.

```
1
2 #include <R.h>
3 #include <Rinternals.h>
4 #include "omp.h"
5 int num_threads=8;
6 void omp_set_num_threads(int num_threads);
7
8 SEXP OMPancestors(SEXP nod, SEXP anc, SEXP des) {
9
10     int numEdges = length(anc);
11     int numNodes = length(nod);
12
13     int* nodes = INTEGER(nod);
14     int* ancestor = INTEGER(anc);
15     int* descendant = INTEGER(des);
16
17     int parent = 0;
18     SEXP isAncestor;
19
```

```
20      PROTECT(isAncestor = allocMatrix(INTSXP, numEdges, numNodes));
21  #pragma omp parallel for collapse(2)
22      for (int n=0; n<numNodes; n++) {
23      //#pragma omp parallel for
24          for (int i=0; i<numEdges; i++) {
25              if (nodes[n]==descendant[i]) {
26                  INTEGER(isAncestor)[i + n*numEdges] = 1;
27              } else {
28                  INTEGER(isAncestor)[i + n*numEdges] = 0;
29              }
30          }
31      }
32
33  #pragma omp parallel for collapse(2)
34      for (int n=0; n<numNodes; n++) {
35      //#pragma omp parallel for
36          for (int i=0; i<numEdges; i++) {
37              if (INTEGER(isAncestor)[i + n*numEdges]==1) {
38                  parent = ancestor[i];
39                  for (int j=i+1; j<numEdges; j++) {
40                      if (descendant[j]==parent) {
41                          INTEGER(isAncestor)[j + n*numEdges]=1;
42                      }
43                  }
44              }
45          }
46      }
47
48      UNPROTECT(1);
49      return isAncestor;
50  }
51
52  SEXP OMPdescendants(SEXP nod, SEXP anc, SEXP des) {
53
54      int numEdges = length(anc);
55      int numNodes = length(nod);
56
57      int* nodes = INTEGER(nod);
58      int* ancestor = INTEGER(anc);
59      int* descendant = INTEGER(des);
60
61      int child = 0;
62      SEXP isDescendant;
63
64      PROTECT(isDescendant = allocMatrix(INTSXP, numEdges, numNodes));
65  #pragma omp parallel for collapse(2)
66     for (int n=0; n<numNodes; n++) {
67          for (int i=0; i<numEdges; i++) {
68              if (nodes[n]==descendant[i]) {
69                  INTEGER(isDescendant)[i + n*numEdges] = 1;
70              } else {
71                  INTEGER(isDescendant)[i + n*numEdges] = 0;
72              }
73          }
74      }
75
76  #pragma omp parallel for collapse(2)
77      for (int n=0; n<numNodes; n++) {
78          for (int i=0; i<numEdges; i++) {
79              if (INTEGER(isDescendant)[i + n*numEdges]==1) {
80                  child = descendant[i];
81                  for (int j=i+1; j<numEdges; j++) {
82                      if (ancestor[j]==child) {
83                          INTEGER(isDescendant)[j + n*numEdges] = 1;
84                      }
85                  }
86              }
87          }
```

```
    }
    UNPROTECT(1);
    return isDescendant;
}
```

```
## get descendants [recursively]
OMPdescendants <- function (phy, node, type=c("tips","children","all")) {
    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="preorder")@edge
        }
        ## extract edge columns
        ancestor <- as.integer(edge[, 1])
        descendant <- as.integer(edge[, 2])

        ## TODO: REPLACE C call with OMP inplementation of descendants
        ## return indicator matrix of ALL descendants (including self)
        isDes <- .Call("OMPdescendants", node, ancestor, descendant)
        storage.mod## get descendants [recursively]
OMPdescendants <- function (phy, node, type=c("tips","children","all")) {
    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="preorder")@edge
        }
        ## extract edge columns
        ancestor <- as.integer(edge[, 1])
        descendant <- as.integer(edge[, 2])

        ## TODO: REPLACE C call with OMP inplementation of descendants
```

```
        ## return indicator matrix of ALL descendants (including self)
        isDes <- .Call("OMPdescendants", node, ancestor, descendant)
        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 <- getNode(phy, descendant[isDes[, seq_along(node)]])
    }

    res
}#OMPdescendants

## get ancestors (all nodes)
OMPancestors <- function (phy, node, type=c("all","parent","ALL")) {
