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:[?]

> “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. It accepts the same input arguments as the original function, can handle the same types of valid inputs, and through new optional arguments, also allows the user to decide on a scheduling policy and chunk size.

Code highlights:

- Load balancing algorithm [?] (next section)
- Avoidance of critical sections and barriers

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 output.

### 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. 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 the index of the previous task. This next distribution is determined by the equation:

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

The succeeding distributions alternates 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 $m$ 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, 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 the 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 timinng comparisons.

### 3.1.4 Cache and Other Performance Tuning Considerations

The following performance tuning applies only to the load balancing algorithm described in 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 `retmat`, 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

### 3.1.5 Comparative Analysis

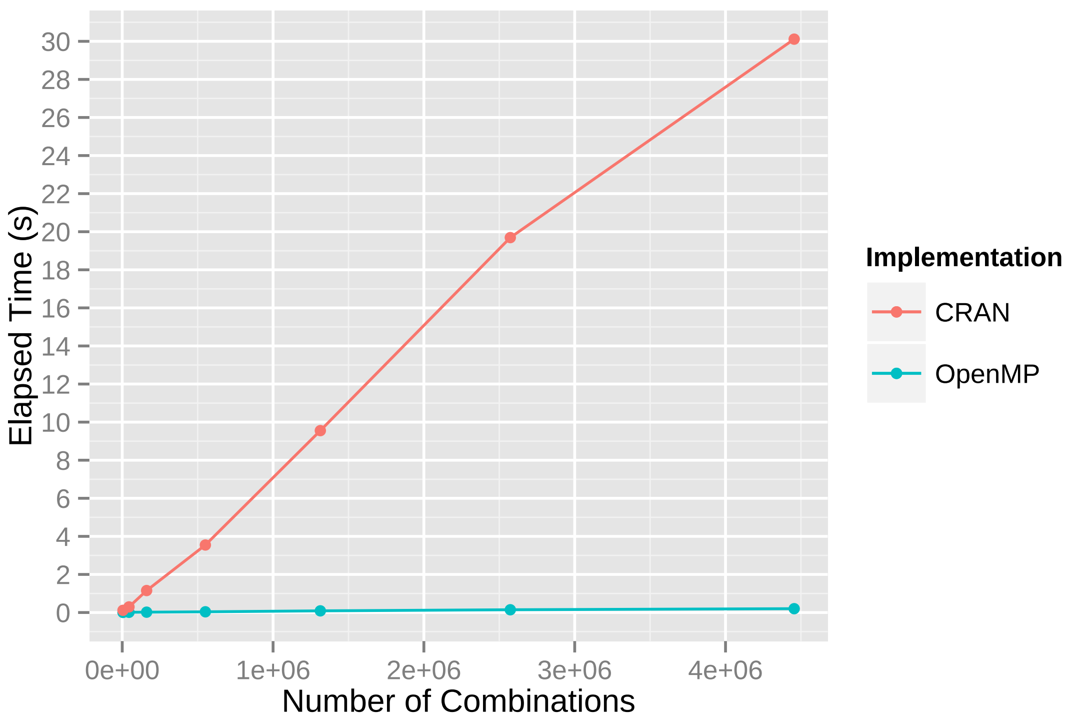
The following input sizes were used for comparing the speeds of the source code and our OpenMP implementation using 8 threads. Tests were done using `system.time(combn(sample(1:n), m))`.

```
> 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 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. `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

### 3.2.1 The Issue of Overhead

After running tests using small inputs, it was observed that parallelization using Snow more often than not took longer than running the CRAN `combn()` (sequentially). 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

In order to reduce network overhead and improve the performance, the code was written so that the nodes will do long calculations and less communications. 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.

### 3.3 Thrust

## 4 Timing Comparisons

Test Cases:

```
Test 1:  system.time(combn(sample(1:100), 2, NULL, TRUE)
Test 2:  system.time(combn(sample(1:300), 2, NULL, TRUE)
Test 3:  system.time(combn(sample(1:100), 3, NULL, TRUE)
Test 4:  system.time(combn(sample(1:150), 3, NULL, TRUE)
Test 5:  system.time(combn(sample(1:200), 3, NULL, TRUE)
Test 6:  system.time(combn(sample(1:250), 3, NULL, TRUE)
Test 7:  system.time(combn(sample(1:300), 3, NULL, TRUE)
Test 8:  system.time(combn(sample(1:120), 3, fun, FALSE)
Test 9:  system.time(combn(sample(1:260), 2, NULL, TRUE)
Test 10:  system.time(combn(sample(1:180), 2, NULL, TRUE)
Test 11:  system.time(combn(sample(1:50), 4, NULL, TRUE)
Test 12:  system.time(combn(sample(1:110), 2, fun, TRUE)
Test 13:  system.time(combn(sample(1:300), 2, NULL, FALSE)
Test 14:  system.time(combn(sample(1:130), 2, fun, FALSE)
Test 15:  system.time(combn(sample(1:420), 2, fun, TRUE)
```

