Efficient Solution of Large Fixed Effects Problems Using R

Appendix: Efficient Indexing of X in Composing $X'X$

Tom Balmat, Jerome P. Reiter

December 3, 2017

When a design matrix, $\boldsymbol{X}$, exhibits low operation density, we know that relatively few operations are actually necessary to produce $\boldsymbol{X'X}$, or more correctly stated, relatively few elements of $\boldsymbol{X}$ need to be operated on. The initial challenge, then, is to develop an efficient method of expanding fixed effects into a representation of design matrix indicator columns that index strictly non-zero effect levels in the design. The ideal solution to this is:

- efficient, solving in minutes, problems on the order of the example OPM federal pay disparity model, and
- general, using base R functions, making it readily portable and robust

In R, a simple function for general purpose indexing is `which()`.[1] `which()` is generally easy to implement: to generate a vector of indices, positions within a column `X` that contain a particular value a, we simply execute `which(X==a)`. By executing `which()` within `lapply()`, once for each unique non-reference level in a fixed effect column, we generate a list of p_k variable length vectors, $x_1, ..., x_{pk}$, one for each fixed effect level, where x_j contains the indices (observation row positions) corresponding to observations coded with effect level j.[2] The density of x_j is $\frac{n_j}{n}$, where n_j is the number of observations coded with level j and n is the total number of observations. The x_j vectors indicate strictly non-zero, effective matrix elements and are the sole ones to be operated on. One drawback to using the `lapply(which())` method is that `which()` is invoked independently for each of the p_k fixed effect levels, requiring a complete scan of a fixed effect column for each level. This is computationally expensive with large p_k. Additional efficiency is gained by executing `lapply()` in parallel, instructing multiple worker processes (cores) to index blocks of levels simultaneously.[3] Specifying n_c cores reduces execution time to slightly greater than $\frac{1}{n_c}$, but incurs, in a MS Windows sockets environment, additional cost in time and memory to export entire fixed effect vectors to each core. An alternative is to `order()` observation indices by fixed effect level, identify level transition positions, using `match()`, and parse

[1] See R Foundation for Statistical Computing (2017), `which()`
[2] See R Foundation for Statistical Computing (2017), `lapply()`
[3] Using `parLapply()` from the `snow` (Tierney et al., 2016) or `parallel` (R-core, 2016) packages

all indices between transition points, giving sets of non-zero observation indices by effect level.[4] For large n, the default implementation of `order()` is relatively inefficient, but its inefficiencies are compensated for by the speed of `match()`. An improvement of `order()` is to convert a fixed effect column to numeric, using `factor()`, then order the resulting indices using `sort.list(method="quick")`.[5] [6] Since all indices are needed by `order()` and `sort.list()` simultaneously, the `order()`, `sort.list()`, `match()` methods are not good candidates for parallelization. So, which method is most efficient? The answer depends on n, p_k, and n_c. Figure 1 shows simulated computation times, in minutes, to index n = 25,000,000 observations in p_k = 10, 50, 100, 250, 500, 1,000, and 5,000, 100,000, and 1,000,000 random, uniformly distributed fixed effect levels using n_c = 16 cores (for `parLapply(which())` method). Results are mean elapsed time (as reported by `proc.time()`) of ten iterations executed on a dedicated server. The clear winner is to convert a fixed effect to a factor then use `sort.list()` and `match()` to parse indices. The important lesson in this exercise is to measure computation time of critical elements of an algorithm and to test alternative methods that exploit high performance features of readily available functions. Although, for indexing, `which()` may come to mind first, it is generally not an efficient method for high dimension problems, although conversion to (numeric) factors and use of `sort.list()` and `match()` are not as intuitive, they remain base R functions and perform very well, eliminating minutes to hours from overall computation time by comparison.

Equipped with minimal sets of fixed effects observation indices, those that must be included in matrix operations, we now proceed with $\boldsymbol{X'X}$ construction. First, knowing that $\boldsymbol{X'X}$ is symmetric, we will construct the diagonal and upper triangle only, then copy the transpose of the upper triangle to the lower triangle. Ignoring (for simplicity, although the algorithm supports) interactions, there are four types of "products" that must be accommodated:

- the transpose of the constant (1) vector "multiplied" by continuous and fixed effects columns
- the transpose of a continuous column "multiplied" by a continuous column
- the transpose of a continuous column "multiplied" by a fixed effect column
- the transpose of fixed effect column "multiplied" by a fixed effect column

Multiplied is quoted to indicate that a result identical to that of standard matrix multiplication operations will be produced, but using where possible, fewer and simpler operations. Each of the four "product" types uses a distinct combination of operations:

- Row 1 (constant times continuous and fixed effects columns) consists of sums of corresponding columns. Column 1 is simply n, the total number of observations. For continuous columns, `sum(`$\boldsymbol{X}_j$`)` is used. For fixed effects columns, `length(`$\boldsymbol{X}_{k_i}$`)` is used, where $\boldsymbol{X}_{k_i}$ is the previously constructed non-zero index vector for the i^{th} level of the k^{th} fixed effect.

