ECS 158 Final Report:

combn() in Snow, OpenMP, and Thrust

Stefan Peterson
Bijan Agahi
Arjun Bharadwaj

Abstract

Parallel architectures are becoming increasingly prevalent in the modern world and as such have spawned multiple frameworks to assist with parallel programming, but these frameworks vary greatly in their implementation and performance. This paper takes a close look at three frameworks: R-Snow, OpenMP and Thrust. We took on the task of re-writing the combn() function found in R using each of these frameworks. We measure run times of these tests at different sizes and with varying inputs and levels of difficulty. In the end, we point out the strengths and weaknesses of each language as well as note on features which we, as programmers, find particularly useful or troublesome.

Contents

1 Introduction

With the relatively recent introduction of widely available multi-processor CPUs in household computers, parallel programming has become a hot topic in the world of computer science. In particular, the graphics and games industry has driven the multithreaded programming style to the forefront of everyone's attention. Through the process of splitting up tasks and distributing those tasks to individual cores on the CPU or GPU, parallel processing in most cases can provide vast improvements in both performance and efficiency. OpenMP, R-Snow, and Thrust are three such parallel languages, and this paper aims to compare and contrast the strengths, weaknesses, and performance of each language.

1.1 Setup for Benchmarks

All tests for the purposes of this paper were run on Tetraat University of California, Davis. This computer consists of a 64-bit 8-core processor with 16 GB or RAM and an Intel Core i7-2600K CPU with a clock speed of 3.4 GHz and a cache size of 8192 KB. The machine was running release 19 of the linux distribution Fedora.

2 combn() in OpenMP

2.1 Strategy

For the OpenMP version of the combinations function, we decided to parallelize by dedicating each thread to a subsection of the problem. Each subsection corresponds to the group of combinations which begin with the same value. In [Appendix Graphic 2] we see all the possible combinations for a $\binom{5}{3}$. Those can be clumped into the three groups each beginning with 1, 2, or 3. `Thread 1` will handle the computation of all combinations beginning with 1, `thread 2` will handle those beginning with 2 and `thread 3` will handle those beginning with 3. Obviously this is not load balanced for small cases since `thread 1` is doing significantly more work than the rest of the threads. When we get to larger problems, we use the built in dynamic scheduling of OpenMP to make an attempt at load balancing. If we had only 2 threads running on the above problem, `thread 1` would still do its chunk of the computation, then `thread 2` would do its and when it finishes (which should be before `thread 1`) it grabs the next process to be executed.

2.2 Implementation

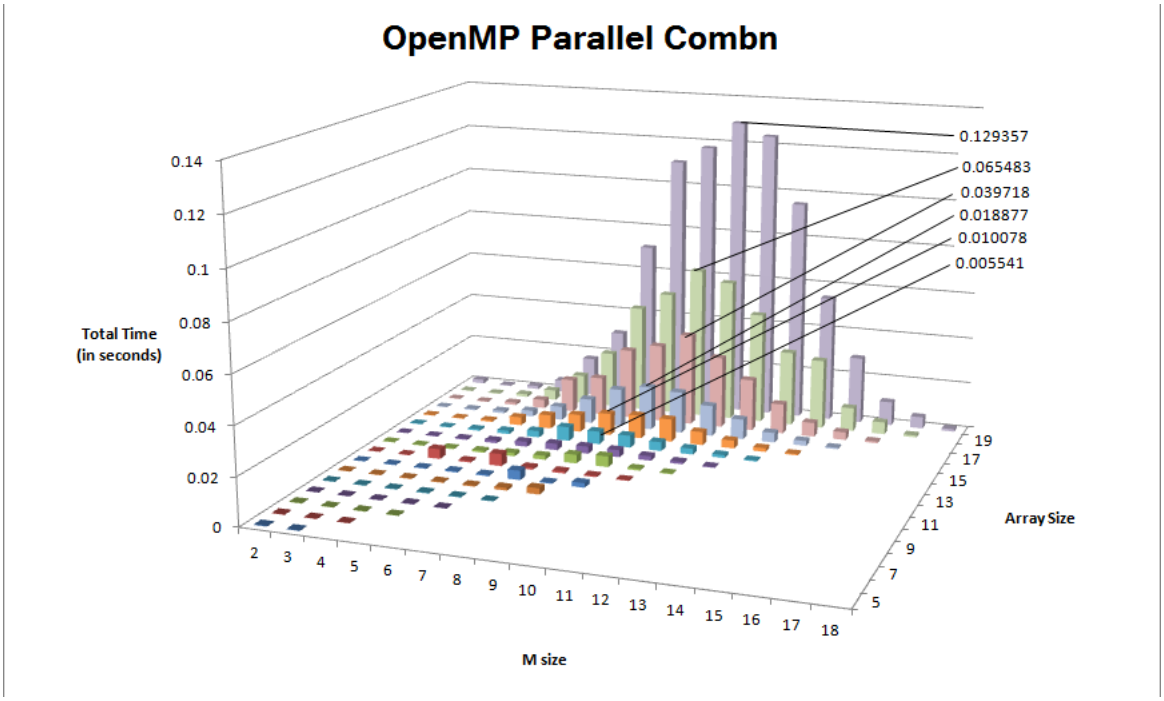
All the references to code will be references to Appendix: A.1

