README : efficient use of full count census data files

Carl Mason

11/05/2019

The .Rmd version of this file is also available as ~/IPUMS2019-1/Documents/README_TSV-2019-1.Rmd. You can copy it into your Rstudio project/directory with the following code:

```
system('cp /ipums-repo2019-1/Documents/README_TSV-2019-
1.Rmd .')
```

Updated for full count data version provided by MPC October 2019

- 1850 version 3-2
- 1860 version 3-0
- 1870 version 3-0
- 1880 version 3-2
- 1900 version 3-3
- 1910 version 3-3
- 1920 version 3-4
- 1930 version 3-4
- 1940 version 3-3

Overview

The full count data for each census, arrives from MPC in the form of one huge .dat file and some code for reading it into STATA or SAS. The size of the .dat file, it's hierarchical structure and its lack of delimiters makes it a serious chore to read either into R or into one of those other packages for which code files are provided. This document describes how to efficiently and selectively read the Tab-separated files that have been prepared for you.

The goals of this document are:

1. To describe the organization of the data files and directories. Particularly the tab delimited files which are the easiest to work with.
2. To present vignettes of simple and efficient strategies for reading the data
3. To present two methods of attaching labels to categorical variables:

- ipumsr
- a homegrown set of functions for producing factors

If your pants are on fire, you can go ahead and fread() one of the sample500k.tsv files from one of the /ipums-repo201901/190X/TSV directories. Those files contain information from 500,000 randomly selected households organized in the IPUMS “rectangular” format with one record per *person* with the corresponding household record appended to each row. Each file contains all the probably useful variables. Below, are general instructions for efficiently creating extracts of the data. So explore the sample500k.tsv if you wish … but don’t be that dufus who never reads the instructions…

Organization

The 2019-1 repository is divided into sub directories as shown below:

```
/ipums-repo2019-1/
├── 1850
|   ├── MPC
|   |   ├── us1850c_usa.dat
|   |   ├── us1850c_usa.dat.do
|   |   ├── us1850c_usa.dat.sas
|   |   ├── us1850c_usa.dat.sps
|   |   └── us1850c_usa.yml
|   └── TSV
|       ├── H_aux.tsv
|       ├── H.tsv
|       ├── makeFactors.R
|       ├── P_aux.tsv
|       ├── P.tsv
|       └── sample500k.tsv
├── 1860
|   ├── MPC
|   |   ├── us1860c_usa.dat
|   |   ├── us1860c_usa.do
|   |   ├── us1860c_usa.sas
|   |   ├── us1860c_usa.sps
|   |   └── us1860c_usa.yml
|   └── TSV
|       ├── H_aux.tsv
|       ├── H.tsv
|       ├── makeFactors.R
|       ├── P_aux.tsv
|       ├── P.tsv
|       └── sample500k.tsv
.
.
.
├── 1940
|   ├── MPC
|   |   ├── us1940b_usa.dat
|   |   ├── us1940b_usa.do
|   |   ├── us1940b_usa.sas
|   |   ├── us1940b_usa.sps
|   |   └── us1940b_usa.yml
|   └── TSV
|       ├── H_aux.tsv
|       ├── H.tsv
|       ├── makeFactors.R
|       ├── P_aux.tsv
|       └── P.tsv
├── fullcount.ddi.xml
├── python
|   ├── do2factor.py
|   └── parse2tsv.py
```

Each census year has a directory called MPC and another called TSV the former contain the raw files as received from the Minnesota Population Center and the latter contains tab separated files of the data. In addition there is a directory called `python` which holds the python code used to produce the TSV directories. And there is a file called fullcount.ddi.xml for use with the ipumsr package.

To make life easier, I suggest creating a symlink in your home directory> In a terminal window running on the securelpums server (or a shell that you can launch under the Tools menu in Rstudio). If your account was just created, the link is already created for you.

```
ln -s /ipums-repo2019-1 ~/IPUMS2019-1
```

I'll assume that you have done that in what follows.

Original files from MPC

The new versions of the raw data now live in a directory called `~/IPUMS2019-1/19x0/MPC` where 19x0 is the census year which of course could be 1**8**50. These directories include the revised "harmonized" ipums files for 1850-1940 except for 1890.

