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 18, 2015

1 Objectve

2 Function

2.1 Description

2.2 Reasons for Selection

3 Parallel Implementations

3.1 OpenMP

3.1.1 Load Balancing

Initially, the program was parallelized by directingly using OpenMP's loop directive `for`. By using the `for` directive, we are allowing OpenMP to handle the distribution of the tasks to the threads. This makes it easier for the programmer; however, it is not the best way to distribute the tasks. For our program, it caused an extreme load imbalance. Upon The distribution of the task was heavily skewed such that the thread with lower id's generated far more combinations than the thread with higher id's. This is 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}.

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 Other Performance Tuning Considerations

- Cache coherency: Coherence overheads occur when two threads access data on the same cache line. This is called false sharing. In order to prevent instances of false sharing, the threads are made to work so each will access data that is not altered by any other thread.
- Synchronization overhead:
-

3.1.4 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 policies since the threads don't have to repeatedly access the task farm for work to, and idleness is still reduced because of the way the tasks are spread among the threads.

Therefore, the static method is the one compared to the original function in the next section.

3.1.5 Comparative Analysis

The following inputs were used for comparing the speeds of the source code and our OpenMP implementation:

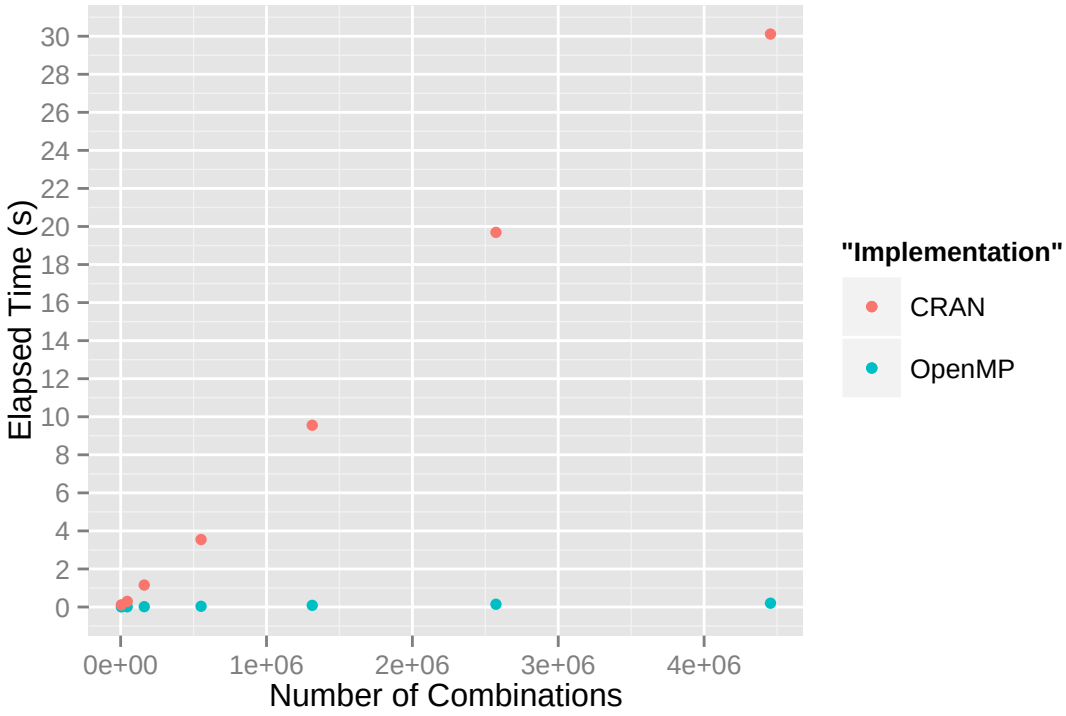
```
> system.time(combn(c(1:100), 2))
> system.time(combn(c(1:300), 2))
> system.time(combn(c(1:100), 3))
> system.time(combn(c(1:150), 3))
```

```
> system.time(combn(c(1:200), 3))
> system.time(combn(c(1:300), 3))
> system.time(combn(c(1:250), 3))
```

Here are the total number of combinations generated for each test above:

```
> 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(300, 3, 0.10000000000000002)
[1] 4455100
> nCm(250, 3, 0.10000000000000002)
[1] 2573000
```

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 is invaluable for even bigger inputs (e.g. `nCm(100, 5)`). The OpenMP implementation still manages to produce an output under 5 seconds, while the original function takes several minutes.



4 Timing Comparisons

5 Conclusions

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

```r
  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 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;
```

```
59
60        nth = omp_get_num_threads();
61        me = omp_get_thread_num();
62
63        // array that will hold all of the possible combinations
64        // of size m of the indexes
65        comb = new int[m];
66
67        // initialize comb array
68        for (int i = 0; i < m; ++i) {
69          comb[i] = i;
70        }
71
72        int temp_n = n;
73        int chunkNum = 1; // the number of chunk that has been distributed
74        int mypos; // variable for the output position
75
76        // each thread gets assign a chunk to work on
77        // each thread will have about the same number of chunks
78        // to work on throughout the lifetime of the program
79        for(int current_x = me; current_x < n-m+1; current_x+=1) {
80          int temp;
81          mypos = pos[current_x];
82          for (int i = 0; i < m; ++i) {
83            temp = comb[i] + current_x;
84            retmat(i, mypos) = x[temp];
85          }
86          mypos++;
87          while(next_comb(comb, m, temp_n-current_x))  {
88            int temp;
89            for (int i = 0; i < m; ++i) {
90              temp = comb[i] + current_x;
91              retmat(i, mypos) = x[temp];
92            }
93            mypos++;
94          }
95
96          // reset comb array for the next chunk this thread will work on
97          for(int i = 0; i < m; i++) {
98            comb[i] = i;
99          }
100
101         chunkNum++; // increment chunkNum for the next chunk distribution
102         // determine which element this thread will work on
103         if (chunkNum % 2 == 0) {
104           current_x = current_x + 2 * (nth - me - 1);
105         }
106         else {
107           current_x = current_x + 2 * me;
108         }
109       }
110     }
111   }
112   else { // dynamic or guided
113     int mypos;
114     #pragma omp parallel
115     {
116       int *comb = new int[m];
117       for (int i = 0; i < m; ++i) {
118         comb[i] = i;
119       }
120
121       int temp_n = n;
122       #pragma omp for schedule(runtime)
123       for(int current_x = 0; current_x < (n - m + 1); current_x++) {
124         int temp;
125         mypos = pos[current_x];
126         for (int i = 0; i < m; ++i) {
127           temp = comb[i] + current_x;
128           retmat(i, mypos) = x[temp];
```

```
129         }
130         mypos++;
131         while(next_comb(comb, m, temp_n-current_x))  {
132           int temp;
133           for (int i = 0; i < m; ++i) {
134               temp = comb[i] + current_x;
135               retmat(i, mypos) = x[temp];
136           }
137           mypos++;
138         }
139
140         for(int i = 0; i < m; i++) {
141           comb[i] = i;
142         }
143       }
144     }
145   }
146   return retmat;
147 }
```