[4] See R Foundation for Statistical Computing (2017), `order()`, `match()`
[5] See R Foundation for Statistical Computing (2017), `factor()`
[6] `sort.list()` is a variant of `order()` (R Foundation for Statistical Computing, 2017, `sort.list()`)

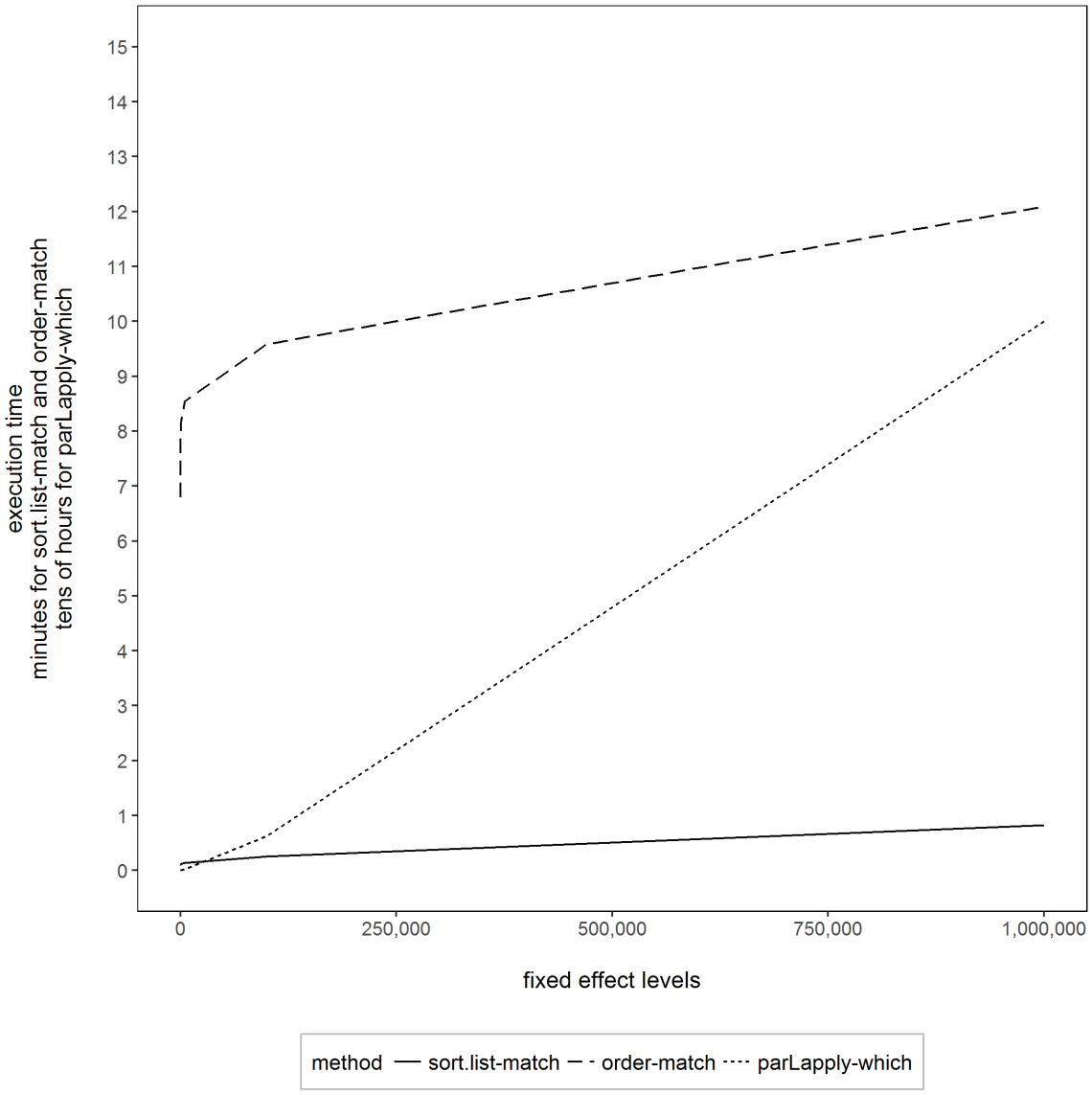

Figure 1: Three Indexing Methods: Comparison of Time to Index a Single High Dimension Fixed Effect Column. `sort.list-match` and `order-match` measured in minutes; due to lengthy execution time, `parallel-apply-which` is measured in hours divided by 10.

- Products involving two continuous columns are composed with $\texttt{sum}(\boldsymbol{X}_i * \boldsymbol{X}_j)$, which is equivalent to the standard matrix operation $\boldsymbol{X}_i'\boldsymbol{X}_j$.
- Products of continuous and fixed effects columns equate to the sum of continuous $\boldsymbol{X}_j$ elements corresponding to non-zero positions of fixed effects and use $\texttt{sum}(\boldsymbol{X}_j[\boldsymbol{X}_{k_i}])$, where $\boldsymbol{X}_{k_i}$ is the previously composed index vector for the i^{th} level of the k^{th} fixed effect.
- Products of fixed effects with fixed effects have three types:
  - On-diagonal products are simply the number of observations coded with the corresponding level (since the product of a fixed effect indicator variable by itself is the sum of n_{k_i} products of 1.0

and 1.0, where n_{k_i} is the number of observations coded with level i of fixed effect k).

- Off-diagonal products of columns from within a single fixed effect are zero, due to orthogonality.
- Off-diagonal products of columns from distinct fixed effects use the length of the intersection of corresponding index columns, equating to the count (sum of products of 1.0 and 1.0) of positions where each index is non-zero. Specifically, $(\boldsymbol{X'X})_{k1_i,k2_j} = \texttt{length(intersect(}\boldsymbol{X}_{k1_i}, \boldsymbol{X}_{k2_j}\texttt{))}$, where $\boldsymbol{X}_{k1_i}$ is the index vector for the i^{th} level of fixed effect 1 and $\boldsymbol{X}_{k2_j}$ is the index vector for the j^{th} level of fixed effect 2.

References

R-core. R Parallel Package, 2016. URL
https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf.

R Foundation for Statistical Computing. R Reference, 2017. URL
https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf.

Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. R snow Package, 2016. URL
https://cran.r-project.org/web/packages/snow/snow.pdf.