For each decade we received

- us19XXd_usa.dat - the raw data
- us19XXd_usa.do - stata code
- us19XXd_usa.sas - sas code
- us19XXd_usa.sps - spss code
- us19XXd_usa.yml - YAML file describing the data

For various reason noted above, it sucks to try to read these files directly with the code provided by MPC. To deal with this I have written python programs which re-write the .dat as four separate TAB delimited files.

Tab separated files (~/IPUMS2019-1/19x0/TSV)

The tsv files include every byte from the original (.dat) data files except that tab characters in the original files have been converted to '_' and of course TABs have been inserted between fields. Column headers have also been added in the way that is expected from tab separated files.

For each census, there are four .tsv files: two containing the household records, H.tsv and H_aux.tsv, the other two containing the person records, P.tsv and P_aux.tsv. In all cases the first row contains variable names as given in the .yml file. **So a simple fread() with all the defaults works well to read in each of the files**. But please don't test this without either an nrows= or select= argument to limit the number of rows read and/or the columns read respectively.

The _aux files contain variables with names like "US1930D_00123". These appear to hold intermediate source information used in constructing the variables with more traditional, meaningful names. I am guessing the few will need these variables but if you are one of the few then note that the H_aux.tsv file has the "SERIAL" column which matches with that same column in the H.tsv file. For the P.tsv and P_aux.tsv, same deal – but there are two columns: SERIALP and PERNUM.

Since in the tsv version of the data, the Household and Person records are in separate files – instead of the usual IPUMS format where the household records are copied onto the end of each corresponding person record, joining the records is necessary if you wish to work with both person and household level information. Fortunately the data.tables package makes the joining process remarkably easy and **fast**. Examples are given below

Ancestry.com files

The “Ancestry” files (from Ancestry.com) on which the harmonized ipums files are based are available in the /ancestry directory (or if your account is new ~/ANCESTRY). They are organized by census decade. This is different from how these files were arranged in /ipums-repo2019, but the data are the same. We do not anticipate any updates to the Ancestry files.

Extracting a useful subset of the data

Far and away the best way to read these into R is with data.table::fread(). data.table is very useful package which will let you do pretty much everything that one does in R – only much much faster. The catch is that you have to learn a new set of commands and new way of doing everything. A reasonable path forward is to learn just enough of data.table to read files, and to subset and merge datatables. You’ll almost be able to figure that out from the code snippets below. Google can help you find other resources.

