Appendix: An Efficient Indexing Algorithm for Solving Large Fixed Effects Problems - Parallel Cholesky Decomposition Algorithm

Tom Balmat, Jerome P. Reiter

June 26, 2017

Following is an algorithm to compute the Cholesky decomposition, $\boldsymbol{R}$, of a symmetric, positive-definite matrix $\boldsymbol{X}$. A parallelized implementation in C, utilizing `Open-MP`, is included, along with instructions for configuration, compilation, and execution from within R.

Method

Consider a symmetric, positive-definite $p \times p$ matrix $\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & \cdots \\ x_{21} & x_{22} & x_{23} & x_{24} & \cdots \\ x_{31} & x_{32} & x_{33} & x_{34} & \cdots \\ x_{41} & x_{42} & x_{43} & x_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$ and $p \times p$ matrix

$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & \cdots \\ 0 & r_{22} & r_{23} & r_{24} & \cdots \\ 0 & 0 & r_{33} & r_{34} & \cdots \\ 0 & 0 & 0 & r_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$, such that $\boldsymbol{R}'\boldsymbol{R} = \boldsymbol{X}$.

Then $\begin{bmatrix} r_{11} & 0 & 0 & 0 & \cdots \\ r_{12} & r_{22} & 0 & 0 & \cdots \\ r_{13} & r_{23} & r_{33} & 0 & \cdots \\ r_{14} & r_{24} & r_{34} & r_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & \cdots \\ 0 & r_{22} & r_{23} & r_{24} & \cdots \\ 0 & 0 & r_{33} & r_{34} & \cdots \\ 0 & 0 & 0 & r_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & \cdots \\ x_{21} & x_{22} & x_{23} & x_{24} & \cdots \\ x_{31} & x_{32} & x_{33} & x_{34} & \cdots \\ x_{41} & x_{42} & x_{43} & x_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$

$$\Rightarrow \boldsymbol{R} = \begin{bmatrix} \sqrt{r_{11}} & \frac{x_{12}}{r_{11}} & \frac{x_{13}}{r_{11}} & \frac{x_{14}}{r_{11}} & \cdots \\ 0 & \sqrt{x_{22} - r_{12}^2} & \frac{x_{23} - r_{12}r_{13}}{r_{22}} & \frac{x_{24} - r_{12}r_{14}}{r_{22}} & \cdots \\ 0 & 0 & \sqrt{x_{33} - r_{13}^2 - r_{23}^2} & \frac{x_{34} - r_{13}r_{14} - r_{23}r_{24}}{r_{33}} & \cdots \\ 0 & 0 & 0 & \sqrt{x_{44} - r_{14}^2 - r_{24}^2 - r_{34}^2} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

A pattern of construction of $\boldsymbol{R}$, beginning at row one and continuing consecutively through row p, is apparent:

- r_{11} is a function of the supplied value x_{11}
- remaining values of row 1 are a function of supplied x_{1j} and calculated r_{11}
- the diagonal element of row i is a function of the supplied x_{ii} and prior computed values of $\boldsymbol{R}$ from rows i through i-1
- off-diagonal elements of row i, where $j > i$, are a function of the supplied x_{ij}, r_{ii}, and prior computed values of $\boldsymbol{R}$ from rows i through i-1

which gives a deceptively simple algorithm for construction of $\boldsymbol{R}$:

for $i = 1, p$

$r_{ii} = \sqrt{x_{ii} - \sum_{k=1}^{i-1} r_{ki}^2}$

for $j = i + 1, p$

$r_{ij} = \frac{x_{ii} - \sum_{k=1}^{i-1} r_{ki} r_{kj}}{r_{ii}}$

C Language Implementation

Following is the C language source for creating an R function to compute the Cholesky decomposition of a symmetric, positive-definite matrix. Objects needed for compilation are the R package `Rcpp`[1], a C compiler (`g++` from `Rtools`[2] has been tested), and the `Open-MP` library[3]. Compilation produces file `choleskyDecomp.o` (Linux) or `choleskyDecomp.dll` (MS Windows) that is loaded into R using `dyn.load()`[4]. The resulting R function is `choleskyDecomp()`, which requires a single parameter, $\boldsymbol{X}$, a (symmetric, positive-definite) numeric matrix and produces matrix $\boldsymbol{R}$, an upper triangular matrix of dimension identical to $\boldsymbol{X}$, such that $mtmRR = \boldsymbol{X}$.

