University of California, Davis

ECS 158

Programming on Parallel Architectures

Parallelization of R's combn() Function

Authors:
Trisha Funtanilla
Syeda Inamdar
Eva Li
Jennifer Wong

Professor:
Norm Matloff

March 20, 2015

Contents

1 Objective

Choose some function `g()` in R, either built-in to base R or in a CRAN package, to parallelize. Do so in Snow, OpenMP and CUDA (or substitute Thrust for either CUDA or OpenMP, but not both), and run timing tests.

2 Function

We chose to parallelize the function `combn()` from the "combinat" package in CRAN.

2.1 Description

Taken from the official documentation:[1]

> "Generate all combinations of the elements of x taken m at a time. If x is a positive integer, returns all combinations of the elements of seq(x) taken m at a time. If argument "fun" is not null, applies a function given by the argument to each point. If simplify is FALSE, returns a list; else returns a vector or an array. "..." are passed unchanged to function given by argument fun, if any."

2.2 Why Speedups Are Possible

We have tried `combn()`, which has the inputs `combn(x, m, fun=NULL, simplify=TRUE, ...)`, with different inputs for the vector x and value m. We noticed that `combn()` became quite slow once x had a size numbering around 100 or higher and m around 5 or higher, taking several minutes to compute (but not print) its results. We think that we can achieve speedups by dividing the input x equally among the threads/nodes, and parallelizing the work of creating the output matrix or list. The goal is to speed up the function, which, based on its source code, currently computes combinations serially.

3 Parallelization

3.1 OpenMP

The C++ OpenMP program is called through R via .Call() using the Rcpp interface. The function `combnomp()` accepts the same input arguments as the original function `combn()` from the CRAN package. `combnomp()` can also handle the same types of valid inputs (see Section 2.1), and through new optional arguments, also allows the user to decide on a scheduling policy and chunk size.

Usage:

```
combnomp <- function(x, m, fun = NULL, simplify = TRUE, sched = NULL,
chunksize = NULL, ...)
```

where `x` is the input vector of integers and/or characters, `m` is number of elements per combination `fun` is the function to be applied to the resulting output, `simplify` indicates whether the output must be printed as a matrix (set to `TRUE`) or as a list (set to `FALSE`), `sched` is the scheduling policy (static, dynamic, or guided; default is static), and `chunksize` is the chunk size for the scheduling policies (default is 1), and `...` are the parameters for `fun`.

Code highlights:

- Load balancing algorithm (lines 101 - 106 of `combn-omp.cpp`, described in the next section)
- No critical sections or barriers (except for the implicit barrier after the `parallel` pragma)

Other notes:

- Similar to the original function, the OpenMP implementation has a limit on the input/output size. The program terminates if R decides that it cannot allocate enough space for the program to successfully run. The max size of `x` depends on the value of `m`.

3.1.1 Load Balancing

Good load balancing is very critical to our program because for each element i in the input vector x, the number of combinations that can be generated with x_i is larger than that with x_{i+1}. A naive task distribution without proper load balancing could heavily skew the distribution of the workload for each thread. For example, directly assigning an x_i for each thread will result in the threads with lower id's generating far more combinations (thus, more work) than threads with higher id's.

In order to compensate for the load imbalance, the assignment of tasks to the threads was done in a wrap around manner [2]. Using this method, for the first distribution, each thread will work on the element indexed in the same value as their id (e.g. thread 0 on $x[0]$, thread 1 on $x[1]$, and so on). Then, the following distribution depends on the total number of threads, the id of each thread, and the index of its previous assignment. With nth corresponding to the total number of threads and me corresponding to the thread id, the assignment following the initial distribution is determined using the equation:

$$next\_task = prev\_task + 2 * (nth - me - 1) + 1 \tag{1}$$

The distribution that will then follow the above depends on the id of each thread and the index of the previous task assigned to that particular thread. This next distribution is determined by the equation:

$$next\_task = prev\_task + 2 * me + 1 \tag{2}$$

The succeeding distributions alternate in using the two equations until the last distribution, which is for $n - m + 1$, or the last x_i that can form a combination of size m together with

the elements succeeding it. In this algorithm, each thread knows which indexes in x to generate combinations for. The threads do not need to communicate with each other as they work on their tasks, avoiding huge overhead especially for large values of n.

3.1.2 Chunk Size

The equations from the section above implicitly defines the chunk size so that each thread gets assigned roughly the same number x_i's to work on. It is essentially the number of times task distributions occur in the load balance algorithm, which is roughly $(n - m + 1)/nth$.

3.1.3 Scheduling Policies

The load balancing algorithm described in Section 3.1.1 implements a static scheduling policy since the threads are assigned specific tasks and only work on the tasks assigned to them. For the purposes of timing comparisons, additional optional arguments `sched` and `chunksize` were added to the function in order to test the speed of the program for the other scheduling policies, namely dynamic and guided. If both arguments are not provided (or set as NULL), the program defaults to the static scheduling policy using the load balancing algorithm. If either dynamic or guided is set, the program uses OpenMP's built-in `schedule` clause to distribute the tasks to the threads. If `chunksize` is not provided, the program sets it to the default chunk size of 1.

We performed various tests using the three scheduling policies and experimented with different chunk sizes (for dynamic and guided). It was clear that the load balancing algorithm using static scheduling was the fastest. It eliminates the communication overhead of the dynamic and guided scheduling policies since the threads don't have to repeatedly access the task farm for work, and idleness is still reduced because of the way the tasks are distributed among the threads.

Hence, the static method is the one compared to the original function in the timing comparisons and analysis.

3.1.4 Cache and Other Performance Tuning Considerations

The following performance tuning applies only to the load balancing algorithm described in Section 3.1.1.

- Avoiding instances of false sharing: In order to avoid instances of false sharing (and the resulting cache coherence overheads), we avoided having shared data that could be modified by multiple threads. The data frequently accessed and shared by the threads in the program, such as the input and position vectors are read-only data. Hence, cache lines are not invalidated whenever data from these vectors are accessed. The main shared data structure that is modified by all threads is the output matrix, but the threads never have to perform any read on this data.
- Synchronization overhead: The threads are made to work as independent of each other as possible. The only synchronization occurs implicitly at the end of the `parallel` pragma.

- Avoidance of critical sections and barriers (the only barrier is the implicit barrier at the end f the `parallel` pragma)