Some preliminaries:

```
library(data.table)
# make a link so we can type shorter file names  -- you probably did this already
#system("ln -s /ipums-repo2019-1 ~/IPUMS2019-1")
## where's the data
basedir="~/IPUMS2019-1/1900/TSV"


# snag column names from P.tsv and H.tsv
Pnames<-names(fread(paste0(basedir,"/P.tsv"),sep="\t",nrows=1)) # get columnames
print("P record variable names")
```

```
## [1] "P record variable names"
```

```
sort(Pnames)
```

```
##   [1] "AGE"          "AGEDIFF"      "AGEMONTH"     "BIRTHMO"
##   [5] "BIRTHYR"      "BPL"          "BPLSTR"       "CHBORN"
##   [9] "CHSURV"       "CITIZEN"      "DURMARR"      "EDSCOR50"
##  [13] "ELDCH"        "ERSCOR50"     "FAMSIZE"      "FAMUNIT"
##  [17] "FBPL"         "FBPLSTR"      "HISPAN"       "HISPRULE"
##  [21] "HISTID"       "IMAGEID"      "IND1950"      "ISRELATE"
##  [25] "LABFORCE"     "LIT"          "MARST"        "MBPL"
##  [29] "MBPLSTR"      "MOMLOC"       "MOMRULE_HIST" "MOUNEMP"
##  [33] "NAMEFRST"     "NAMELAST"     "NATIVITY"     "NCHILD"
##  [37] "NCHLT5"       "NSIBS"        "OCC1950"      "OCCSCORE"
##  [41] "OCCSTR"       "PERNUM"       "PERWT"        "PERWTREG"
##  [45] "POPLOC"       "POPRULE_HIST" "PRESGL"       "QAGE"
##  [49] "QAGEMONT"     "QBIRTHMO"     "QBPL"         "QCHBORN"
##  [53] "QCHSURV"      "QCITIZEN"     "QDURMARR"     "QFBPL"
##  [57] "QIND"         "QLIT"         "QMARST"       "QMBPL"
##  [61] "QOCC"         "QQTRUNEM"     "QRACE"        "QRELATE"
##  [65] "QSCHOOL"      "QSEX"         "QSURSIM"      "QTRUNEMP"
##  [69] "QYRIMM"       "RACAMIND"     "RACASIAN"     "RACBLK"
##  [73] "RACE"         "RACESING"     "RACOTHER"     "RACPACIS"
##  [77] "RACWHT"       "RECTYPEP"     "RELATE"       "RELSTR"
##  [81] "SAMPLEP"      "SCHLMNTH"     "SCHOOL"       "SEI"
##  [85] "SERIALP"      "SEX"          "SFRELATE"     "SFTYPE"
##  [89] "SLWT"         "SPEAKENG"     "SPLOC"        "SPRULE_HIST"
##  [93] "STEPMOM"      "STEPPOP"      "SUBFAM"       "SURSIM"
##  [97] "YEARP"        "YNGCH"        "YRIMMIG"      "YRSUSA1"
## [101] "YRSUSA2"
```

```
Hnames<-names(fread(paste0(basedir,"/H.tsv"),sep="\t",nrows=1)) # get columnames
print("H record variable names")
```

```
## [1] "H record variable names"
```

```
sort(Hnames)
```

```
##  [1] "APPAL"     "CITY"      "CITYPOP"   "CNTRY"     "COUNTYICP"
##  [6] "DWELLING"  "ENUMDIST"  "FARM"      "GQ"        "GQFUNDS"
## [11] "GQSTR"     "GQTYPE"    "HEADLOC"   "HHTYPE"    "HHWT"
## [16] "HHWTREG"   "LINE"      "MDSTATUS"  "METAREA"   "METDIST"
## [21] "METRO"     "MORTGAGE"  "MULTGEN"   "NCOUPLES"  "NENGPOP"
## [26] "NFAMS"     "NFATHERS"  "NMOTHERS"  "NSUBFAM"   "NUMPERHH"
## [31] "NUMPREC"   "OWNERSHP"  "PAGENO"    "QFARM"     "QGQFUNDS"
## [36] "QGQTYPE"   "QMORTGAG"  "QOWNERSH"  "RECTYPE"   "REEL"
## [41] "REGION"    "SAMPLE"    "SEA"       "SERIAL"    "SIZEPL"
## [46] "SPLIT"     "SPLITHID"  "SPLITNUM"  "STATEFIP"  "STATEICP"
## [51] "STCOUNTY"  "STDCITY"   "STREET"    "SUBSAMP"   "URBAN"
## [56] "URBAREA"   "URBPOP"    "YEAR"
```

```
# We'll choose from these variables those which we want
```

Extracting all California residents

Let's extract all of the records of California household and append them to the person records of the people who live in them. There are three steps to this:

1. Reading in a subset of columns of the household records and dropping all those records which are not from California
2. Reading in a subset of variables but all person records
3. Doing an inner join of the person records onto the (California) household records.
4. cleanup and garbage collection #### Household records

```
# Calfornia is statefip 6; other states go by other numbers
state=6
# Specify a subset of columns to read:
Hvars=c("SERIAL","STATEFIP","COUNTYICP",Hnames[c(5,7,13)]);Hvars
```

```
## [1] "SERIAL"    "STATEFIP"  "COUNTYICP" "NUMPREC"   "HHWTREG"   "METAREA"
```

```
# use data.table::fread() to snork up a tremendous amount of data very fast

# qupte="" tells fread that there are no tabs within fields.
H <- fread(paste0(basedir,"/H.tsv"),sep="\t",select=Hvars,quote="",showProgress = FALSE)
```

```
# use data.table syntax to create a new object that holds only the CA records
Hca<-H[STATEFIP == state]
# drop the original H data set that we read in an no longer need
rm(H)
# ensure that the RAM used to store H is returned to the OS so that other
# scientists can use it too
gc()
```

```
##           used (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells  605513 32.4    1128982  60.3  1128982  60.3
## Vcells 7419594 56.7   72620149 554.1 84372625 643.8
```

Notice a couple of important things in the above code.