We decided to take a recursive approach to the implementation of the combinations function in C++. The core functionality of the program can be found on lines 23-33 in the method findCombs. This method takes in as its parameters 2 integers, a vector, and a 2d vector. Upon the first call of findCombs() the vector combination will already have been initialized to contain the first value in the combination (i.e. if we assume we are looking for the combination [1,2,3] the vector will be initialized to contain [1] and this is done on line 71 in the combn() function. From there the recursive algorithm will add the next number (i.e. combinations will now contain [1,2]) and then

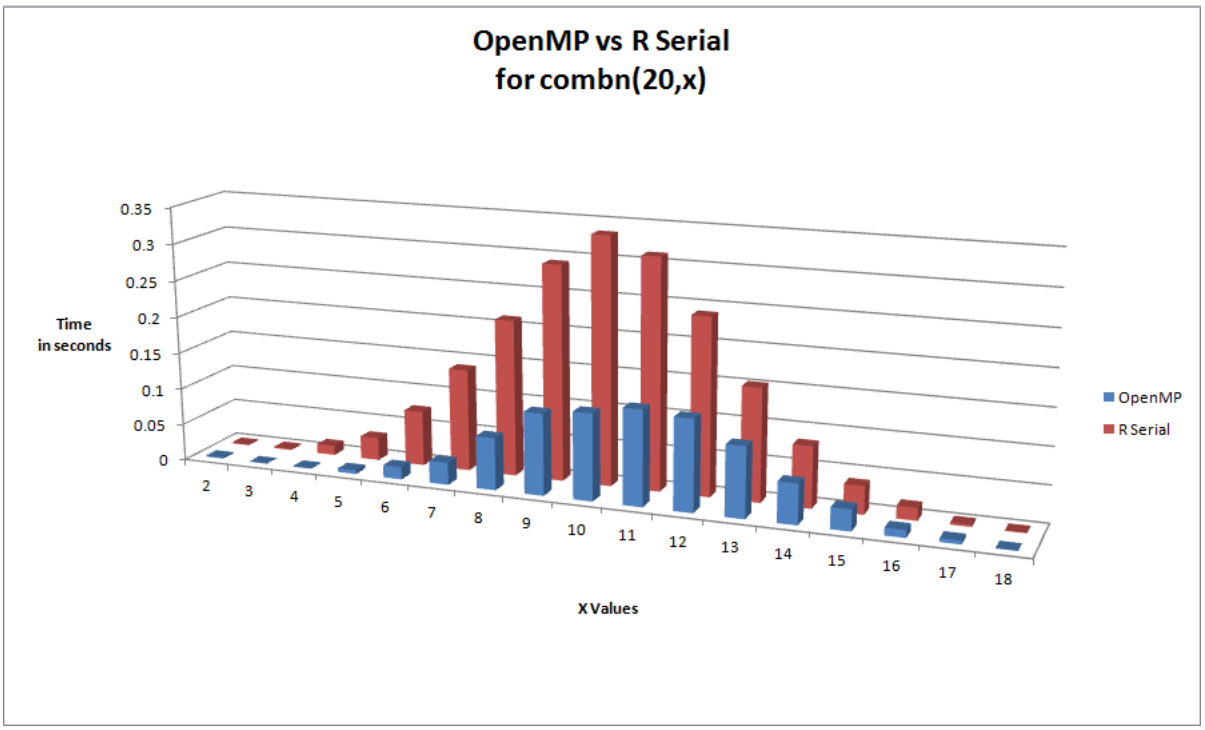
will call findCombs() recursively until the base case is met upon which the vector combination is pushed on to the 2D vector. When the recursion is done and the 2D vector contains all its possible combinations, the function vectorToArray() is called and the 2D vector is parsed and its values are added to the appropriate region of the global array.