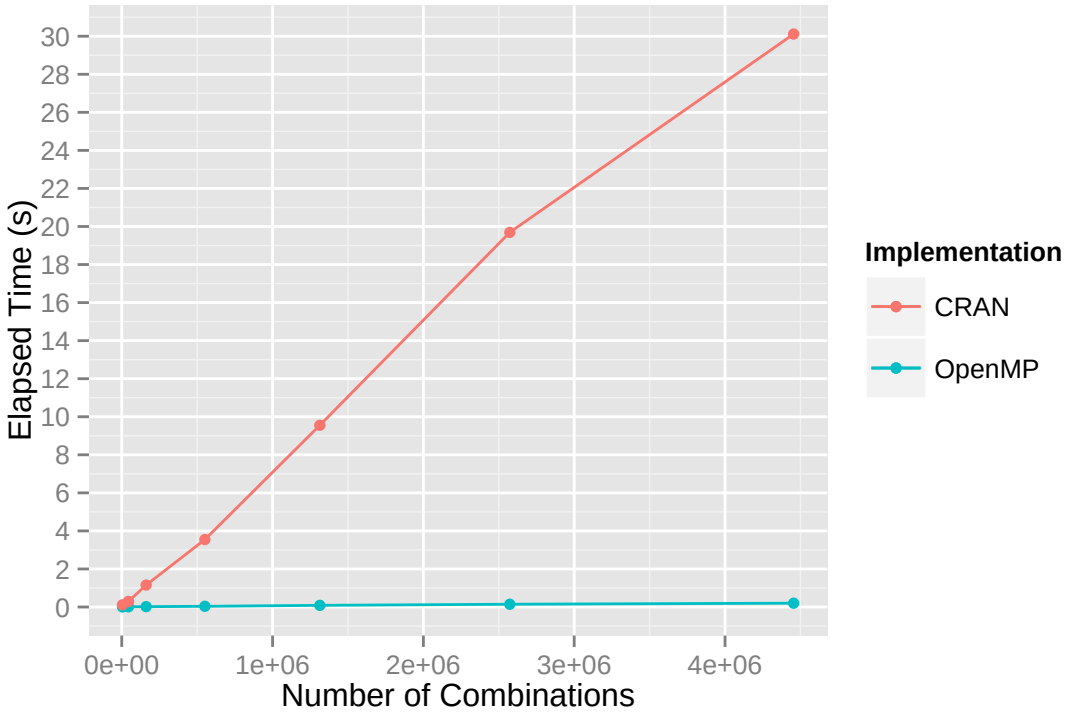
3.1.5 Comparative Analysis

The following input sizes were used for comparing the speeds of the source code and our OpenMP implementation using 8 threads set on the environment. The function `nCm()` is also from the CRAN "combinat" package. `nCm()` is the function that calculates the binomial coefficient ("n choose m") or the total number of combinations of size m given n elements. The third argument controls the rounding of the integers. [1]

```
> nCm(100, 2, 0.10000000000000002)
[1] 4950
> nCm(300, 2, 0.10000000000000002)
[1] 44850
> nCm(100, 3, 0.10000000000000002)
[1] 161700
> nCm(150, 3, 0.10000000000000002)
[1] 551300
> nCm(200, 3, 0.10000000000000002)
[1] 1313400
> nCm(250, 3, 0.10000000000000002)
[1] 2573000
> nCm(300, 3, 0.10000000000000002)
[1] 4455100
```

The elapsed time that is recorded for each test is the average of three trials. The OpenMP parallelization massively improved the performance of the function. The improvement becomes increasingly invaluable as the input size increases (e.g. for `nCm(100, 5)`, the OpenMP implementation still manages to produce an output under 5 seconds, while the original function takes several minutes).

The following plot illustrates the differences in the speeds of the two implementations.



3.2 Snow

The function `combnsnow()` accepts the same input arguments as the original function `combn()` from the CRAN package. `combnsnow()` can also handle the same types of valid inputs (see Section 2.1).

Usage:

```
combnsnow <- function(cls, x, m, fun = NULL, simplify = TRUE, ...)
```

where `cls` is the clusters, `x` is the input vector of integers and/or characters, `m` is number of elements per combination `fun` is the function to be applied to the resulting output, `simplify` indicates whether the output must be printed as a matrix (set to `TRUE`) or as a list (set to `FALSE`), and `...` are the parameters for `fun`.

Code highlights:

- Load balancing algorithm implemented in R Snow

Other notes:

- Again, the program terminates if R decides that it cannot allocate enough space for the program to successfully run.

3.2.1 The Issue of Communication Overhead

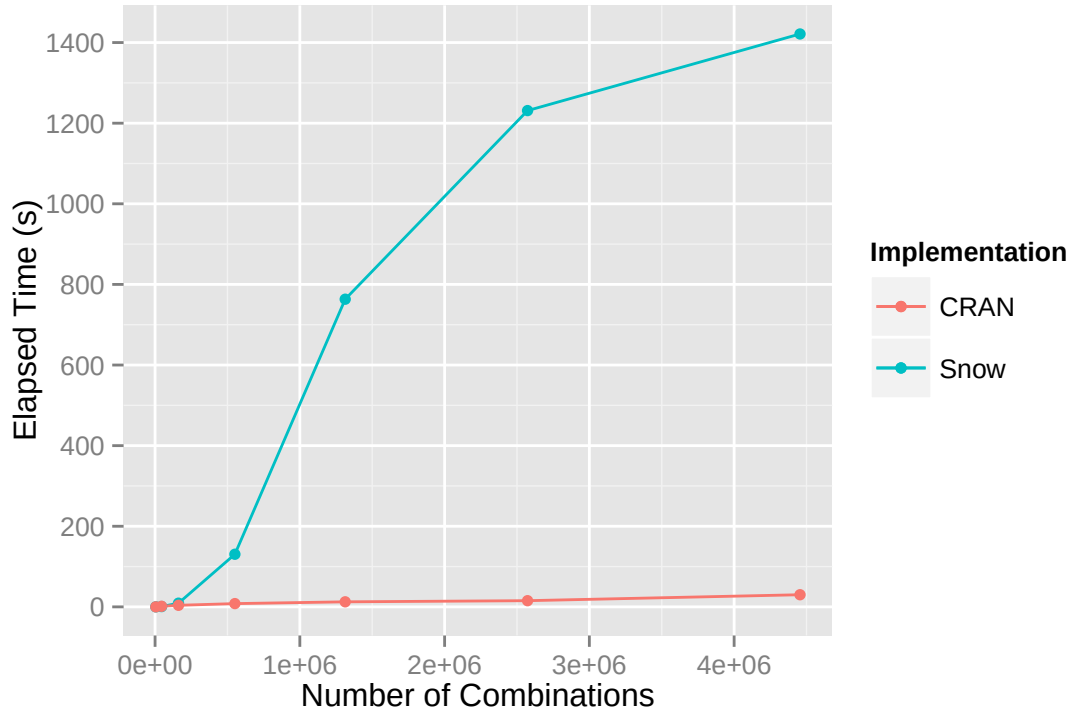
After running tests using small inputs, it was observed that parallelization using Snow more often than not took longer than running the CRAN function `combn()`. This is because of the communication overhead. The communication between the nodes in the cluster took more time than the actual computations in the function. If the jobs sent to the worker nodes are not relatively computationally extensive, the overhead of communicating ends up deteriorating the performance.

3.2.2 Reducing Network Overhead

Communication is much slower than computation. There is already communication overhead to setting up clusters. In order to reduce network overhead and improve the performance, the code was written so that the nodes will do long calculations, and the tasks are going to be pre-assigned so that there will be less communications among the nodes as each work on its own tasks. The same load balancing algorithm as in Section 3.1.1 was used to distribute the tasks to each nodes.

3.2.3 Comparative Analysis

The same input sizes and function arguments as in Section 3.1.5 were used for testing. For the Snow tests, 8 clusters were used. The following plot illustrates the differences in speeds between the Snow and CRAN implementations:



Despite our efforts to achieve speedups, the Snow implementation was actually much slower than the serial implementation of `combn()`. While the load balancing algorithm

helped in distributing the tasks, there are other aspects in the code that makes the program slower. For instance, sorting and formatting of the output is not in parallel, which can take a really long time in R when the input size is really big. There is also the inevitable communication overhead due to the use of clusters.

3.3 Thrust

The function `combnthrust()` accepts the same input arguments as the original function `combn()` from the CRAN package. `combnthrust()` can also handle the same types of valid inputs (see Section 2.1).

Usage:

```
combnthrust <- function(x, m, fun = NULL, simplify = TRUE, ...)
```

where `x` is the input vector of integers and/or characters, `m` is number of elements per combination `fun` is the function to be applied to the resulting output, `simplify` indicates whether the output must be printed as a matrix (set to `TRUE`) or as a list (set to `FALSE`), and `...` are the parameters for `fun`.