- Hvars is a vector of strings that hold the names of the variables – these are in the first row of the H.tsv. It is cool that fread understands them. Note that **SERIAL is critical** it is the unique identifier for household records. We'll need it to merge H and P records later.
- H[STATEFIP == state] looks wrong and IS wrong in base R. data.table understands it to mean select the rows in which the STATEFIP column is equal to 6. In base R you write something like Hca<-H[H$TATEFIP==state,].
- creating the new datatable, Hca, is a better practice than to simply overwriting H with the subset of H. Experience has taught us that **R does not relinquish RAM when we overwrite a large object with a smaller one** R gives back that RAM only when a variable is explicitly erased (rm()'ed). Calling gc() after the

rm() is a kindness. gc() stands for “garbage collection” calling gc() forces R to return the unneeded RAM to the OS right away – which is better than … whenever.

Person records

```
## select some columns of interest -- which must include "SERIALP" in order to match wit
h SERIAL in the H records

Pvars=c("PERNUM","SERIALP","AGE","SEX","BPL","NAMELAST","NAMEFRST",Pnames[c(73,19,6)]);P
vars
```

```
##  [1] "PERNUM"   "SERIALP"  "AGE"      "SEX"      "BPL"      "NAMELAST"
##  [7] "NAMEFRST" "EDSCOR50" "ELDCH"    "PERWTREG"
```

```
# read in person recrods. this takes time -- but the fewer the columns the faster it goe
s.
P <- fread(paste0(basedir,"/P.tsv"),sep="\t", select = Pvars,quote="",showProgress = FAL
SE)
# perform inner join to match all person recs where SERIALP is among the SERIAL values
# of California Housholds
```

Notice that P.tsv is much larger than H.tsv and therefore even with the magic of fread, it’s going to take 4 minutes and 35 seconds to read. You might see some “Status code 504” pop ups if you doing this in Rstudio. That’s Rstudio being impatient with R. Click OK and try to deal with your anxiety.

The `quote=""` tells fread that there are no records that include a TAB character that is that all TAB characters are delimiters. Without this argument, fread gets confused by a couple of missing quotation marks in the data.

The inner join

```
## data.table requires that we set the primary "key" of each datatable which it will use
in the inner join
setkey(Hca,SERIAL)
setkey(P,SERIALP)
## this is the join command ridiculously simple and fast
California<-P[Hca,nomatch=0]
## remove and garbage colled the un needed bits
rm(P)
rm(Hca)
gc()
```

```
##            used  (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   862786  46.1    4425780  236.4   4771811  254.9
## Vcells 69024197 526.7  590469465 4505.0 575964974 4394.3
```

The interesting bit in the chunk above is the inner join of P on Hca. The result of this command is a datatable with number of rows equal to the number of records in P with a value of SERIALP equal to one of the values of SERIAL in Hca. In other words there is one record in `California` for each person observed to reside in a household in

California. Each record in `California` includes all of the variables specified in Var as well as all of the variables specified in Hvar. Since the values of variables in Hca refer to the household, all members of the same household will have identical values in the Hvar columns in California. Hca records are just repeated.

https://rstudio-pubs-static.s3.amazonaws.com/52230_5ae0d25125b544caab32f75f0360e775.html (https://rstudio-pubs-static.s3.amazonaws.com/52230_5ae0d25125b544caab32f75f0360e775.html)

California is the standard IPUMS “rectangular” format that we all know and love from ipums.org.

Attaching labels to variables

Although many of the IPUMS variables are categorical, in the MPC and the TSV files everything is stored a number. This is not a problem: you can just remember that “1” means Male and “2” means Female. It becomes a little inconvenient, however, to have to remember that “46000”, “46100” and “46200” mean Estonia, Latvia and Lithuania respectively. So with some variables, you are *probably* going to want to use the character string to which the integer refers.

