

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

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

Abstract

This report attempts parallelize CRAN's phylobase package's shortestPath function in RSnow, OpenMP and CUDA. The function takes in a phylogenetic tree, two nodes in the tree and produces the shortest path of nodes inbetween them. The RSnow implementation built on top of the existing code and parallelized 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, bioinformatics 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 comparative data. (CITE) We will be focusing our efforts in making the "treewalk" utility functions, such as finding descendants/ancestors and shortest paths, 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 between 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):

```

1 descendants(tree, given_node){
2   #let x be the edges of tree listed in PREORDER,
3   #with the older node occupying the first column
4   # C function call
5   isDescendant <- Cdescendants(x[,1], x[,2], given_node)
6   retval <- getNode(tree, isDescendant)
7 }
8
9 ancestors(tree, node1){
10  #let x be the edges of tree listed in POSTORDER,
11  #with the older node occupying the first column
12  # C function call
13  isAncestor <- Cancestors(x[,1], x[,2], given_node)
14  retval <- getNode(tree, isAncestor)
15 }
16
17 MRCA(tree, node1, node2 ... noden){
18   nodes <- unique(node1, node2, ..., noden)
19   ancests <- lapply(nodes, ancestors, phy=phy, type="ALL")
20   retval <- getNode(phy, max(Reduce(intersect, ancests)))
21 }
22 shortestPath(tree, node1, node2){
23   t1 <- getNode(tree, node1)
24   t2 <- getNode(tree, node2)
25
26   # most recent common ancestor
27   comAnc <- MRCA(tree, t1, t2)
28   desComAnc <- descendants(tree, comAnc)
29
30   # path: common ancestor to t1
31   ancT1 <- ancestors(x, t1)
32   path1 <- intersect(desComAnc, ancT1)
33
34   # path: common ancestor to t2
35   ancT2 <- ancestors(x, t2)
36   path2 <- intersect(desComAnc, ancT2)
37
38   # union of the path above paths
39   retval <- union(path1, path2)
40 }

```

2.2 RSnw

The RSnw implementation 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 efficient serial approach for calculating a given node's ancestors.

Pseudocode: (see code in Appendix A.1)

```

1 descendants(tree, given_node){
2   #let x be the list of all nodes except the given_node
3   for i from 1 to height of tree
4     if x == given_node -> append node to retval

```

```

5         update x with its corresponding ancestor
6     return retval
7 }

```

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 GPU's 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 pages). For the current day, the sheer amount of data that needs to be 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 tutorials were an essential part of our learning. We would like to thank the teaching assistant Shengren Li for offering invaluable advice and feedback on our codes (especially our CUDA) throughout the quarter.

7 Appendix

A Codes

INSERT ALL CODES HERE ALONG WITH A PARAGRAPH EXPLAINING IT

A.1 RSnow Code

```

1 SNOW <- function(x, size, root, type=c("descendants")){
2   ans <- rep(0, size)
3   mystart <- (myid-1)*length(x)+1
4   myend <- myid*length(x)
5
6   type <- match.arg(type)
7   if (type == "descendants"){
8     v1 <- descendant
9     v2 <- ancestor
10    #initialization
11    temp <- v1[mystart:myend]
12
13    #second and beyond iteration
14    for (j in 1:size){
15      if (node %in% temp){
16        setthese <- which(temp == node) + mystart-1
17        ans[setthese] <- 1
18      }
19      blah <- rep(-1, length(temp))
20      for (i in (1:length(temp))){
21        matched_pos <- which(v1 == temp[i])
22        if (length(matched_pos) != 0){
23          blah[which(temp == temp[i])] <- matched_pos
24        }
25        else{#matched_pos == 0
26          ## R is 1 INDEXED!
27          if (type == "descendants"){
28            blah[i] <- 1
29          }
30        }
31      }#for i
32      #go to your parents set
33      difference <- length(temp) - length(v2[blah])
34      temp <- v2[blah]
35      if (difference > 0){
36        temp <- c(rep(0, difference), temp)
37      }
38      if (node %in% temp){
39        setthese <- which(temp == node) + mystart-1
40        ans[setthese] <- 1
41      }
42    }#j loop
43  }#new endif for type==descendants
44  return(ans)
45 }# end SNOW
46
47 setmyid <- function(i){
48   myid <- i
49 }
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"),
79                   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,
83                             "descendants")
84     isDes <- (matrix(Reduce('+', newisDes), nrow=length(ancestor), ncol=1))
85     storage.mode(isDes) <- "logical"
86     ## for internal nodes only, drop self (not sure why this rule?)
87     int.node <- intersect(node, nodeId(phy, "internal"))
88     isDes[cbind(match(int.node, descendant),
89                 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
107 RSnowshortestPath <- function(phy, node1, node2, cls){
108
109   ## conversion from phylo, phylo4 and phylo4d
110   if (class(phy) == "phylo4d") {
111     x <- extractTree(phy)
112   }
113   else if (class(phy) != "phylo4"){
114     x <- as(phy, "phylo4")