2.3 Data

2.3.1 Graphic A1



2.3.2 Graphic A2



2.4 Analysis

Looking at `Graphic A1` we can see that our implementation of *combn*() in OpenMP follows the standard bell curve where given a value x, its maximum number of calculations and therefore its longest running time occurs when $m = \frac{x}{2}$. We can see this by looking at Pascal's Triangle [Appendix Graphic 1] which directly correlates to the problem of creating a list of combinations. Those values which lie at the center of each row, vertically down the middle, are those with the greatest value for that given row. These values correspond to $\binom{x}{m}$ where again $m = \frac{x}{2}$. When *combn*() is passed these values this is when we see the greatest running time for the function.

Looking at `Graphic A2` we can see that our implementation of *combn*() is on average 2-3 times faster than the serial built in version provided in the utils package of the R library. This is to be expected since we are able to generate combinations in parallel speed up could be further improved by taking our approach of parallelization further down and assigning tasks recursively.

3 combn() in Thrust

3.1 Strategy

For the Thrust version of the combinations function, we decided to parallelize by using bit vectors of size `n`. Specifically, we generate a bit vector filled with 1s and 0s. Initially, the bit vector is initialized with 1s (indicating true) from m-th positions till the end of the vector. We then

lexicographically generate the next permutations of the bit strings. The final operation involves finding the positions of 0s in parallel. Upon doing so, we find all the combinations.