There are (at least) two easy ways of doing this which are outlined below.

makeFactors.R – homebuilt factor code

The old but not entirely loved way of dealing with categorical variables in R is with “factors”. Google will explain what that means if you are not already familiar with it – if you are familiar, Google will tell you why you should be unhappy about it but also why you cannot escape using them.

In each TSV directory is a file called `makeFactor.R` which was produced by the `/ipums-repo2019-1/python/do2factors.py` program from information found in the .do file in the MPC sub directory.

To use makeFactors.R, simply source it and call the appropriate function

```
#This will create a bunch of functions with names like <tt>BPL_F</tt> which can be used
 as follows
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse
1.2.1 ──
```

```
## ✔ ggplot2 3.2.1     ✔ purrr   0.3.3
## ✔ tibble  2.1.3     ✔ dplyr   0.8.3
## ✔ tidyr   1.0.0     ✔ stringr 1.4.0
## ✔ readr   1.3.1     ✔ forcats 0.4.0
```

```
## ── Conflicts ────────────────────────────────────────── tidyverse_confli
cts() ──
## ✖ dplyr::between()   masks data.table::between()
## ✖ dplyr::filter()    masks stats::filter()
## ✖ dplyr::first()     masks data.table::first()
## ✖ dplyr::lag()       masks stats::lag()
## ✖ dplyr::last()      masks data.table::last()
## ✖ purrr::transpose() masks data.table::transpose()
```

```
source("~/IPUMS2019-1/1900/TSV/makeFactors.R")

California$bpl <-BPL_F(California$BPL)
California$sex <-SEX_F(California$SEX)
California %>% group_by(bpl,sex) %>% summarise(N=n()) %>%
  arrange(desc(N)) %>% filter(N>15000)
```

```
## # A tibble: 17 x 3
## # Groups:   bpl [10]
##    bpl        sex         N
##    <fct>      <fct>   <int>
##  1 California Female 337248
##  2 California Male   335728
##  3 China      Male    38774
##  4 Germany    Male    38079
##  5 New York   Male    30956
##  6 Ireland    Male    28327
##  7 Ireland    Female  26871
##  8 Germany    Female  24956
##  9 New York   Female  24461
## 10 Illinois   Male    22283
## 11 England    Male    21621
## 12 Illinois   Female  20611
## 13 Ohio       Male    19652
## 14 Missouri   Male    18572
## 15 Missouri   Female  16557
## 16 Italy      Male    16359
## 17 Ohio       Female  15794
```

The _F functions in makeFactors.R take two additional optional arguments: + strings=FALSE if you want a vector of character strings *instead* of a factor set strings=TRUE + na2na=TRUE if set to false “NA” will be treated as a valid category represented by the string “NA”, if set to TRUE (the default), the “NA” will be interpreted as missing values and will be treated as NA’s. **which is probably what you want**

ipumsr

Ipumsr is a relatively new package so how it works might change by the time you read this–Google will explain.

To use ipumsr we first need to read a ddi file. Unfortunately, the secure full count data does not include such a file. Consequently, I have thoughtfully downloaded one for you that is based on a dataset that includes ALL of the full count data sets available at ipums.org. That is, I created a dataset on the ipums.org website that includes all variables and all public full count samples (but no observations). The ddi files from that exercise is called:

```
/ipums-repo2019-1/fullcount.ddi.xml
```