```
fun <- function(x) {return(x*x)}
```

The following plot compares all four implementations:

## 5 Conclusions

# A Appendix

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

# Function Arguments:
# 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
# where nCm is the total number of combinations generated
# ... <- parameters for fun

# Helper functions for handling characters in input vector x:
# is.letter <- # function to check if there’s a char in x
# asc <- convert char to ASCII decimal value
# chr <- convert decimal value to ASCII character
# nCm <- calculating the total number of combinations
  # (taken directly from R combinat package)
  # inserted into this file since combinat could not be installed in CSIF
################################################################################

combn <- function(x, m, fun = NULL, simplify = TRUE,
  sched = NULL, chunksize = NULL, ...)
{
  require(Rcpp)
  dyn.load("combn-final.so")

  # Input checks taken directly from combn 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)

  # Error checks for the scheduling variables: sched and chunksize
  # R handles the error when ’sched’ is not a string/character vector

  # If sched is provided, then sched must be static, dynamic, guided, or NULL
  if (!grepl(’static’, sched) && !grepl(’dynamic’, sched) && !grepl(’guided’, sched) &&
      !is.null(sched)) {
      stop("Scheduling policy must be static, dynamic, or guided.")
  }
  # Set to default values depending on what is/are provided
  if (is.null(sched) && is.null(chunksize)) {
    sched <- ’static’
    chunksize <- 1
  }
  else if (!is.null(sched) && is.null(chunksize)) {
    chunksize <- 1
  }
  else if (is.null(sched) && !is.null(chunksize)) { # if sched is provided, but chunk
      size is not
    sched <- ’static’
    warning("’sched’ is replaced with default ’static’ and ’chunksize’ is overriden with
        default value.")
```

```
  }

  # Checks if input vector x has characters
  # If so, then convert chars to their ASCII decimal values
  # Operate on the ASCII decimal values for the chars
  ischarx <- match('TRUE', is.letter(x))
  if (!is.na(ischarx)) {
    ischarx_arr <- is.letter(x)
    for (i in 1:length(charx)) {
      if (ischarx_arr[i]) {
        if (length(asc(x[i])) == 1) {
          x[i] <- asc(x[i])
        }
        else {
          x[i] <- as.character(x[i])
        }
      }

    }
    x <- strtoi(x, base=10)
  }


  #Calculate positions for output
  pos <- vector()
  temp_n <- n
  for (i in 1:(n-m+1)) {
    pos <- c(pos, nCm(temp_n-i, m-1))
  }
  temp <- pos[1]
  pos[1] <- 0
  for (i in 2:length(pos)) {
    temp2 <- pos[i]
    pos[i] <- temp
    temp <- pos[i] + temp2
  }

  # Initialize output matrix
  retmat <- matrix(0, m, count)
  # Call the function through Rcpp
  retmat <- .Call("combn", x, m, n, count, sched, chunksize, pos)

  # Convert from ASCII decimal values back to chars if necessary
  if (!is.na(ischarx)) {
    for (i in 1:length(retmat)) {
      if ((as.integer(retmat[i]) >= 97 && as.integer(retmat[i]) <= 122)
      || (as.integer(retmat[i]) >= 65 && as.integer(retmat[i]) <= 90)) {
        retmat[i] <- chr(retmat[i]);
      }
    }
  }

  # Apply provided function to the output
  if (!is.null(fun)) {
    apply(retmat, 2, fun(...))
  }

  # Format results
  if (simplify) {
    out <- retmat
  }
  else {
    out <- list()
    for (i in 1:ncol(retmat)) {
      out <- c(out, list(c(retmat[, i])))
    }
  }
  return(out)
}
```

```
# 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))
    }
    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)
```

```r
  smalldif <- abs(rout - outnow) < tol
  outnow[smalldif] <- rout[smalldif]
  out[nna] <- outnow
  out
}
```

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

Called from R using .Calll() through Rcpp
********************************************************************************/

#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") {
    omp_set_schedule(omp_sched_guided, chunksize);
  }

  if (sched == "static") { // use the load balancing algorithm
    #pragma omp parallel
    {
      // this thread id, total number of threads, combination indexes array
      int me, nth, *comb;

      nth = omp_get_num_threads();
```

```
      me = omp_get_thread_num();

      // array that will hold all of the possible combinations
      // of size m of the indexes
      comb = new int[m];

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

      int chunkNum = 1; // the number of chunk that has been distributed
      int mypos; // variable for the output position

      // each thread gets assign a chunk to work on
      // each thread will have about the same number of chunks
      // to work on throughout the lifetime of the program
      for(int current_x = me; current_x < n-m+1; current_x+=1) {
        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++;
        }

        // reset comb array for the next chunk this thread will work on
        for(int i = 0; i < m; i++) {
          comb[i] = i;
        }

        chunkNum++; // increment chunkNum for the next chunk distribution
        // determine which element this thread will work on
        if (chunkNum % 2 == 0) {
          current_x = current_x + 2 * (nth - me - 1);
        }
        else {
          current_x = current_x + 2 * me;
        }
      }
    }
  }
  else { // dynamic or guided
    int mypos;
    #pragma omp parallel
    {
      int *comb = new int[m];
      for (int i = 0; i < m; ++i) {
        comb[i] = i;
      }

      #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;
}
```

# 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`