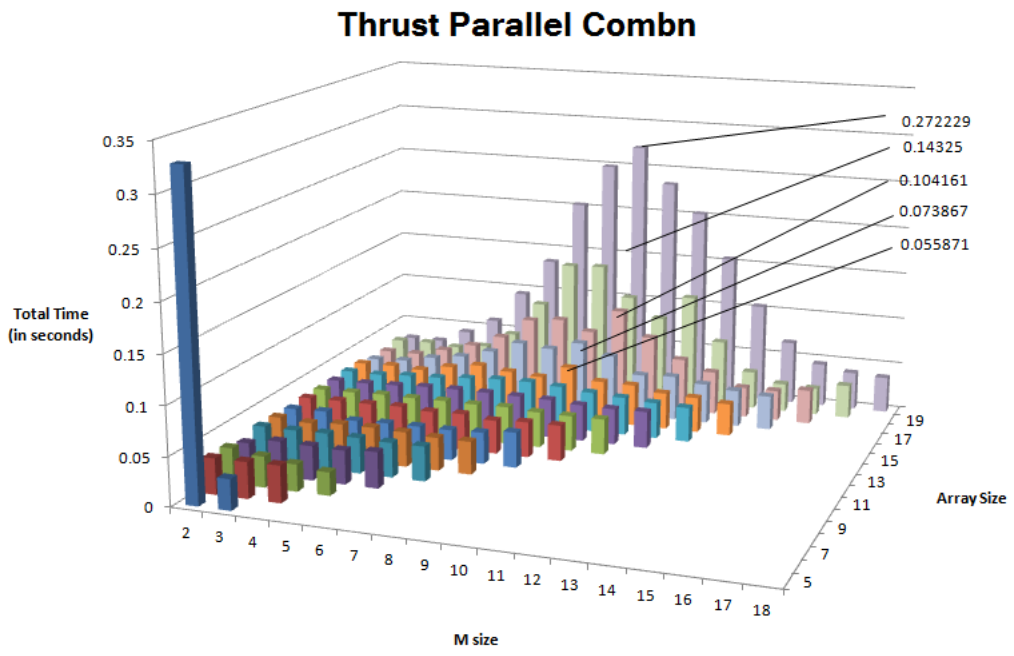
3.2 Implementation

All the references to code will be references to Appendix: A.2

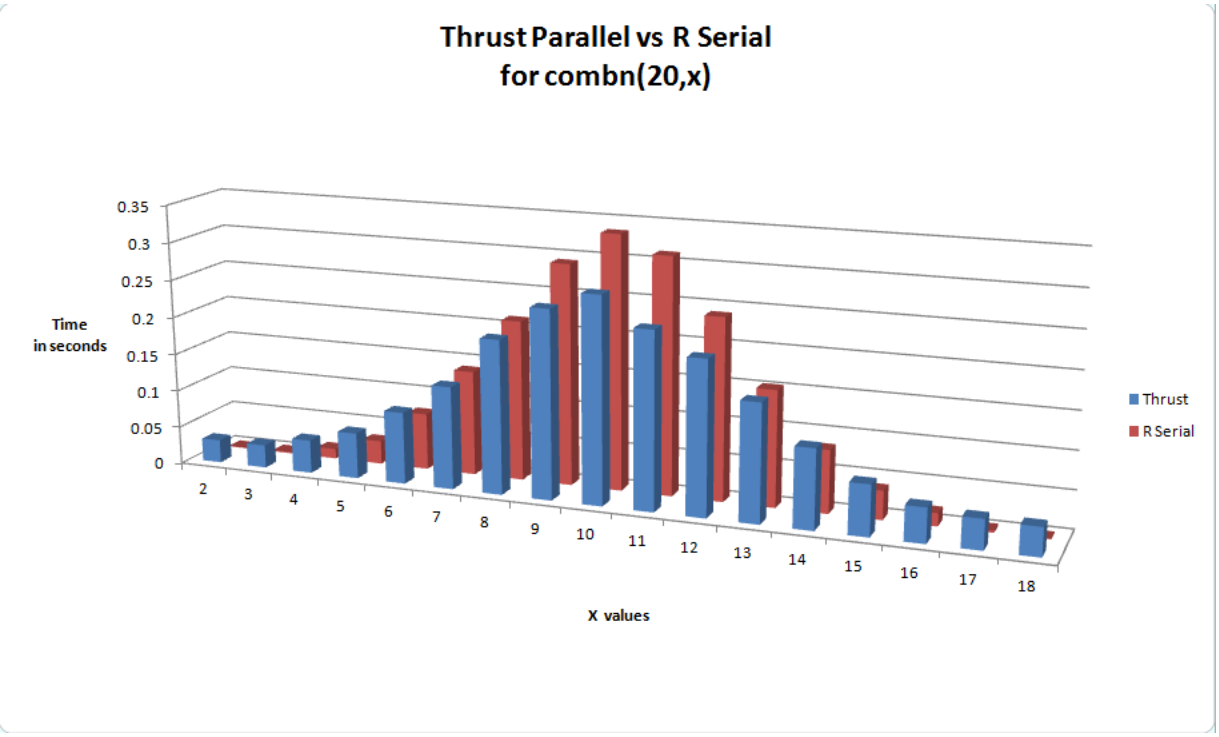
We decided to take an iterative approach to the implementation of the Thrust function in C++. The important aspect to note here is that the device functions in thrust can only be operated on 1d vectors as opposed to 2d vectors. Hence, the `combn()` function returns and operates on a 1d vector. The core functionality of the program can be found in the function `combn()` found on lines 56-82. The function takes in `n`(the endpoint of the sequence) and `r`(how many things to be taken at a time). In this function, we allocate certain space for the permutations and generate all the possibilities of the bit strings. Once we have found all the permutations of the bit strings, we essentially find all the indices where the element is equal to 0. This process takes advantage of powerful thrust functions such as `transform` and the fancy iterators of thrust such as the `zip_iterators()`. After we find all the positions where the bit value is 0, we transfer it to an array and return it.