It seems to work pretty well most of the time but there are a few GOTCHA’s.

```
##Birthplace by sex as the data come to us -- no labels
California %>% group_by(BPL,SEX) %>% summarise(N=n()) %>% arrange(desc(N)) %>% filter(N>
15000)
```

```
## # A tibble: 17 x 3
## # Groups:   BPL [10]
##      BPL   SEX      N
##    <int> <int>  <int>
##  1   600     2 337248
##  2   600     1 335728
##  3 50000     1  38774
##  4 45300     1  38079
##  5  3600     1  30956
##  6 41400     1  28327
##  7 41400     2  26871
##  8 45300     2  24956
##  9  3600     2  24461
## 10  1700     1  22283
## 11 41000     1  21621
## 12  1700     2  20611
## 13  3900     1  19652
## 14  2900     1  18572
## 15  2900     2  16557
## 16 43400     1  16359
## 17  3900     2  15794
```

```
## adding labels with ipumsr
library(ipumsr)
```

```
## Registered S3 methods overwritten by 'ipumsr':
##   method                                  from
##   format.pillar_shaft_haven_labelled_chr haven
##   format.pillar_shaft_haven_labelled_num haven
##   pillar_shaft.haven_labelled            haven
```

```
# read the ddi object
ddi <- read_ipums_ddi("/ipums-repo2019-1/fullcount.ddi.xml")

## GOTCHA!  'BPL' in our data set is coded as BPLD (the "detailed" version)
## changing the variable's name to BPLD *before* doing the ipums_collect is a workaroun
d.
California$BPLD<-California$BPL
California<-ipums_collect(California,ddi)

## no problem with SEX though


California %>% mutate(
  bplf= as_factor(BPLD),
  sex=as_factor(SEX)
) %>%
  group_by(bplf,sex) %>% summarise(N=n()) %>% arrange(desc(N)) %>% filter(N>15000)
```

```
## # A tibble: 17 x 3
## # Groups:   bplf [10]
##    bplf       sex         N
##    <fct>      <fct>   <int>
##  1 California Female 337248
##  2 California Male   335728
##  3 China      Male    38774
##  4 Germany    Male    38079
##  5 New York   Male    30956
##  6 Ireland    Male    28327
##  7 Ireland    Female  26871
##  8 Germany    Female  24956
##  9 New York   Female  24461
## 10 Illinois   Male    22283
## 11 England    Male    21621
## 12 Illinois   Female  20611
## 13 Ohio       Male    19652
## 14 Missouri   Male    18572
## 15 Missouri   Female  16557
## 16 Italy      Male    16359
## 17 Ohio       Female  15794
```

The key message from the above cell is that ipumsr is easy to use BUT there is something amiss with the labels of the BPL variable (and perhaps others?) – the work around for BPL which is shown.

Aside from that, you just need to :

1. Read the ddi.xml file into an object
2. use ipums_collect() to link the ddi object and the data.frame, tibble, data.table or whatever object you like to use to hold data.
3. use as_factor() to create factors.

Note that ipumsr does not do by default recognize “N/A” as NA

```
## using ipumsr as_factor "N/A" is a valid observation
sum(as_factor(California$EDSCOR50)=="N/A")
```

```
## [1] 853668
```

```
sum(is.na(as_factor(California$EDSCOR50)))
```

```
## [1] 0
```

```
## using Carl's home made functions, "N/A" == NA
sum(is.na(EDSCOR50_F(California$EDSCOR50)))
```

```
## [1] 853668
```

But just to make life really confusing, not all N/A's are coded the same

```
sum(levels(as_factor(California$BPLD)) == "Missing/blank")
```

```
## [1] 1
```

Parallel processing

The securelpums server has 24 processors (generally), but your R process uses just one processor at a time unless you avail yourself of parallel processing. It is not always worth the trouble to do this – in fact, it can be counterproductive. If your code is going to finish in 2 hours on a single processor, it might not save you any time to figure this thing out. If your code is slow because it's always reading in data – parallel processing isn't going to help. The limit there is the speed of the disks. Parallel processing is particularly helpful bootstrapping and simulations where you find yourself doing the same damn thing over and over again.

There are a couple of ways of doing PP but here is one:

Suppose we wish to find the most common first name by birth place in California. There a lots of ways of doing this that would be better faster and cheaper than using parallel processing for example

```
getmode <- function(v) {
   uniqv <- unique(v)
   uniqv[which.max(tabulate(match(v, uniqv)))]
}
#fnameByCnty<-California[,getmode(NAMEFRST),.(COUNTYICP,SEX)]
#fnameByCnty[1:5]
fnameBybpl<-California[,getmode(NAMEFRST),.(bpl,SEX)]
fnameBybpl[202:212]
```

```
##                     bpl SEX        V1
##  1:              Poland   2     SARAH
##  2:               India   2    MARY A
##  3:              Poland   1    JOSEPH
##  4:        Europe, n.s.   1   MICHAEL
##  5:        Europe, n.s.   2      MARY
##  6:             Florida   2    MARY E
##  7:             Romania   2      LENA
##  8: Africa, n.s./n.e.c.   1     JAMES
##  9:              Azores   1    MANUEL
## 10:     Prussia, n.e.c.   1 RICHARD Z
## 11:    Dakota Territory   1     FRANK
```

