Introduction to Topics

Advanced Statistical Programming Camp
Jonathan Olmsted (Q-APS)

Day 1: May 27th, 2014
AM Session

Administrative Issues

- Materials posted at `blackboard.princeton.edu`.
  - Set up instructions to create **Nobel** and **Adroit** accounts and basic setup for interacting with **Nobel** and **Adroit**.
  - Syllabus, handouts, slides, datasets.
  - Everyone should be pre-enrolled.

- Q&A at `https://piazza.com/princeton/summer2014/aspc/home`.

Schedule

- Location:
  - Robertson Bowl 02
  - except Friday PM, Corwin 127
- Morning session and Afternoon session: Tue Wed Thur Fri
  - AM session: 9:30am – 10:30am
  - PM session: 1:30pm – 2:30pm
  - Office Hour session: 3:30pm – 4:30pm
- Extended Morning session: Sat
  - 10:00am – 12:00pm (with lunch)

Code Distribution

- Most R and C++ code will be distributed "in-text" for handouts and slides.
- This makes it easy to get all of the code you need by downloading one file.
- Some chunks of code don't copy perfectly from the PDF (e.g., multiple columns). Code transcripts will be posted too.

Interactively:

```
> for (i in 1:4) {
+     print(i ^ 2)
+ }

[1] 1
[1] 4
[1] 9
[1] 16
```