3.3 Data

3.3.1 Graphic B1



3.3.2 Graphic B2



3.4 Analysis

Looking at `Graphic B1` we can see that our implementation of *combn*() in Thrust follows a similar bell curve to that of the one in our OpenMP implementation. There is a large difference though, in that in the Thrust implementation, you never see a close to zero time operation like you do in a lot of the cases for the OpenMP. This is due to the fact that Thrust operates on the GPU and as such has to transfer data both to and from the GPU before and after computation. This travel time between CPU and GPU accounts for the "baseline" being roughly .025 seconds meaning no operations ever run faster than this.

Looking at `Graphic B2` we can see that our implementation of *combn*() in Thrust outperforms the serial built in version provided in the utils package of the R library for values where given an x and m for $\binom{x}{m}$ the value of m is roughly $\frac{x}{4} < m < \frac{3x}{4}$. So for values which are generally computation heavy, the Thrust implementation is faster and provides performance gains, but for less computation heavy problems, the transport cost is not worth the small speed up of the parallelization.

4 combn() in R-Snow

4.1 Strategy

4.2 Implementation

4.3 Data

4.4 Analysis

5 Conclusion

Overall, the run times achieved by parallelization seem to have improvements, significantly so in the OpenMP implementation. Each run time seems to have its own pros and cons. Thrust provided strong performance gains over the serial version but required the overhead of having to transport data on a slow connection from the CPU to GPU. The algorithm we used for parallel computation was also probably not the most efficient and we could have seen even better gains in performance, but with Thrust this still doesn't matter for "easy" computations where transport time far exceeds computation time. OpenMP runs the fastest of the three and this is due to the fact that both our algorithm for OpenMP was probably the most efficient of our three algorithms and that there is not overhead necessary before beginning the computations. The biggest hurdle with our OpenMP implementation is making sure that it is load balanced.

6 Citations

R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

A Appendix: Code

A.1 combn() — OpenMP implementation

combnOpenMP.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <algorithm>    // std::find
#include <omp.h>

using namespace std;

vector<int> values;
int *allCombs;
int *numEntriesPerLevel;

double choose(int n, int k) {
    if(k == 0) return 1;
    return (n * choose(n - 1, k - 1)) / k;
}

void addComb(vector<vector<int> > &v, vector<int> &v2) {
  v.push_back(v2);
}

void findCombs(int offset, int k, vector<vector <int> > &v, vector<int>
    combination) {
  if (k == 0) {
    addComb(v, combination);
    return;
  }
  for(int i = offset; i <= values.size() - k; ++i) {
    combination.push_back(values[i]);
    findCombs(i + 1, k - 1, v, combination);
    combination.pop_back();
  }
}

void findEntriesPerLevel(int *array, int x, int m) {
  array[0] = 0;
  for(int i = 1; i < x - m + 1; i++) {
    array[i] = array[i - 1] + (choose(x - i, m - 1) * m);
  }
}

void vectorToArray(vector<vector <int> > &v, int a[], int startIndex) {
  vector< vector<int> >::const_iterator row;
    vector<int>::const_iterator col;
    int index = startIndex;

    for (row = v.begin(); row != v.end(); ++row)
    {
         for (col = row->begin(); col != row->end(); ++col)
         {
            a[index] = *col;
```

```
52              index++;
53            }
54      }
55  }
56
57  int * combn(int x, int m) {
58    int size = choose(x,m);
59    for(int i = 0; i < x; ++i)
60      values.push_back(i+1);
61
62    allCombs = new int[size * m];
63    numEntriesPerLevel = new int[x - m + 1];
64    findEntriesPerLevel(numEntriesPerLevel, x, m);
65
66
67    #pragma omp parallel for schedule(dynamic)
68    for(int i = 1; i <= x - m + 1; i++){
69      vector<vector <int> > levelCombinations;
70      vector<int> combination;
71      combination.push_back(i);
72      findCombs(i, m - 1, levelCombinations, combination);
73      vectorToArray(levelCombinations, allCombs, numEntriesPerLevel[i - 1]);
74      combination.pop_back();
75    }
76    #pragma omp barrier
77
78    return allCombs;
79  }
```

This is a recursive implementation of the combinations function. Here we used `#pragma omp parallel for schedule(dynamic)` on line 67 to help load balance the parallelization of the threading.

A.2 combn() — Thrust implementation

combn.cu