Code highlights:

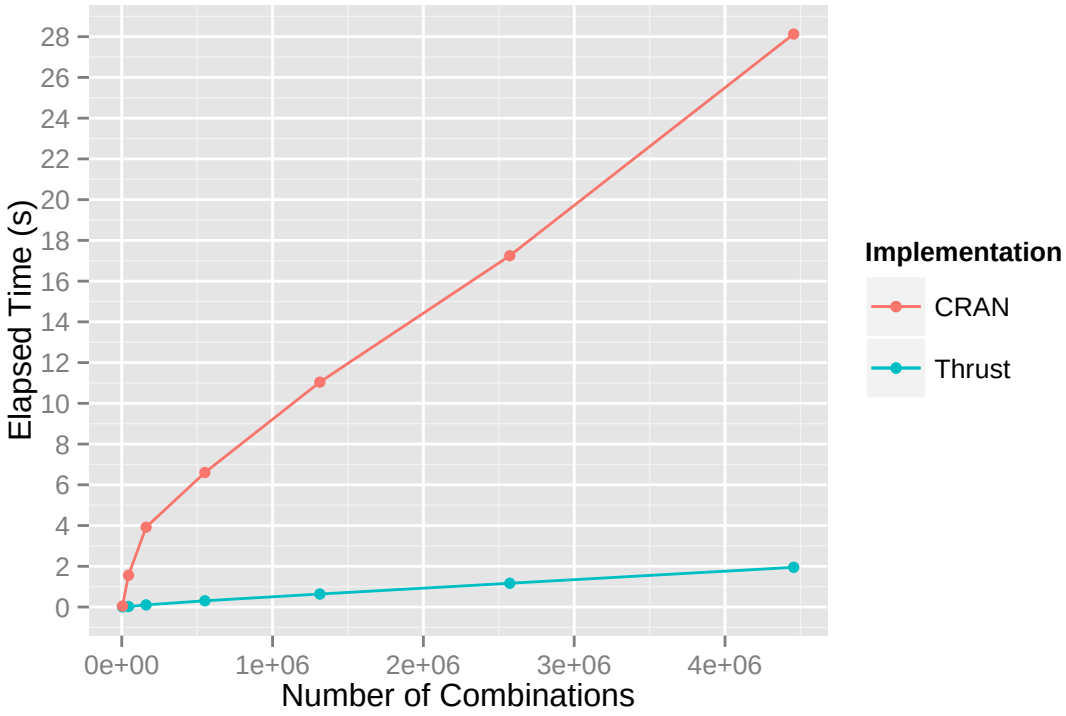
- We followed this tutorial [3] to link Rcpp, Thrust, and C++.
- We used Thrust's for_each function to call on the functor that finds the combination for each element, up to `n - m + 1` in a parallel manner.

Other notes:

- The program terminates if Thrust is unable to allocate enough space on the GPU to have the program run successfully.
- Since Thrust calls are CUDA kernel calls, there is some latency involved in this, which slows the program down.
- Thrust uses shared memory because the CUDA backend utilizes the GPU and GPUs have shared memory.

3.4 Comparative Analysis

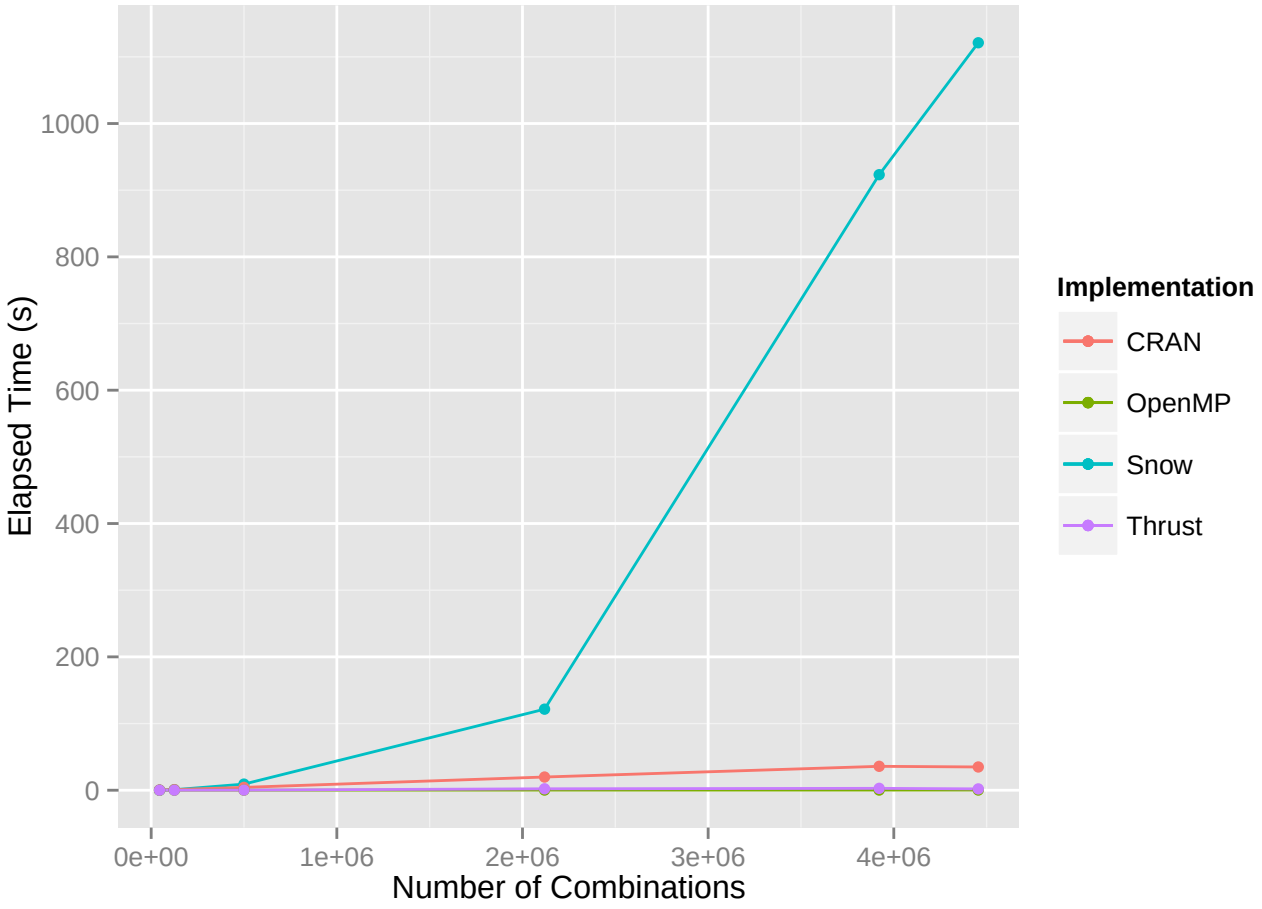
The same input sizes and function arguments as in Section 3.1.5 were used for testing Thrust. The following plot illustrates the differences in speeds between the Thrust and CRAN implementations:



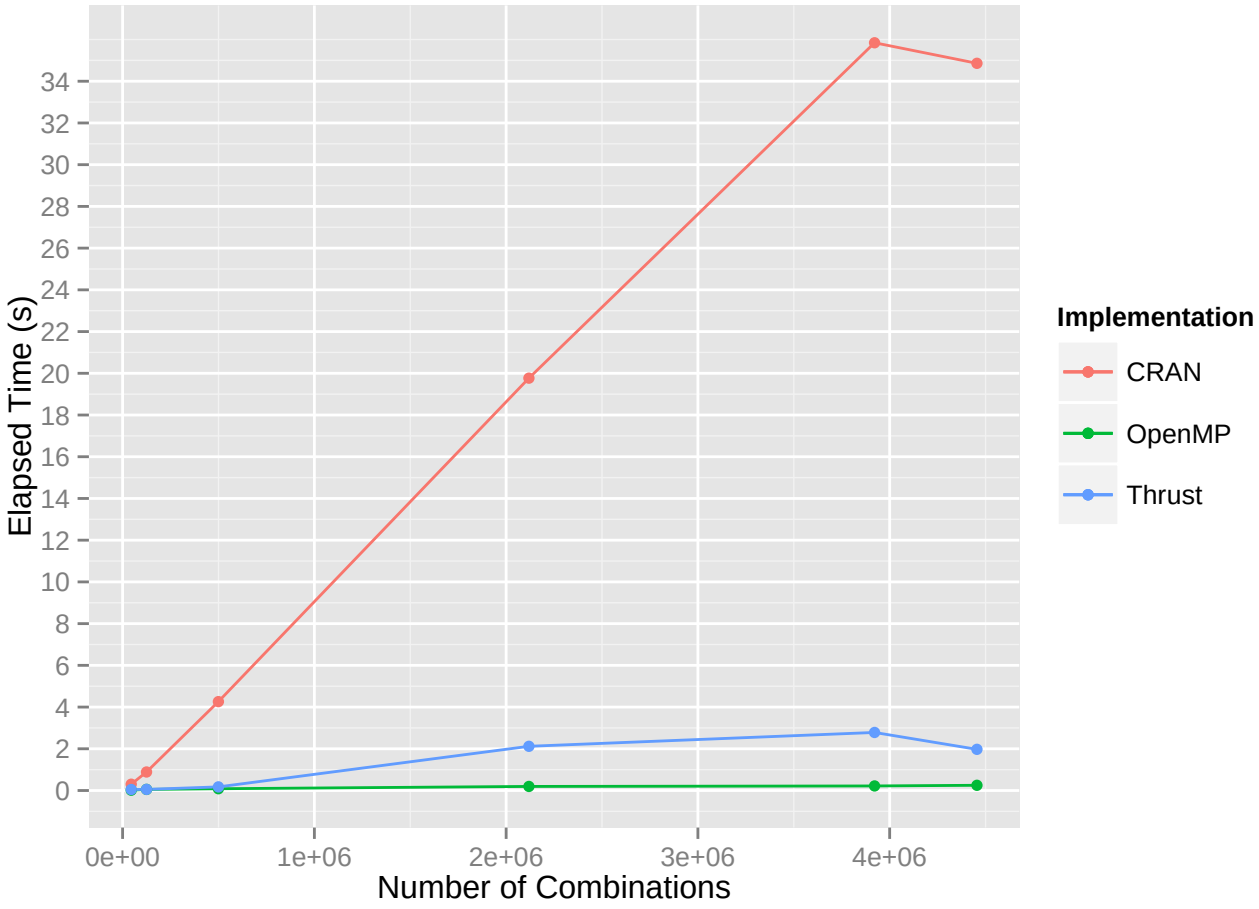
We were able to achieve speedup when using Thrust over the CRAN implementation for smaller input sizes of less than 600. Although the program terminated if the GPU could not allocate enough memory for large input sizes, Thrust was consistently faster than CRAN by using a functor. The program also modified the elements of the output matrix in parallel by making sure the indices didn't conflict.

4 Timing Comparisons and Analysis

The following plot compares all four implementations for various values of `x` and `m`. The number of clusters used for Snow and the number of threads used for OpenMP are both 8.



Clearly, the Snow implementation is simply too slow. By excluding Snow, we can have a closer look on the speeds of the other three implementations:



OpenMP and Thrust both massively improved the performance of `combn()` especially for large inputs. OpenMP was faster, and it also worked for even bigger inputs. We were able to get results for up to 75,287,520 combinations (`nCm(100, 5)`) in under 5 seconds. The differences in speeds between the OpenMP and Thrust implementations started to become quite significant at around 20,708,500 combinations (`nCm(500, 3)`). The Thrust program also started to have memory issues past this input size because of the limitations in the memory of the GPU. The speeds for OpenMP and Snow also didn't vary very significantly when using threads/nodes in the range of 6-10.