    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 (length(node) == 0) {
      return(NA)
    }

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

        ## TODO: REPLACE C call with OMP inplementation of ancestors
        ## return indicator matrix of ALL ancestors (including self)
        isAnc <- .Call("OMPancestors", node, ancestor, descendant)
        storage.mode(isAnc) <- "logical"

        ## drop self if needed
        if (type=="all") {
            isAnc[cbind(match(node, descendant), seq_along(node))] <- FALSE
        }
        res <- lapply(seq_along(node), function(n) getNode(phy,
            descendant[isAnc[,n]]))
    }
```

```
114     names(res) <- as.character(oNode[isValid])
115
116     ## if just a single node, return as a single vector
117     if (length(res)==1) res <- res[[1]]
118     res
119 }#OMPancestors
120
121 OMPMRCA <- function(phy, ...) {
122     nodes <- list(...)
123     ## if length==1 and first element is a vector,
124     ##   use it as the list
125     if (length(nodes)==1 && length(nodes[[1]])>1) {
126         nodes <- as.list(nodes[[1]])
127     }
128
129     ## Correct behavior when the root is part of the nodes
130     testNodes <- lapply(nodes, getNode, x=phy)
131     ## BMB: why lapply, not sapply?
132     lNodes <- unlist(testNodes)
133     if (any(is.na(lNodes)))
134       stop("nodes not found in tree: ",paste(names(lNodes)[is.na(lNodes)],
135                                              collapse=", "))
136     uniqueNodes <- unique(testNodes)
137     root <- nTips(phy)+1
138     ## Handles case where root is a node of interest, return root
139     if(root %in% uniqueNodes) {
140         res <- getNode(phy, root)
141         return(res)
142     }
143     ## Correct behavior in case of MRCA of identical taxa
144     if(length(uniqueNodes) == 1) {
145         res <- uniqueNodes[[1]]
146         return(res)
147     }
148     else { ## else length(uniqueNodes > 1)
149         ancests <- lapply(nodes, OMPancestors, phy=phy, type="ALL")
150         res <- getNode(phy, max(Reduce(intersect, ancests)))
151         return(res)
152     }
153 }#OMPMRCA
154
155
156 ################
157 # shortestPath
158 ################
159 OMPshortestPath <- function(phy, node1, node2){
160   ## conversion from phylo, phylo4 and phylo4d
161   if (class(phy) == "phylo4d") {
162     x <- extractTree(phy)
163   }
164   else if (class(phy) != "phylo4"){
165     x <- as(phy, "phylo4")
166   }
167
168     ## some checks
169     ## if (is.character(checkval <- checkPhylo4(x))) stop(checkval) # no need
170     t1 <- getNode(x, node1)
171     t2 <- getNode(x, node2)
172     if(any(is.na(c(t1,t2)))) stop("wrong node specified")
```

```
173     if(t1==t2) return(NULL)
174
175     ## main computations
176     comAnc <- OMPMRCA(x, t1, t2) # common ancestor
177     desComAnc <- OMPdescendants(x, comAnc, type="all")
178     ancT1 <- OMPancestors(x, t1, type="all")
179     path1 <- intersect(desComAnc, ancT1) # path: common anc -> t1
180
181     ancT2 <- OMPancestors(x, t2, type="all")
182     path2 <- intersect(desComAnc, ancT2) # path: common anc -> t2
183
184     res <- union(path1, path2) # union of the path
185     ## add the common ancestor if it differs from t1 or t2
186     if(!comAnc %in% c(t1,t2)){
187         res <- c(comAnc,res)
188     }
189
190     res <- getNode(x, res)
191
192     return(res)
193 } # end shortestPathe(isDes) <- "logical"
194
195         ## for internal nodes only, drop self (not sure why this rule?)
196         int.node <- intersect(node, nodeId(phy, "internal"))
197         isDes[cbind(match(int.node, descendant),
198             match(int.node, node))] <- FALSE
199
200         ## if only tips desired, drop internal nodes
201         if (type=="tips") {
202             isDes[descendant %in% nodeId(phy, "internal"),] <- FALSE
203         }
204
205         res <- getNode(phy, descendant[isDes[, seq_along(node)]])
206     }
207
208     res
209 }#OMPdescendants
210
211 ## get ancestors (all nodes)
212 OMPancestors <- function (phy, node, type=c("all","parent","ALL")) {