```
1  #include <thrust/device_vector.h>
2  #include <thrust/copy.h>
3  #include <thrust/iterator/zip_iterator.h>
4  #include <thrust/device_vector.h>
5  #include <thrust/tuple.h>
6  #include <vector>
7  #include <algorithm>
8  #include <iostream>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <thrust/reverse.h>
12 #include <thrust/iterator/counting_iterator.h>
13 #include <thrust/functional.h>
14 #include <thrust/sequence.h>
15 #include <thrust/remove.h>
16
17 typedef thrust::tuple<int, int>            tpl2int;
18 typedef thrust::device_vector<int>::iterator intiter;
19 typedef thrust::counting_iterator<int>     countiter;
20 typedef thrust::tuple<intiter, countiter>  tpl2intiter;
21 typedef thrust::zip_iterator<tpl2intiter>  idxzip;
```

```
22
23 long long int factorial(int n) {
24   long int prod = 1;
25   long int i;
26   for(i = 1; i <= n; i++) {
27     prod = prod * i;
28   }
29   return prod;
30 };
31
32
33 struct find_index : public thrust::unary_function<tpl2int, int>
34 {
35   const int sizeA;
36
37   find_index(int _sizeA) : sizeA(_sizeA) {}
38
39   __host__ __device__ int operator()(const tpl2int& x) const
40   {
41     // If an element is 0, then get the index of that element
42     if (x.get<0>() == 0) {
43       return (x.get<1>() % sizeA)+1;
44     }
45     else return -1;
46    }
47 };
48
49 struct remove_one {
50   __host__ __device__
51   bool operator() (const int i) {
52     return i != -1;
53   }
54 };
55
56 thrust::host_vector<int> combn(int n, int r) {
57   std::vector<int> v(n);
58   std::fill(v.begin() + r, v.end(), 1);
59   int chooseSize = factorial(n) / (factorial(n-r)*factorial(r));
60   thrust::host_vector<int> allBits( chooseSize * n  );
61   int offset = 0;
62
63   do {
64     thrust::copy(v.begin(), v.end(), allBits.begin() + offset);
65     offset += n;
66   } while(next_permutation(v.begin(), v.end()));
67
68   thrust::device_vector<int> dev_bits = allBits;
69   thrust::device_vector<int> allPositions(dev_bits.size());
70   thrust::counting_iterator<int> first_iter(0);
71   thrust::counting_iterator<int> last_iter = first_iter + dev_bits.size();
72   thrust::device_vector<int> allComb(chooseSize * r);
73
74   idxzip first = thrust::make_zip_iterator(thrust::make_tuple(dev_bits.begin(),
       first_iter));
75   idxzip last = thrust::make_zip_iterator(thrust::make_tuple(dev_bits.end(),
       last_iter));
76   thrust:: transform(first, last, allPositions.begin(), find_index(n));
77   thrust:: copy_if(allPositions.begin(), allPositions.end(), allComb.begin(),
       remove_one());
78
```

```
  thrust:: host_vector<int> returnArray(allComb.size());
  thrust:: copy(allComb.begin(), allComb.end(), returnArray.begin());
  return returnArray;
}

int main (int argc, char** argv) {
  int x = atoi(argv[1]);
  int m = atoi(argv[2]);
  thrust::host_vector<int> vals = combn(x, m);
  return 0;
}
```

A.3 combn() — R-Snow implementation

combn.R

```
#Arjun Bharadwaj abharadwaj@ucdavis.edu
#Bijan Agahi bsagahi@ucdavis.edu
#Stefan Peterson stpeterson@ucdavis.edu

#Import the existing libraries(mainly snow nad parallel)
#library("snow")

combn_R <- function(x, m, FUN = NULL, simplify = TRUE) {
  # Determine the total size
  size <- choose(x, m)
  # Generate from 1 to n range
  values <- seq_len(x)
  # All combinations
  allCombs <- vector(mode = "numeric", length = size * m)
  # Number of entries Per Level
  numEntriesPerLevel <- vector(mode = "numeric", length = x - m + 1)
  # All permutations
  allCombinations <- list()
  # Setup the initial stuff
  findEntriesPerLevel <- function() {
    # Indices in R start at 1
    numEntriesPerLevel[1] <- 0
    for(i in 2:(x - m + 1)) {
      numEntriesPerLevel[i] <- numEntriesPerLevel[i - 1] + (choose(x - i + 1, m -
          1) * m)
    }
    numEntriesPerLevel
  }

  push_back <- function(combination, value) {
    combination[length(combination)+1] <- value
  }

  pop_back <- function(combination) {
    combination <- combination[-(length(combination))]
  }