5 Conclusions

A Appendix

```
1  ##############################################################################
2  # R call function for the OpenMP parallelization of combn() from the CRAN
3  # combinat package: http://cran.r-project.org/web/packages/combinat/index.html
4
5  # Function Arguments:
6  # x <- input vector of integers and/or characters
7  # m <- number of elements in a combination
8  # fun <- function to apply to the resulting output
9  # simplify <- if TRUE, print output as a matrix with m rows and nCm columns
10   # else <- print output as a list
11   # nCm is the total number of combinations generated
12 # sched <- scheduling policy: static, dynamic, guided; default is static
13 # chunksize <- chunksize for scheduling policy; default is 1
14 # ... <- parameters for fun
15
16 # Helper functions for handling characters in input vector x:
17 # is.letter <- function to check if there's a char in x
18 # asc <- convert char to ASCII decimal value
19 # chr <- convert decimal value to ASCII character
20 # nCm <- calculate the total number of combinations
21   # taken directly from R combinat package nCm.R
22   # inserted into this file saw combinat need not be installed when program is run
23 ##############################################################################
24
25 combnomp <- function(x, m, fun = NULL, simplify = TRUE,
26   sched = NULL, chunksize = NULL, ...)
27 {
28   require(Rcpp)
29   dyn.load("combn-omp.so")
30
31   # Input checks taken directly from combn source code: combn.R
32   # from the CRAN combinat package
33   if(length(m) > 1) {
34     warning(paste("Argument m has", length(m),
35       "elements: only the first used"))
36     m <- m[1]
37   }
38   if(m < 0)
39     stop("m < 0")
40   if(m == 0)
41     return(if(simplify) vector(mode(x), 0) else list())
42   if(is.numeric(x) && length(x) == 1 && x > 0 && trunc(x) == x)
43     x <- seq(x)
44   n <- length(x)
45   if(n < m)
46     stop("n < m")
47
48   # total number of combinations
49   count <- nCm(n, m, 0.10000000000000002)
50
51   # Error checks for the scheduling variables: sched and chunksize
52   # R handles the error when 'sched' is not a string/character vector
53   # If sched is provided, then sched must be static, dynamic, guided, or NULL
54   if (!grepl('static', sched) && !grepl('dynamic', sched) && !grepl('guided', sched) &&
         !is.null(sched)) {
55       stop("Scheduling policy must be static, dynamic, or guided.")
56   }
57   # Set to default values depending on what is/are provided
58   if (is.null(sched) && is.null(chunksize)) {
59     sched <- 'static'
60     chunksize <- 1
61   }
62   else if (!is.null(sched) && is.null(chunksize)) {
63     chunksize <- 1
64   }
65   else if (is.null(sched) && !is.null(chunksize)) { # if sched is provided, but chunk
         size is not
```

```
66      sched <- 'static'
67      warning("'sched' is replaced with default 'static' and 'chunksize' is overriden with
            default value.")
68    }
69
70    # Checks if input vector x has characters
71    # If so, then convert chars to their ASCII decimal values
72    # Operate on the ASCII decimal values for the chars
73    ischarx <- match('TRUE', is.letter(x))
74    if (!is.na(ischarx)) {
75      ischarx_arr <- is.letter(x)
76      for (i in 1:length(ischarx_arr)) {
77        if (ischarx_arr[i]) {
78          if (length(asc(x[i])) == 1) {
79            x[i] <- asc(x[i])
80          }
81          else {
82            x[i] <- as.character(x[i])
83          }
84        }
85
86      }
87      x <- strtoi(x, base=10)
88    }
89
90
91    # Calculate positions for output
92    # Need this to make sure output is sorted
93    pos <- vector()
94    temp_n <- n
95    for (i in 1:(n-m+1)) {
96      pos <- c(pos, nCm(temp_n-i, m-1))
97    }
98    temp <- pos[1]
99    pos[1] <- 0
100   for (i in 2:length(pos)) {
101     temp2 <- pos[i]
102     pos[i] <- temp
103     temp <- pos[i] + temp2
104   }
105
106   # Initialize output matrix
107   retmat <- matrix(0, m, count)
108   # Call the function through Rcpp
109   retmat <- .Call("combn", x, m, n, count, sched, chunksize, pos)
110
111   # Convert from ASCII decimal values back to chars if necessary
112   if (!is.na(ischarx)) {
113     for (i in 1:length(retmat)) {
114       if ((as.integer(retmat[i]) >= 97 && as.integer(retmat[i]) <= 122)
115       || (as.integer(retmat[i]) >= 65 && as.integer(retmat[i]) <= 90)) {
116         retmat[i] <- chr(retmat[i]);
117       }
118     }
119   }
120
121   # Apply provided function to the output
122   if (!is.null(fun)) {
123     retmat <- apply(retmat, 2, fun(...))
124   }
125
126   # Format results
127   if (simplify) {
128     out <- retmat
129   }
130   else {
131     out <- list()
132     for (i in 1:ncol(retmat)) {
133       out <- c(out, list(c(retmat[, i])))
134     }
```

```
  }

  return(out)
}

##############################################################################################
# Helper Functions
##############################################################################################

# function to check if there's a char in x
is.letter <- function(x) grepl("[[:alpha:]]", x)

# convert char to ascii decimal value
asc <- function(x) { strtoi(charToRaw(x),16L) }

# convert decimal value to ascii character
chr <- function(n) { rawToChar(as.raw(n)) }

# n choose m - calculates the total number of combinations for a given input
"nCm"<-
function(n, m, tol = 9.9999999999999984e-009)
{
#  DATE WRITTEN:  7 June 1995               LAST REVISED:  10 July 1995
#  AUTHOR:  Scott Chasalow
#
#  DESCRIPTION:
#        Compute the binomial coefficient ("n choose m"),  where n is any
#        real number and m is any integer.  Arguments n and m may be vectors;
#        they will be replicated as necessary to have the same length.
#
#        Argument tol controls rounding of results to integers.  If the
#        difference between a value and its nearest integer is less than tol,
#        the value returned will be rounded to its nearest integer.  To turn
#        off rounding, use tol = 0.  Values of tol greater than the default
#        should be used only with great caution, unless you are certain only
#        integer values should be returned.
#
#  REFERENCE:
#        Feller (1968) An Introduction to Probability Theory and Its
#        Applications, Volume I, 3rd Edition, pp 50, 63.
#
  len <- max(length(n), length(m))
  out <- numeric(len)
  n <- rep(n, length = len)
  m <- rep(m, length = len)
  mint <- (trunc(m) == m)
  out[!mint] <- NA
  out[m == 0] <- 1  # out[mint & (m < 0 | (m > 0 & n == 0))] <-  0
  whichm <- (mint & m > 0)
  whichn <- (n < 0)
  which <- (whichm & whichn)
  if(any(which)) {
    nnow <- n[which]
    mnow <- m[which]
    out[which] <- ((-1)^mnow) * Recall(mnow - nnow - 1, mnow)
  }
  whichn <- (n > 0)
  nint <- (trunc(n) == n)
  which <- (whichm & whichn & !nint & n < m)
  if(any(which)) {
    nnow <- n[which]
    mnow <- m[which]
    foo <- function(j, nn, mm)
    {
      n <- nn[j]
      m <- mm[j]
      iseq <- seq(n - m + 1, n)
      negs <- sum(iseq < 0)
      ((-1)^negs) * exp(sum(log(abs(iseq))) - lgamma(m + 1))
    }
```

```r
    out[which] <- unlist(lapply(seq(along = nnow), foo, nn = nnow,
      mm = mnow))
  }
  which <- (whichm & whichn & n >= m)
  nnow <- n[which]
  mnow <- m[which]
  out[which] <- exp(lgamma(nnow + 1) - lgamma(mnow + 1) - lgamma(nnow -
    mnow + 1))
  nna <- !is.na(out)
  outnow <- out[nna]
  rout <- round(outnow)
  smalldif <- abs(rout - outnow) < tol
  outnow[smalldif] <- rout[smalldif]
  out[nna] <- outnow
  out
}
```

```cpp
/*****************************************************************************
OpenMP (C++) implementation of R's combn() function from the CRAN combinat package

Called from combn-omp.R using .Calll() through Rcpp interface
*****************************************************************************/
#include <Rcpp.h>
#include <omp.h>

using namespace std;
using namespace Rcpp;

// Computes the indices of the next combination to generate
// The indices then get mapped to the actual values from the input vector
int next_comb(int *comb, int m, int n)
{
  int i = m - 1;
  ++comb[i];

  while ((i >= 0) && (comb[i] >= n - m + 1 + i)) {
    --i;
    ++comb[i];
  }

  if(comb[0] == 1) {
    return 0;
  }

  for (i = i + 1; i < m; ++i) {
    comb[i] = comb[i - 1] + 1;
  }

  return 1;
}

RcppExport SEXP combn(SEXP x_, SEXP m_, SEXP n_, SEXP nCm_, SEXP sched_, SEXP chunksize_
    , SEXP pos_, SEXP out)
{
  // Convert SEXP variables to appropriate C++ types
  NumericVector x(x_); // input vector
  NumericVector pos(pos_); // position vector for the combinations so that the output is
       sorted
  int m = as<int>(m_), n = as<int>(n_), nCm = as<int>(nCm_), chunksize = as<int>(
      chunksize_);
  string sched = as<string>(sched_);

  NumericMatrix retmat(m, nCm);

  // OpenMP schedule clauses
  if (sched == "dynamic") {
    omp_set_schedule(omp_sched_dynamic, chunksize);
  }
  else if (sched == "guided") {
```

```
50      omp_set_schedule(omp_sched_guided, chunksize);
51    }
52
53    if (sched == "static") { // use load balancing algorithm
54      #pragma omp parallel
55      {
56        // this thread id, total number of threads, combination indexes array
57        int me, nth, *comb;
58
59        nth = omp_get_num_threads();
60        me = omp_get_thread_num();
61
62        // array that will hold all of the possible combinations
63        // of size m of the indexes
64        comb = new int[m];
65
66        // initialize comb array
67        for (int i = 0; i < m; ++i) {
68          comb[i] = i;
69        }
70
71        int chunkNum = 1; // the number of chunk that has been distributed
72        int mypos; // variable for the output position
73
74        // each thread gets assign a chunk to work on
75        // each thread will have about the same number of chunks
76        // to work on throughout the lifetime of the program
77        for(int current_x = me; current_x < n-m+1; current_x+=1) {
78          int temp;
79          mypos = pos[current_x];
80          for (int i = 0; i < m; ++i) {
81            temp = comb[i] + current_x;
82            retmat(i, mypos) = x[temp];
83          }
84          mypos++;
85          while(next_comb(comb, m, n-current_x))  {
86            int temp;
87            for (int i = 0; i < m; ++i) {
88              temp = comb[i] + current_x;
89              retmat(i, mypos) = x[temp];
90            }
91            mypos++;
92          }
93
94          // reset comb array for the next chunk this thread will work on
95          for(int i = 0; i < m; i++) {
96            comb[i] = i;
97          }
98
99          chunkNum++; // increment chunkNum for the next chunk distribution