```

```

115     }
116     ## some checks
117     t1 <- getNode(x, node1)
118     t2 <- getNode(x, node2)
119     if(any(is.na(c(t1,t2)))) stop("wrong node specified")
120     if(t1==t2) return(NULL)
121
122     ## main computations
123     comAnc <- MRCA(x, t1, t2) # common ancestor
124     desComAnc <- RSnowdescendants(x, comAnc, type="all",cls)
125     ancT1 <- ancestors(x, t1, type="all")
126     path1 <- intersect(desComAnc, ancT1) # path: common anc -> t1
127
128     ancT2 <- ancestors(x, t2, type="all")
129     path2 <- intersect(desComAnc, ancT2) # path: common anc -> t2
130
131     res <- union(path1, path2) # union of the path
132     ## add the common ancestor if it differs from t1 or t2
133     if(!comAnc %in% c(t1,t2)){
134         res <- c(comAnc,res)
135     }
136
137     res <- getNode(x, res)
138
139     return(res)
140 } # end shortestPath

```

A.2 OpenMP Code

```

1 ALICIA CODE GOES HERE

```

A.3 CUDA Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4
5 //nvcc shortestPath.cu
6
7 struct node
8 {
9     int nodeID, ancestor;
10    char label[20]; //max size of label is 20
11 }; //node
12
13 void setNode(node &phy, int numNodes, int id, int aID, char * label)
14 {
15     phy.nodeID = id;
16     phy.ancestor = aID;
17     memset(phy.label, '\0', sizeof(label));
18     strcpy(phy.label, label);
19 } //setNode
20
21 __global__ void kernel(node * array, int numNodes, int id1, int id2,
22                        int * ancestorID1, int * ancestorID2)
23 {

```

```

24 int idx = blockIdx.x * blockDim.x + threadIdx.x;
25 if (idx < numNodes)
26 {
27     if (array[idx].nodeID == id1) //if found target node
28     {
29         int ancestorIndex = 0;
30         node temp = array[idx]; //start from current node
31         while (temp.ancestor != 0)
32         {
33             ancestorID1[ancestorIndex++] = temp.ancestor; //add all ancestors
34             for (int i=0; i<numNodes; i++)
35             {
36                 if (array[i].nodeID == temp.ancestor)
37                 {
38                     temp = array[i];
39                     break;
40                 } //if
41             } //for
42         } //while
43     } //if
44     else if (array[idx].nodeID == id2) //if found target node
45     {
46         int ancestorIndex = 0;
47         node temp = array[idx]; //start from current node
48         while (temp.ancestor != 0)
49         {
50             ancestorID2[ancestorIndex++] = temp.ancestor; //add all ancestors
51             for (int i=0; i<numNodes; i++)
52             {
53                 if (array[i].nodeID == temp.ancestor)
54                 {
55                     temp = array[i];
56                     break;
57                 } //if
58             } //for
59         } //while
60     } //if
61 }
62 } //if
63 } //kernel
64
65 void shortestPath(node * phy, int numNodes, char * label1, char * label2)
66 {
67     node * deviceArray;
68     int * deviceID1;
69     int * deviceID2;
70     int * ancestorID1 = new int[numNodes]; //initialize max size to number of nodes
71     int * ancestorID2 = new int[numNodes];
72     float blockSize = 1024; //num threads per block
73
74     //check if invalid query
75     node temp1, temp2;
76     for (int i=0; i<numNodes; i++)
77     {
78         ancestorID1[i] = 0;
79         ancestorID2[i] = 0;
80         if (strcmp(label1, phy[i].label) == 0)
81             temp1 = phy[i];
82         else if (strcmp(label2, phy[i].label) == 0)
83             temp2 = phy[i];
84     } //for
85
86     if ((temp1.ancestor == temp2.nodeID) || (temp2.ancestor == temp1.nodeID))
87     {
88         printf("named integer(0)\n");
89         return;
90     } //if
91

```