213     type <- match.arg(type)
214
215     ## look up nodes, warning about and excluding invalid nodes
216     oNode <- node
217     node <- getNode(phy, node, missing="warn")
218     isValid <- !is.na(node)
219     node <- as.integer(node[isValid])
220
221     if (length(node) == 0) {
222       return(NA)
223     }
224
225     if (type == "parent") {
226         res <- lapply(node, function(x) ancestor(phy, x))
227     } else {
228         ## edge matrix must be in postorder for the C function!
229         if (phy@order=="postorder") {
230             edge <- phy@edge
231         } else {
```

```
232             edge <- reorder(phy, order="postorder")@edge
233         }
234         ## extract edge columns
235         ancestor <- as.integer(edge[, 1])
236         descendant <- as.integer(edge[, 2])
237 
238         ## TODO: REPLACE C call with OMP inplementation of ancestors
239         ## return indicator matrix of ALL ancestors (including self)
240         isAnc <- .Call("OMPancestors", node, ancestor, descendant)
241         storage.mode(isAnc) <- "logical"
242 
243         ## drop self if needed
244         if (type=="all") {
245             isAnc[cbind(match(node, descendant), seq_along(node))] <- FALSE
246         }
247         res <- lapply(seq_along(node), function(n) getNode(phy,
248             descendant[isAnc[,n]]))
249     }
250     names(res) <- as.character(oNode[isValid])
251 
252     ## if just a single node, return as a single vector
253     if (length(res)==1) res <- res[[1]]
254     res
255 }#OMPancestors
256 
257 OMPMRCA <- function(phy, ...) {
258     nodes <- list(...)
259     ## if length==1 and first element is a vector,
260     ##   use it as the list
261     if (length(nodes)==1 && length(nodes[[1]])>1) {
262         nodes <- as.list(nodes[[1]])
263     }
264 
265     ## Correct behavior when the root is part of the nodes
266     testNodes <- lapply(nodes, getNode, x=phy)
267     ## BMB: why lapply, not sapply?
268     lNodes <- unlist(testNodes)
269     if (any(is.na(lNodes)))
270       stop("nodes not found in tree: ",paste(names(lNodes)[is.na(lNodes)],
271                                              collapse=", "))
272     uniqueNodes <- unique(testNodes)
273     root <- nTips(phy)+1
274     ## Handles case where root is a node of interest, return root
275     if(root %in% uniqueNodes) {
276         res <- getNode(phy, root)
277         return(res)
278     }
279     ## Correct behavior in case of MRCA of identical taxa
280     if(length(uniqueNodes) == 1) {
281         res <- uniqueNodes[[1]]
282         return(res)
283     }
284     else { ## else length(uniqueNodes > 1)
285         ancests <- lapply(nodes, OMPancestors, phy=phy, type="ALL")
286         res <- getNode(phy, max(Reduce(intersect, ancests)))
287         return(res)
288     }
289 }#OMPMRCA
290 
```

```
291
292 ################
293 # shortestPath
294 ################
295 OMPshortestPath <- function(phy, node1, node2){
296   ## conversion from phylo, phylo4 and phylo4d
297   if (class(phy) == "phylo4d") {
298     x <- extractTree(phy)
299   }
300   else if (class(phy) != "phylo4"){
301     x <- as(phy, "phylo4")
302   }
303
304     ## some checks
305     ## if (is.character(checkval <- checkPhylo4(x))) stop(checkval) # no need
306     t1 <- getNode(x, node1)
307     t2 <- getNode(x, node2)
308     if(any(is.na(c(t1,t2)))) stop("wrong node specified")
309     if(t1==t2) return(NULL)
310
311     ## main computations
312     comAnc <- OMPMRCA(x, t1, t2) # common ancestor
313     desComAnc <- OMPdescendants(x, comAnc, type="all")
314     ancT1 <- OMPancestors(x, t1, type="all")
315     path1 <- intersect(desComAnc, ancT1) # path: common anc -> t1
316
317     ancT2 <- OMPancestors(x, t2, type="all")
318     path2 <- intersect(desComAnc, ancT2) # path: common anc -> t2
319
320     res <- union(path1, path2) # union of the path
321     ## add the common ancestor if it differs from t1 or t2
322     if(!comAnc %in% c(t1,t2)){
323         res <- c(comAnc,res)
324     }
325
326     res <- getNode(x, res)
327
328     return(res)
329 } # end shortestPath
```