100         // determine which element this thread will work on
101         if (chunkNum % 2 == 0) {
102           current_x = current_x + 2 * (nth - me - 1);
103         }
104         else {
105           current_x = current_x + 2 * me;
106         }
107       }
108     }
109   }
110   else { // dynamic or guided; use OpenMP's schedule clause
111     int mypos;
112     #pragma omp parallel
113     {
114       int *comb = new int[m];
115       for (int i = 0; i < m; ++i) {
116         comb[i] = i;
117       }
118
119       #pragma omp for schedule(runtime)
```

```
      for(int current_x = 0; current_x < (n - m + 1); current_x++) {
        int temp;
        mypos = pos[current_x];
        for (int i = 0; i < m; ++i) {
          temp = comb[i] + current_x;
          retmat(i, mypos) = x[temp];
        }
        mypos++;
        while(next_comb(comb, m, n-current_x))  {
          int temp;
          for (int i = 0; i < m; ++i) {
              temp = comb[i] + current_x;
              retmat(i, mypos) = x[temp];
          }
          mypos++;
        }

        for(int i = 0; i < m; i++) {
          comb[i] = i;
        }
      }
    }
  }

  return retmat;
}
```

```
############################################################################################
# Snow parallelization of combn() from the CRAN
# combinat package: http://cran.r-project.org/web/packages/combinat/index.html

# Function Arguments:
# cls <- clusters
# x <- input vector of integers and/or characters
# m <- number of elements in a combination
# fun <- function to apply to the resulting output
# simplify <- if TRUE, print output as a matrix with m rows and nCm columns
  # else <- print output as a list
  # nCm is the total number of combinations generated
# ... <- parameters for fun

# Helper functions for handling characters in input vector x:
# nCm <- calculate the total number of combinations
  # taken directly from R combinat package nCm.R
  # inserted into this file saw combinat need not be installed when program is run
############################################################################################

combnsnow <- function(cls, x, m, fun = NULL, simplify = TRUE, ...) {
  # Input checks taken directly from the source code
  if(length(m) > 1) {
    warning(paste("Argument m has", length(m),
                  "elements: only the first used"))
    m <- m[1]
  }
  if(m < 0)
    stop("m < 0")
  if(m == 0)
    return(if(simplify) vector(mode(x), 0) else list())
  if(is.numeric(x) && length(x) == 1 && x > 0 && trunc(x) == x)
    x <- seq(x)
  n <- length(x)
  if(n < m)
    stop("n < m")
  nofun <- is.null(fun)
  count <- nCm(n, m, 0.10000000000000002)
  retval <- mycombn(cls, x, m)
  retval <- array(unlist(retval))
  # apply function
  if (!nofun) {
```

```
43      retval <- sapply(retval, fun)
44    }
45    # format output
46    if(!simplify) {
47      mat <- matrix(retval, m, count)
48      retval <- mat
49      l <- list()
50      for (i in 1:count) {
51        l <- c(l, list(c(retval[, i])))
52      }
53      retval <- l
54    }
55    else {
56      mat <- matrix(retval, m, count)
57      retval <- mat
58    }
59    return(retval)
60  }
61  next_comb <- function(comb, k, n) {
62    i <- k
63    comb[i] <- comb[i] + 1
64    while( (i >= 1) && (comb[i] >= n - k + i)) {
65      i <- i - 1
66      comb[i] <- comb[i] + 1
67    }
68    if(comb[1] == 1)
69      return(list(comb, 0)) #exit function when no more combns to be generated
70    for(j in (i+1):(k)) {
71      if((i+1) <= k)
72        comb[j] <- comb[j-1] + 1
73    }
74    return(list(comb,1)) #return a combination
75  }
76  # get each node's group of combs according to what is in their mychunk
77  # e.g. if mychunk contains 1,2, then grab all combinations that start with a 1 and 2
78  findmycomb <- function() {
79    mychunk <- mychunk + 1
80    len <- length(mychunk) # get the number of values in mychunk
81    out <- c() # store this node's found combinations
82    myn <- c(n - mychunk+1) # store the lengths of the subsets this node gets
83    # cae[[1]] contains comb[]; cae[[2]] contains the exit value
84    for(i in 1:len) {
85      out <- c(out, x[cae[[1]]+mychunk[i]])
86      while(1) {
87        cae <- next_comb(cae[[1]], m, myn[i])
88        if(cae[[2]] == 0) # if next_combn() returns 0, exit
89          break;
90        out <- c(out, x[cae[[1]]+mychunk[i]])
91      }
92      cae<-list(c(0:(m-1)), 1) # reset comb and exit value
93      myn[i] <- myn[i] - 1
94    }
95    return(list(mychunk, out))
96  }
97
98  # using "wrap" allocation - assigning node work from front and back of input
99  setmychunk <- function() {
100   mychunk <<- c()
101   chunkNum <- 1
102   i<-myid
103   while(i < n-m+1) {
104     mychunk <<- c(mychunk, i)
105     chunkNum <- chunkNum + 1
106     if(chunkNum %% 2 == 0)
107       i <- i + 2 * (ncls - myid - 1)
108     else
109       i <- i + 2 * myid
110     i <- i + 1
111   }
112 }
```

```
113
114 mycombn <- function(cls, x, m) {
115   ncls <- length(cls) # number of nodes in cluster
116   n <- length(x) # length of array
117   comb <- c(0:(m-1)) # initialize comb
118   # if you have more nodes than there are groups of combinations to be assigned, need to
          reduce # of nodes
119   # there should be at most n-m+1 nodes, one per group of combinations
120   # reassigning will cause some intial lag at the start of program
121   if(n-m+1 < ncls) {
122     warning(paste("Argument cls has more nodes than will be used,
123                   reassigning ", n-m+1, "nodes only"))
124     cls <- makePSOCKcluster(rep("localhost", n-m+1))
125     ncls <- length(cls)
126   }
127   cae <- list(comb, 1) # stores comb and exit value (1 to continue looping; 0 to exit)
128   numGroups <- n-m+1 # total number of groups of combinations to find
129   # ship needed objects to workers
130   clusterExport(cls, c("m", "n", "x", "cae", "numGroups", "setmychunk",
131                        "ncls", "next_comb", "findmycomb"), envir=environment())
132   # set id of each node
133   setmyid <- function(i) {
134     myid <<- i
135   }
136
137   clusterApply(cls, 0:(ncls-1), setmyid)
138   clusterEvalQ(cls, setmychunk()) # split up the work evenly
139   ret_chunk <- clusterEvalQ(cls, findmycomb()) #list containing a node's groups, and
          combinations returned
140   #Reduce(c,ret_chunk)
141
142   # All of the below code, up to the end of this function, places the ret_chunks in
          order inside a vector
143
144   #calculate position to insert into output array
145   comblen <- vector() # stores lengths of each group of combinations in order
146                       #ex/ 7 combinations that start with 1, then comblen[2] = 7*m
147   comblen[1] <- 0 #first position
148   for(i in 1:ncls) {
149     grouplen <- length(unlist(ret_chunk[[i]][1])) #find number of group of combns this
            node had to generate
150     for(j in 1:grouplen) {
151       groupnum <- unlist(ret_chunk[[i]][1])[j]  #get just one value from groupnum which
              is a list
152       #calculate start index with that group number
153
154       comblen[groupnum+1] <- (nCm(n-groupnum, m-1))*m # length of array it acted on is n
            = n-groupnum+1
155     }
156   }
157
158   startpos <- comblen #stores the starting position for each group of nums
159   for(i in 1:(length(startpos)-1))
160     startpos[i+1] <- startpos[i+1] + startpos[i]
161   startpos <- startpos + 1
162
163   temp_start <- 0
164   temp_end <- 0
165   out <- vector() #contains sorted combinations
166   #store combinations in out at the right positions
167   for(i in 1:ncls) {
168     allcombns <- unlist(ret_chunk[[i]][2]) #get all combns found by a node
169     #print(allcombns)
170     grouplen <- length(unlist(ret_chunk[[i]][1])) #find number of group of combns
171     for(j in 1:grouplen) {
172       groupnum <- unlist(ret_chunk[[i]][1])[j]
173
174       #start and end for out
175       start<-startpos[groupnum]
176       end<-startpos[groupnum+1]-1
```

```
177
178       #start and end inside ret_chunk for that groupnum
179       temp_start <- temp_end + 1
180       temp_end <- comblen[groupnum+1] + temp_end
181
182       out[start:end] <- unlist(ret_chunk[[i]][2])[temp_start:temp_end]
183     }
184     temp_end <- 0
185   }
186   out
187
188 }
189
190 ##############################################################################################
191 # Helper Function
192 ##############################################################################################
193
194 # n choose m - calculates the total number of combinations for a given input
195 "nCm"<-
196   function(n, m, tol = 9.9999999999999984e-009)
197   {
198     # DATE WRITTEN: 7 June 1995 LAST REVISED: 10 July 1995
199     # AUTHOR: Scott Chasalow
200     #
201     # DESCRIPTION:
202     # Compute the binomial coefficient ("n choose m"), where n is any
203     # real number and m is any integer. Arguments n and m may be vectors;
204     # they will be replicated as necessary to have the same length.
205     #
206     # Argument tol controls rounding of results to integers. If the
207     # difference between a value and its nearest integer is less than tol,
208     # the value returned will be rounded to its nearest integer. To turn
209     # off rounding, use tol = 0. Values of tol greater than the default
210     # should be used only with great caution, unless you are certain only
211     # integer values should be returned.
212     #
213     # REFERENCE:
214     # Feller (1968) An Introduction to Probability Theory and Its
215     # Applications, Volume I, 3rd Edition, pp 50, 63.
216     #
217     len <- max(length(n), length(m))
218     out <- numeric(len)
219     n <- rep(n, length = len)
220     m <- rep(m, length = len)
221     mint <- (trunc(m) == m)
222     out[!mint] <- NA
223     out[m == 0] <- 1 # out[mint & (m < 0 | (m > 0 & n == 0))] <- 0
224     whichm <- (mint & m > 0)
225     whichn <- (n < 0)
226     which <- (whichm & whichn)
227     if(any(which)) {
228       nnow <- n[which]
229       mnow <- m[which]
230       out[which] <- ((-1)^mnow) * Recall(mnow - nnow - 1, mnow)
231     }
232     whichn <- (n > 0)
233     nint <- (trunc(n) == n)
234     which <- (whichm & whichn & !nint & n < m)
235     if(any(which)) {
236       nnow <- n[which]
237       mnow <- m[which]
238       foo <- function(j, nn, mm)
239       {
240         n <- nn[j]
241         m <- mm[j]
242         iseq <- seq(n - m + 1, n)
243         negs <- sum(iseq < 0)
244         ((-1)^negs) * exp(sum(log(abs(iseq))) - lgamma(m + 1))
245       }
246       out[which] <- unlist(lapply(seq(along = nnow), foo, nn = nnow,
```

```
                                    mm = mnow))