```

92 //allocate device memory
93 cudaMalloc(&deviceArray, sizeof(node) * numNodes);
94 cudaMalloc(&deviceID1, sizeof(int) * numNodes);
95 cudaMalloc(&deviceID2, sizeof(int) * numNodes);
96 cudaMemcpy(deviceArray, phy, sizeof(node) * numNodes, cudaMemcpyHostToDevice);
97 cudaMemcpy(deviceID1, ancestorID1, sizeof(int) * numNodes, cudaMemcpyHostToDevice);
98 cudaMemcpy(deviceID2, ancestorID2, sizeof(int) * numNodes, cudaMemcpyHostToDevice);
99
100 dim3 dimBlock(blockSize);
101 dim3 dimGrid(ceil(numNodes/blockSize));
102
103 //compute ancestors
104 kernel <<< dimGrid, dimBlock >>> (deviceArray, numNodes, temp1.nodeID, temp2.nodeID, ←
    deviceID1, deviceID2);
105 cudaMemcpy(ancestorID1, deviceID1, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
106 cudaMemcpy(ancestorID2, deviceID2, sizeof(int) * numNodes, cudaMemcpyDeviceToHost);
107 cudaFree(deviceArray);
108 cudaFree(deviceID1);
109 cudaFree(deviceID2);
110
111 //find shortest path
112 int * path = new int[numNodes];
113 int currentPath = ancestorID1[0];
114 int pathIndex = 0;
115 bool isLCAPath = false;
116 //check if path converges at LCA
117 for (int i=0; i<numNodes; i++)
118 {
119     path[i] = 0;
120     if (temp1.nodeID == ancestorID2[i])
121     {
122         for (int j=0; j<i; j++)
123             path[j] = ancestorID2[j];
124         isLCAPath = true;
125         break;
126     }//if
127     else if (temp2.nodeID == ancestorID1[i])
128     {
129         for (int j=0; j<i; j++)
130             path[j] = ancestorID1[j];
131         isLCAPath = true;
132         break;
133     }//else if
134 }//for
135
136 //if one node is the ancestor of another
137 if (!isLCAPath)
138 {
139     for(int i=0; i<numNodes; i++)
140     {
141         for (int j=0; j<numNodes; j++)
142         {
143             if (currentPath == ancestorID2[j])
144                 break;
145             if ((ancestorID2[j] == 0) || (j == numNodes-1))
146             {
147                 path[pathIndex++] = ancestorID1[i];
148                 currentPath = ancestorID1[i];
149                 break;
150             }//if
151         }//for
152     }//for
153
154     if (pathIndex == 0)
155         path[pathIndex++] = currentPath;
156
157     for (int i=0; i<numNodes; i++)
158     {

```

```

159         if (ancestorID2[i] == currentPath)
160             break;
161         path[pathIndex++] = ancestorID2[i];
162     } // for
163 } // if
164 for (int i=0; i<numNodes; i++)
165 {
166     if (path[i] == 0)
167         break;
168     for (int j=0; j<numNodes; j++)
169     {
170         if (path[i] == phy[j].nodeID)
171         {
172             printf("%s ", phy[j].label);
173             break;
174         } // if
175     } // for
176 } // for
177
178 printf("\n");
179 for (int i=0; i<numNodes; i++)
180 {
181     if (path[i] == 0)
182         break;
183     printf("%d ", path[i]);
184 } // for
185 printf("\n");
186
187 delete [] ancestorID1;
188 delete [] ancestorID2;
189 delete [] path;
190 } // 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 L^AT_EX.

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*, available 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