Source:

```
# Cholesky decomposition
cSource <- "
#include <Rcpp.h>
#include <omp.h>
// [[Rcpp::plugins(openmp)]]
// following prefixes Rcpp type with Rcpp::
using namespace Rcpp;
// create array of p vector pointers, each addressing one matrix row
// this compensates for the following issues when the input matrix is large
// local arrays (created within function) of size greater than [2047,2047] cause
// crash on execution
// static arrays of size greater than approximately [10000,10000] cause
// compilation errors, assembler messages are returned:
// value of x too large for field of 4 bytes, possibly due to, say,
// 20,000 X 20,000 = 400,000,000 * 8 (double) = 3.2 billion, which is
// greater than unsigned long capacity (it is assumed that assembler
// instructions must be generated to address matrix positions offset
// from position 0)
// declaring a matrix with long long constant, as in static double
// R1[20000LL][20000LL]; also produces compiler 'too large' message
// it is assumed that the input matrix, X, is square, symmetric, and positive-
// definite
// [[Rcpp::export]]
NumericMatrix choleskyDecomp(NumericMatrix X) {
  // use long indices for large addressing, signed since i is decremented to -1
  long p=X.ncol();
  long i, j, k;
  double ssqRii, s;
  // create p-length arrays of pointers, each addressing one p-length matrix row
  double **R = new double *[p];
  // create array of matrix row, one pointer for each row
  for(i=0; i<p; i++) {
    R[i] = new double [p];
    for(j=0; j<p; j++)
      R[i][j]=0;
  }
  // note that, where the analytical matrix equations solve for Rij by
  // accumulating prior computed elements in order of i, the following algorithm
  // places computed values for analytical Rij in R[j][i], causing accumulation
  // accumulation along columns, taking advantage of the row-major matrix
  // storage of C (successive columns are in contiguous memory)
  // in empirical testing, this improves performance by a factor of three
  // note, also, the corresponding return of R', placing Rji in Rij
  // compute row 0, it is trivial and referenced in subsequent rows
  R[0][0]=sqrt(X(0,0));
  for(j=1; j<p; j++)
    R[j][0]=X(0,j)/R[0][0];
  // compute rows 1 through p-1
  for(i=1; i<p; i++) {
    // accumulate sum of squared Rij elements, needed for R(i+1,i+1)
    ssqRii=0;
    for(k=0; k<i; k++)
```

```
54        ssqRii+=R[i][k]*R[i][k];
55      // compute diagonal element of current row
56      R[i][i]=sqrt(X(i,i)-ssqRii);
57      // compute off diagonal elements, Rij for j>i
58      // recall R is upper diagonal by definition
59      if(i<p-1)
60        #pragma omp parallel for private(j, k, s)
61        for(j=i; j<p; j++) {
62          // accumulate prior computed Rki*Rkj products
63          s=0;
64          for(k=0; k<i; k++)
65            s+=R[i][k]*R[j][k];
66          // compute Rij using Xij and sum of prior Rki*Rkj products
67          R[j][i]=(X(i,j)-s)/R[i][i];
68        }
69    }
70    // copy R to array to be output
71    // this is necessary since R is constructed as an array of independent pointers
72    // to rows of the decomposition
73    NumericMatrix Rout(p,p);
74    std::fill(Rout.begin(), Rout.end(), 0);
75    for(i=0; i<p; i++)
76      for(j=i; j<p; j++)
77        Rout(i,j)=R[j][i];
78    // release memory allocated to dynamic rows
79    for(i=0; i<p; i++)
80      delete [] R[i];
81    // release memory allocated to arrays of row pointers
82    delete [] R;
83    return(Rout);
84  }"
```

C source code notes:

- Line 5 loads the `openmp` library for parallel processing.
- Line 24 specifies the type of value to return (a numeric R matrix), the name of the function (`cholDecomp`), and the input parameter type (a numeric R matrix).
- Lines 30 through 36 configure an array of p vector pointers, one for each row of $\boldsymbol{R}$. Due to compiler limitations and errors during execution, large, single block matrices (`double R[p][p];`) could not be declared.
- Lines 46 through 69 implement the simple algorithm from above. A few special notes:
  - Line 60 (`# pragma omp parallel for`) instructs the compiler to implement the following for loop in parallel. Index j traverses columns i+1 to p for the current row i. A block of j indices, one j for each column, is distributed to each available worker process (core, thread). The clause `private(j, k, s)` instructs the compiler to create one set of local, unshared, variables j, k, and

s for each worker process. It is important that these indices and accumulation variables not be modified from a process outside of the one for which they are declared.

- In line 4 of the simple algorithm, it is seen that k is the highest frequency iterative index. Also, k corresponds to rows of $\boldsymbol{R}$. Because C stores a matrix in row major order (columns of a given row are physically stored in contiguous memory), it is advantageous, from a memory retrieval standpoint, to traverse along columns (when the compiler detects the ability to do so, it can implement *loop unrolling*, where multiple columns are retrieved simultaneously, thus improving bus utilization and synchronization with numeric processors). Lines 64 and 65 implement the high frequency iterative loop from the simple algorithm, and it is seen that the indices are transposed from k,i and k,j to i,k and j,k, causing matrix row traversal to be executed in a more efficient column-wise manner. Of course, to maintain mathematical integrity, all operations involving $\boldsymbol{R}$ must use its transpose (where the simple algorithm calls for $\boldsymbol{R}_{ij}$, we use $\boldsymbol{R}_{ji}$). As a result, the transpose of the computed $\boldsymbol{R}$ must ultimately be returned by the function.

- Lines 72 through 77 assemble the transpose of the computed $\boldsymbol{R}$ to be returned.

Compilation Instructions (from within R)

```
# compile and create .dll
# cacheDir contains .dll and R instructions for loading and creating the function in R
library(Rcpp)
sourceCpp(code=cSource, rebuild=T, showOutput=T, cacheDir=getwd(), cleanupCacheDir=F)
```

Function Execution (from within R)

```
# X is a symmetric, positive-definite matrix
R <- choleskyDecomp(X)
```

Notes

[1]Available at (include CRAN `Rcpp` link)

[2]Available at (include `Rtools` link)

[3]Included with `Rtools`

[4]`dyn.load()` is a base R function