    }
    which <- (whichm & whichn & n >= m)
    nnow <- n[which]
    mnow <- m[which]
    out[which] <- exp(lgamma(nnow + 1) - lgamma(mnow + 1) - lgamma(nnow -
                                                                   mnow + 1))
    nna <- !is.na(out)
    outnow <- out[nna]
    rout <- round(outnow)
    smalldif <- abs(rout - outnow) < tol
    outnow[smalldif] <- rout[smalldif]
    out[nna] <- outnow
    out
  }
```

```
1  ##############################################################################
2  # R call function for the Thrust parallelization of combn() from the CRAN
3  # combinat package: http://cran.r-project.org/web/packages/combinat/index.html
4
5  # Function Arguments:
6  # x <- input vector of integers and/or characters
7  # m <- number of elements in a combination
8  # fun <- function to apply to the resulting output
9  # simplify <- if TRUE, print output as a matrix with m rows and nCm columns
10   # else <- print output as a list
11   # nCm is the total number of combinations generated
12 # ... <- parameters for fun
13
14 # Helper functions for handling characters in input vector x:
15 # is.letter <- function to check if there's a char in x
16 # asc <- convert char to ASCII decimal value
17 # chr <- convert decimal value to ASCII character
18 # nCm <- calculate the total number of combinations
19   # taken directly from R combinat package nCm.R
20   # inserted into this file saw combinat need not be installed when program is run
21 ##############################################################################
22
23 combnthrust<- function(x, m, fun = NULL, simplify = TRUE, ...)
24 {
25   require(Rcpp)
26   #require(combinat) # necessary for nCm in line 24
27   dyn.load("combnthrust.so")
28
29   # Input checks taken directly from combn source code
30   if(length(m) > 1) {
31     warning(paste("Argument m has", length(m),
32       "elements: only the first used"))
33     m <- m[1]
34   }
35   if(m < 0)
36     stop("m < 0")
37   if(m == 0)
38     return(if(simplify) vector(mode(x), 0) else list())
39   if(is.numeric(x) && length(x) == 1 && x > 0 && trunc(x) == x)
40     x <- seq(x)
41   n <- length(x)
42   if(n < m)
43     stop("n < m")
44
45   nofun <- is.null(fun)
46   count <- nCm(n, m, 0.10000000000000002)
47
48   # Error checks for the scheduling variables: sched and chunksize
49   # R handles the error when 'sched' is not a string/character vector
50
51   # If sched is provided, then sched must be static, dynamic, guided, or NULL
52   # if (!grepl('static', sched) && !grepl('dynamic', sched) && !grepl('guided', sched) &
       & !is.null(sched)) {
```

```
53   #     stop("Scheduling policy must be static, dynamic, or guided.")
54   # }
55   # # Set to default values depending on what is/are provided
56   # if (is.null(sched) && is.null(chunksize)) {
57   #   sched <- 'static'
58   #   chunksize <- 1
59   # }
60   # else if (!is.null(sched) && is.null(chunksize)) {
61   #   chunksize <- 1
62   # }
63   # else if (is.null(sched) && !is.null(chunksize)) { # if sched is provided, but chunk
       size is not
64   #   sched <- 'static'
65   #   warning("'sched' is replaced with default 'static' and 'chunksize' is overriden
       with default value.")
66   # }
67
68   # Checks if input vector x has characters
69   # If so, then convert chars to their ASCII decimal values
70   # Operate on the ASCII decimal values for the chars
71   ischarx <- match('TRUE', is.letter(x))
72   if (!is.na(ischarx)) {
73     ischarx_arr <- is.letter(x)
74     for (i in 1:length(ischarx_arr)) {
75       if (ischarx_arr[i]) {
76         if (length(asc(x[i])) == 1) {
77           x[i] <- asc(x[i])
78         }
79         else {
80           x[i] <- as.character(x[i])
81         }
82       }
83
84     }
85     x <- strtoi(x, base=10)
86   }
87
88   # Initialize output matrix
89   retmat <- matrix(0, m, count)
90   # Call the function through Rcpp
91   retmat <- .Call("combnthrust", x, m, n, count)
92
93   # Convert from ASCII decimal values back to chars if necessary
94   if (!is.na(ischarx)) {
95     for (i in 1:length(retmat)) {
96       if ((as.integer(retmat[i]) >= 97 && as.integer(retmat[i]) <= 122)
97       || (as.integer(retmat[i]) >= 65 && as.integer(retmat[i]) <= 90)) {
98         retmat[i] <- chr(retmat[i]);
99       }
100    }
101  }
102
103  # Apply provided function to the output
104  if (!is.null(fun)) {
105    apply(retmat, 2, fun(...))
106  }
107
108  # Format results
109  if (simplify) {
110    out <- retmat
111  }
112  else {
113    out <- list()
114    for (i in 1:ncol(retmat)) {
115      out <- c(out, list(c(retmat[, i])))
116    }
117  }
118  return(out)
119 }
120
```

```r
# function to check if there's a char in x
is.letter <- function(x) grepl("[[:alpha:]]", x)
# convert char to ascii decimal value
asc <- function(x) { strtoi(charToRaw(x),16L) }
# convert decimal value to ascii character
chr <- function(n) { rawToChar(as.raw(n)) }

"nCm"<-
function(n, m, tol = 9.9999999999999984e-009)
{
#  DATE WRITTEN:  7 June 1995               LAST REVISED:  10 July 1995
#  AUTHOR:  Scott Chasalow
#
#  DESCRIPTION:
#        Compute the binomial coefficient ("n choose m"),  where n is any
#        real number and m is any integer.  Arguments n and m may be vectors;
#        they will be replicated as necessary to have the same length.
#
#        Argument tol controls rounding of results to integers.  If the
#        difference between a value and its nearest integer is less than tol,
#        the value returned will be rounded to its nearest integer.  To turn
#        off rounding, use tol = 0.  Values of tol greater than the default
#        should be used only with great caution, unless you are certain only
#        integer values should be returned.
#
#  REFERENCE:
#        Feller (1968) An Introduction to Probability Theory and Its
#        Applications, Volume I, 3rd Edition, pp 50, 63.
#
  len <- max(length(n), length(m))
  out <- numeric(len)
  n <- rep(n, length = len)
  m <- rep(m, length = len)
  mint <- (trunc(m) == m)
  out[!mint] <- NA
  out[m == 0] <- 1  # out[mint & (m < 0 | (m > 0 & n == 0))] <-  0
  whichm <- (mint & m > 0)
  whichn <- (n < 0)
  which <- (whichm & whichn)
  if(any(which)) {
    nnow <- n[which]
    mnow <- m[which]
    out[which] <- ((-1)^mnow) * Recall(mnow - nnow - 1, mnow)
  }
  whichn <- (n > 0)
  nint <- (trunc(n) == n)
  which <- (whichm & whichn & !nint & n < m)
  if(any(which)) {
    nnow <- n[which]
    mnow <- m[which]
    foo <- function(j, nn, mm)
    {
      n <- nn[j]
      m <- mm[j]
      iseq <- seq(n - m + 1, n)
      negs <- sum(iseq < 0)
      ((-1)^negs) * exp(sum(log(abs(iseq))) - lgamma(m + 1))
    }
    out[which] <- unlist(lapply(seq(along = nnow), foo, nn = nnow,
      mm = mnow))
  }
  which <- (whichm & whichn & n >= m)
  nnow <- n[which]
  mnow <- m[which]
  out[which] <- exp(lgamma(nnow + 1) - lgamma(mnow + 1) - lgamma(nnow -
    mnow + 1))
  nna <- !is.na(out)
  outnow <- out[nna]
  rout <- round(outnow)
  smalldif <- abs(rout - outnow) < tol
```

```
  outnow[smalldif] <- rout[smalldif]
  out[nna] <- outnow
  out
}
```

```
/*****************************************************************************
Thrust (C++) implementation of R's combn() function from the CRAN combinat package

Called from combn-thrust.R using .Calll() through Rcpp interface
*****************************************************************************/
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/random.h>
#include <stdlib.h>
#include <thrust/transform.h>
#include <stdio.h>
#include <boost/math/special_functions/binomial.hpp>

#include <Rcpp.h>

using namespace std;
using namespace Rcpp;


struct comb {

  const thrust::device_vector<int>::iterator x;
  const thrust::device_vector<int>::iterator pos;
  const thrust::device_vector<int>::iterator retmat;
  int n;
  int m;
  int *x_arr, *position, *ret;


  comb(thrust::device_vector<int>::iterator _x_arr, thrust::device_vector<int>::iterator
       _pos, int _n, int _m, thrust::device_vector<int>::iterator _retmat):
    x(_x_arr),
    pos(_pos),
    n(_n),
    m(_m),
    retmat(_retmat)
  {
    x_arr = thrust::raw_pointer_cast(&x[0]);
    position = thrust::raw_pointer_cast(&pos[0]);
    ret = thrust::raw_pointer_cast(&retmat[0]);
  }

  __device__
  void operator()(int i)
  {
        if(i <= n - m)
        {
            find_comb(i, x_arr, m, n, position, ret);
        }
  }
  __device__
  void store(int *pos, int *output, int idx, int m, int *x, int *comb, int &outidx){
    for(int i = 0; i < m; i++){
      output[outidx++] = x[comb[i]+idx];
    }

  }

  __device__
  void find_comb(int idx, int *x, int m, int n, int *pos, int *output){
    int *comb = new int[m];
    for(int i = 0; i < m; i++){
      comb[i] = i;
    }
```

```
64      int new_n= n - idx;
65
66          int outidx = pos[idx];
67
68      store(pos, output, idx, m, x, comb, outidx);
69
70      while(true){
71        int i = m - 1;
72        ++comb[i];
73
74        while((i >= 0) && (comb[i] >= new_n - m + 1 + i)){
75          --i;
76          ++comb[i];
77        }
78        if(comb[0] == 1){
79          break;
80        }
81        for(i = i + 1; i < m; ++i){
82          comb[i] = comb[i-1] + 1;
83        }
84              store(pos, output, idx, m, x, comb, outidx);
85      }
86    }
87
88 };
89
90 RcppExport SEXP combnthrust(SEXP x_, SEXP m_, SEXP n_, SEXP nCm_, SEXP out){
91    NumericVector x(x_);
92    int m = as<int>(m_), n = as<int>(n_), nCm = as<int>(nCm_);
93    NumericMatrix retmat(m, nCm);
94
95      // keeps track of the position in output
96      int *pos = new int[n-m+1];
97
98      // Calculate combination possibilities for each element in the list that
99      // start with the element in the 0th index
100     int k = 0;
101     for(int i = 0; i < (n-m+1); i++){
102         pos[i] = boost::math::binomial_coefficient<double>(n - i - 1, m-1);
103         k++;
104     }
105     // Calcluate the position vector respective to the possiblities
106     int temp = pos[0]*m;
107     int temp2;
108     pos[0]=0;
109     for (int j=1; j<(n-m+1); j++){
110         temp2 = pos[j];
111         pos[j]=temp;
112         temp=pos[j]+(m*temp2);
113     }
114
115   thrust::device_vector<int> d_x(x.begin(), x.end());
116     thrust::device_vector<int> d_pos(pos, pos + (n-m+1));
117   thrust::device_vector<int> d_mat(retmat.begin(), retmat.end());
118
119   thrust::counting_iterator<int> begin(0);
120   thrust::counting_iterator<int> end = begin + n;
121
122   thrust::for_each(begin, end, comb(d_x.begin(), d_pos.begin(), n, m, d_mat.begin()));
123
124   thrust::copy(d_mat.begin(), d_mat.end(), retmat.begin());
125
126   return retmat;
127
128 }
```