But if we were to (foolishly) use PP here is the drill:

```
library(doParallel)
```

```
## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
cl <- makeCluster(4)

registerDoParallel(cl)

fnameBybpl<-foreach(bpli=sort(unique(California$bpl)), .export="California") %dopar% {
    library(data.table)
    library(tidyverse)
    res<-California[bpl==bpli,getmode(NAMEFRST),sex]
    ## need to capture bpli in result
    res %>% mutate(bpl=bpli)
}
stopCluster(cl)
rm(cl)


bind_rows(fnameBybpl)[100:110,]
```

```
##        sex       V1                   bpl
## 100   Male    PETER               Denmark
## 101 Female     MARY               Denmark
## 102   Male  WILLIAM   District of Columbia
## 103 Female     MARY   District of Columbia
## 104   Male DIETRICH     Dominican Republic
## 105 Female     MARY            East Indies
## 106   Male  CHARLES            East Indies
## 107   Male  LEOPOLD                Ecuador
## 108 Female     DORA                Ecuador
## 109   Male JOSEPH R Egypt/United Arab Rep.
## 110 Female   ESTHER Egypt/United Arab Rep.
```

Some interesting things to note about the code above: * makeCluster(4) creates a “cluster” object using 4 CPU cores. The right number of cores to use depends on what you’re going to do. If you are using big data sets then each core is going to consume a pant load of RAM and that can lead to embarrassing lapses in the progress of human knowledge. Estimate the size of the data used inside the core and multiply that by the number of cores +1 to get an estimate of what you’ll use. Run top in a terminal window to see how much your processes actually consume.

- foreach() %dopar% is a loop structure and R is not generally good at loops. If it is possible to “vectorize” your code, you should do that. Using something clever like data.table to effectively (but not actually) loop through your data will generally work even better. You can use data.tables with parallel processing of course.
- the .export argument of foreach() is important. The code block running in the foreach() block does not have access to your entire workspace. In the above code, if we did not include the .export=“California” in our foreach(), California would not be accessible and the code would fail.
- library(data.table) is also required within the foreach code block in order for the parallel processes to have access to it.
- the stopCluster(cl) command is important. If you don’t stop the cluster when you are finished, the resources (RAM) that you have used will not be returned to the operating system. It is very cheap to rm() the cluster when you are finished as well.

Running BATCH jobs why and how

Once you know that your code will run, you can store it in a file and run it as a “BATCH” job. This has the advantage of not screwing up Rstudio and even allowing you to start working on the next thing while your job runs. In addition to the annoying 504 warnings, Rstudio can be very stubborn about reloading the environment after a longish pause. When Rstudio determines that you have been idle long enough, it writes your session to disk and goes to sleep. When you come back the wake up process can be interminable if there were big objects in your workspace. That of course is going to happen if you torch of a multi-hour parallel processing job.

Running in BATCH is much cleaner – once the code is solid. You don’t use Rstudio at all for BATCH jobs – which means that if you’re an ambitious young scholar, you can develop code in Rstudio *while* your big job is running in BATCH. When the batch job finishes, you can check out what happened by reading the .Rout file where the console output is stored.

A well crafted BATCH job will almost certainly write something to a file. Otherwise, when the job ends, R exits and workspace is gone. The save() and load() functions are handy ways to store results in R's native format. They read and write pretty fast and you can recover objects with all their attributes some of which might not fit well into a .csv file.

To launch a job in BATCH mode:

1. Secure a terminal running on secureIpums – noMachine is your friend here.
2. nohup nice R CMD BATCH –no-restore –no-save path-to-program.R &
3. tail -f path-to-program.Rout

nohup in the above instructs the shell to not "hang up" even if the terminal is closed. The "nice" command reduces the priority of your job –which is a nice thing to do. The & puts the program in the background and tail -f path-to-program.Rout displays the output of your program (the .Rout file) as it is being produced.