A.3 CUDA Code

Here are some highlights of the coming CUDA code.

Line 7: Node struct contains information about tree's nodes (node id, ancestor id, node label)

Line 13: setNode function translates R tree nodes to node structs

Line 22: kernel function GPU function that computes ancestors of all nodes in the tree

Line 67: shortestPath function takes an array of nodes, number of nodes given, and two character array labels and finds the shortest path between the given nodes. An error message is printed if the given nodes are invalid input (ex: they are a parent-child)

Line 196: main function driver function, reads in input and calls the shortestPath function

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

//to compile: 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
```

```
64 }//kernel
65
66
67 void shortestPath(node * phy, int numNodes, char * label1, char * label2)
68 {
69     node * deviceArray;
70     int * deviceID1;
71     int * deviceID2;
72     int * ancestorID1 = new int[numNodes]; //initialize max size to number of nodes
73     int * ancestorID2 = new int[numNodes];
74     float blockSize = 1024; //num threads per block
75
76     //check if invalid query
77     node temp1, temp2;
78     for (int i=0; i<numNodes; i++)
79     {
80         ancestorID1[i] = 0;
81         ancestorID2[i] = 0;
82         if (strcmp(label1, phy[i].label) == 0)
83             temp1 = phy[i];
84         else if (strcmp(label2, phy[i].label) == 0)
85             temp2 = phy[i];
86     }//for
87
88     if ((temp1.ancestor == temp2.nodeID) || (temp2.ancestor == temp1.nodeID))
89     {
90         printf("named integer(0)\n");
91         return;
92     }//if
93
94     //allocate device memory
95     cudaMalloc(&deviceArray, sizeof(node) * numNodes);
96     cudaMalloc(&deviceID1, sizeof(int) * numNodes);
97     cudaMalloc(&deviceID2, sizeof(int) * numNodes);
98     cudaMemcpy(deviceArray, phy, sizeof(node) * numNodes, cudaMemcpyHostToDevice);
99     cudaMemcpy(deviceID1, ancestorID1, sizeof(int) * numNodes, cudaMemcpyHostToDevice);
100    cudaMemcpy(deviceID2, ancestorID2, sizeof(int) * numNodes, cudaMemcpyHostToDevice);
101
102    dim3 dimBlock(blockSize);
103    dim3 dimGrid(ceil(numNodes/blockSize));
104
105    //compute ancestors
106    kernel <<< dimGrid, dimBlock >>> (deviceArray, numNodes, temp1.nodeID, temp2.nodeID, ↩
           deviceID1, deviceID2);
107    cudaMemcpy(ancestorID1, deviceID1, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
108    cudaMemcpy(ancestorID2, deviceID2, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
109    cudaFree(deviceArray);
110    cudaFree(deviceID1);
111    cudaFree(deviceID2);
112
113    //find shortest path
114    int * path = new int[numNodes];
115    int currentPath = ancestorID1[0];
116    int pathIndex = 0;
117    bool isLCAPath = false;
118    //check if path converges at LCA
119    for (int i=0; i<numNodes; i++)
120    {
121        path[i] = 0;
122        if (temp1.nodeID == ancestorID2[i])
123        {
124            for (int j=0; j<i; j++)
125                path[j] = ancestorID2[j];
126            isLCAPath = true;
127            break;
128        }//if
129        else if (temp2.nodeID == ancestorID1[i])
130        {
```

```
131             for (int j=0; j<i; j++)
132                 path[j] = ancestorID1[j];
133             isLCAPath = true;
134             break;
135         }//else if
136     }//for
137 
138     //if one node is the ancestor of another
139     if (!isLCAPath)
140     {
141         for(int i=0; i<numNodes; i++)
142         {
143             for (int j=0; j<numNodes; j++)
144             {
145                 if (currentPath == ancestorID2[j])
146                     break;
147                 if ((ancestorID2[j] == 0) || (j == numNodes-1))
148                 {
149                     path[pathIndex++] = ancestorID1[i];
150                     currentPath = ancestorID1[i];
151                     break;
152                 }//if
153             }//for
154         }//for
155 
156         if (pathIndex == 0)
157             path[pathIndex++] = currentPath;
158 
159         for (int i=0; i<numNodes; i++)
160         {
161             if (ancestorID2[i] == currentPath)
162                 break;
163             path[pathIndex++] = ancestorID2[i];
164         }//for
165     }//if
166 
167     for (int i=0; i<numNodes; i++)
168     {
169         if (path[i] == 0)
170             break;
171         for (int j=0; j<numNodes; j++)
172         {
173             if (path[i] == phy[j].nodeID)
174             {
175                 printf("%s ", phy[j].label);
176                 break;
177             }//if
178         }//for
179     }//for
180 
181     printf("\n");
182     for (int i=0; i<numNodes; i++)
183     {
184         if (path[i] == 0)
185             break;
186         printf("%d ", path[i]);
187     }//for
188     printf("\n");
189 
190     delete [] ancestorID1;
191     delete [] ancestorID2;
192     delete [] path;
193 }//shortestPath
194 
195 
196 int main()
197 {
198     int numNodes = 27;
```

```
199     node * phy = new node[numNodes];
200     FILE * infile = fopen("geospiza", "r");
201 
202     int nodeID, ancestor;
203     char label[20];
204     for (int i=0; i<numNodes; i++)
205     {
206         fscanf(infile, "%d", &nodeID);
207         fscanf(infile, "%d", &ancestor);
208         fscanf(infile, "%s", &label);
209         setNode(phy[i], numNodes, nodeID, ancestor, label);
210     }//for
211     fclose(infile);
212 
213 //test shortest path
214     shortestPath(phy, numNodes, "fusca", "fortis");
215 
216     delete [] phy;
217     return 0;
218 }//main
```

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] F. Michonneau, *http://cran.r-project.org/web/packages/phylobase/phylobase.pdf* February 20, 2015.