Account of Work Done

Trisha Funtanilla: I worked on the OpenMP implementation, putting together the Rcpp interface, error checks, and output formatting for the codes, writing the OpenMP and Snow section in the report, and creating the timing comparison plots.

Syeda Inamdar: I worked with Jennifer on the Thrust implementation and integrating the Rcpp file Trisha wrote to the Thrust interface. I also worked with Jennifer to write the Thrust section in the report and conduct timing comparisons.

Eva Li: I worked on the OpenMP and Thrust implementations.

Jennifer Wong:

References

[1] Scott Chasalow *Package 'combinat'*
http://cran.r-project.org/web/packages/combinat/combinat.pdf

[2] Junior Barrera, Alfredo Goldman, and Martha Torres *A Parallel Algorithm for Enumerating Combinations* http://www.ime.usp.br/ gold/ipp03v3.pdf

[3] Norm Matloff *Building the Rth routines manually:*
http://heather.cs.ucdavis.edu/ matloff/rth.html

[4] Norm Matloff *Function Documentation*
https://thrust.github.io/doc/group__modifying.html

[5] *Generating Combinations*
https://compprog.wordpress.com/2007/10/17/generating-combinations-1/

[6] Norm Matloff *Parallel Programming with GPUs and R*
http://blog.revolutionanalytics.com/2015/01/parallel-programming-with-gpus-and-r.html