  # # Assume v is a list since we are pushing to a 2d vector
  # addComb <- function(v, v2) {
  #   v[[length(v) + 1]] <- v2
  # }
  position <- 1

  findCombs <- function(offset, k, v, combination) {
```

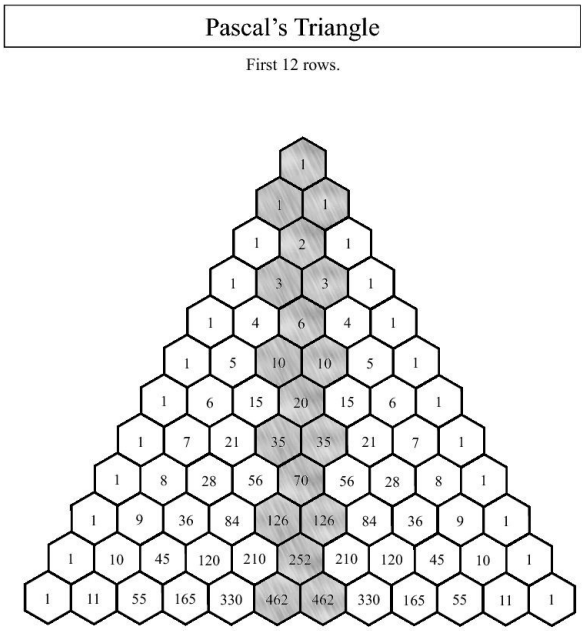
```
    #position <- position + 1
    if(k == 0) {
      allCombinations[[position]] <<- combination
      position <<- position + 1
      #v <- append(v, combination)
      #print(combination)
      #print("reaching 0")
      #print(allCombinations)
      return(0)
    }
    for(i in offset:((length(values) - k)+1) ) {
      combination <- c(combination, values[i])
      findCombs(i + 1, k - 1, v, combination)
      combination <- pop_back(combination)
    }
  }

  numEntriesPerLevel <- findEntriesPerLevel()
  #print(numEntriesPerLevel)

  # This is the main loop
  for(i in 1:(x - m + 1)) {
    # List of vectors. In C++, 2d Vector
    levelCombinations <- list()
    # Our local 1d array
    combination <- c()
    #i <- 1
    combination <- c(combination, i)
    # Call to recursive function
    findCombs(i + 1, m - 1, levelCombinations, combination)
    #print(combination)
    combination <- pop_back(combination)
    #print(paste("Now Popping back\n"))
  }
  print(allCombinations)

}

# Test code
x <- 5
m <- 3
combn_R(x, m)
```

This is a recursive implementation of the combinations function. It does not run in parallel due to time constraints and unforeseen complications but is a different approach than the iterative one taken by the combn() function found in the utils package of the R library.

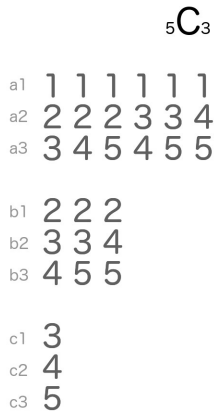
There are also a variety of files specific for testing purposes included in the submitted .tar file.

B Appendix: Graphics

B.1 Graphic D1



B.2 Graphic D2



C Appendix: Member Contributions

Stefan Peterson

- LaTeX Formatting, editing
- OpenMP implementation
- OpenMP write-up
- Abstract
- Introduction
- Conclusion
- Relevant research

Bijan Agahi

- R implementation
- R write-up
- Experiment data collection
- Test scripting
- Graphing in R, tabling
- Relevant research

Arjun Bharadwaj

- Thrust implementation
- Thrust write-up
- Relevant research
- R implementation