In the slides:

```
for (i in 1:4) {
    print(i ^ 2)
}

## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

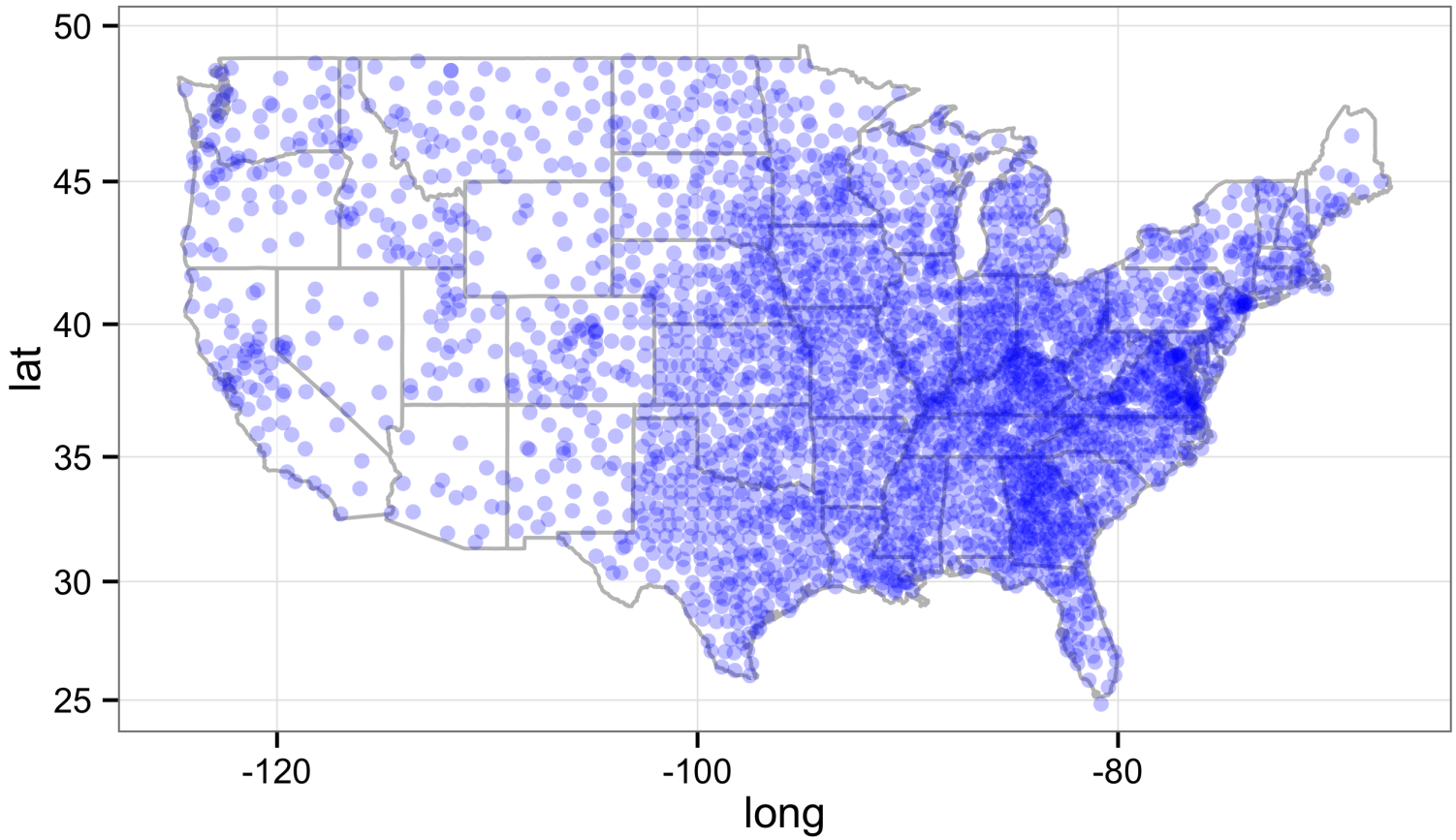
Pairwise Calculations from Spatial data

Examples:

- Geo-coded event data: insurgent attacks, natural disasters
- Geo-coded administrative data: grant programs, voter files

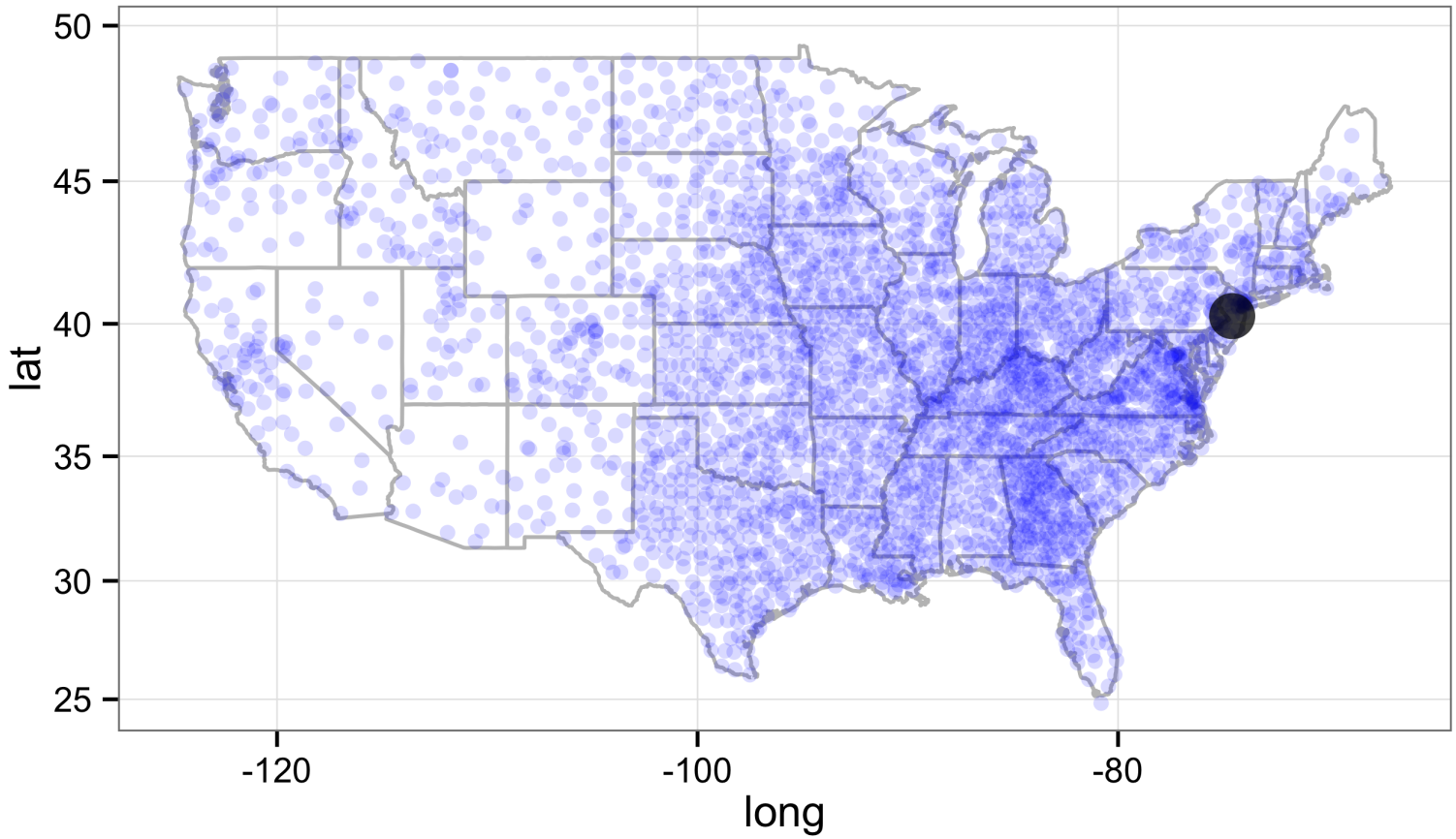
- Some pairwise "distances" are simple to calculate from network-like spatial data
- But, coordinate-based distance metrics have no simple linear algebraic representation $\rightarrow$ many individual pair-wise calculations

US Counties



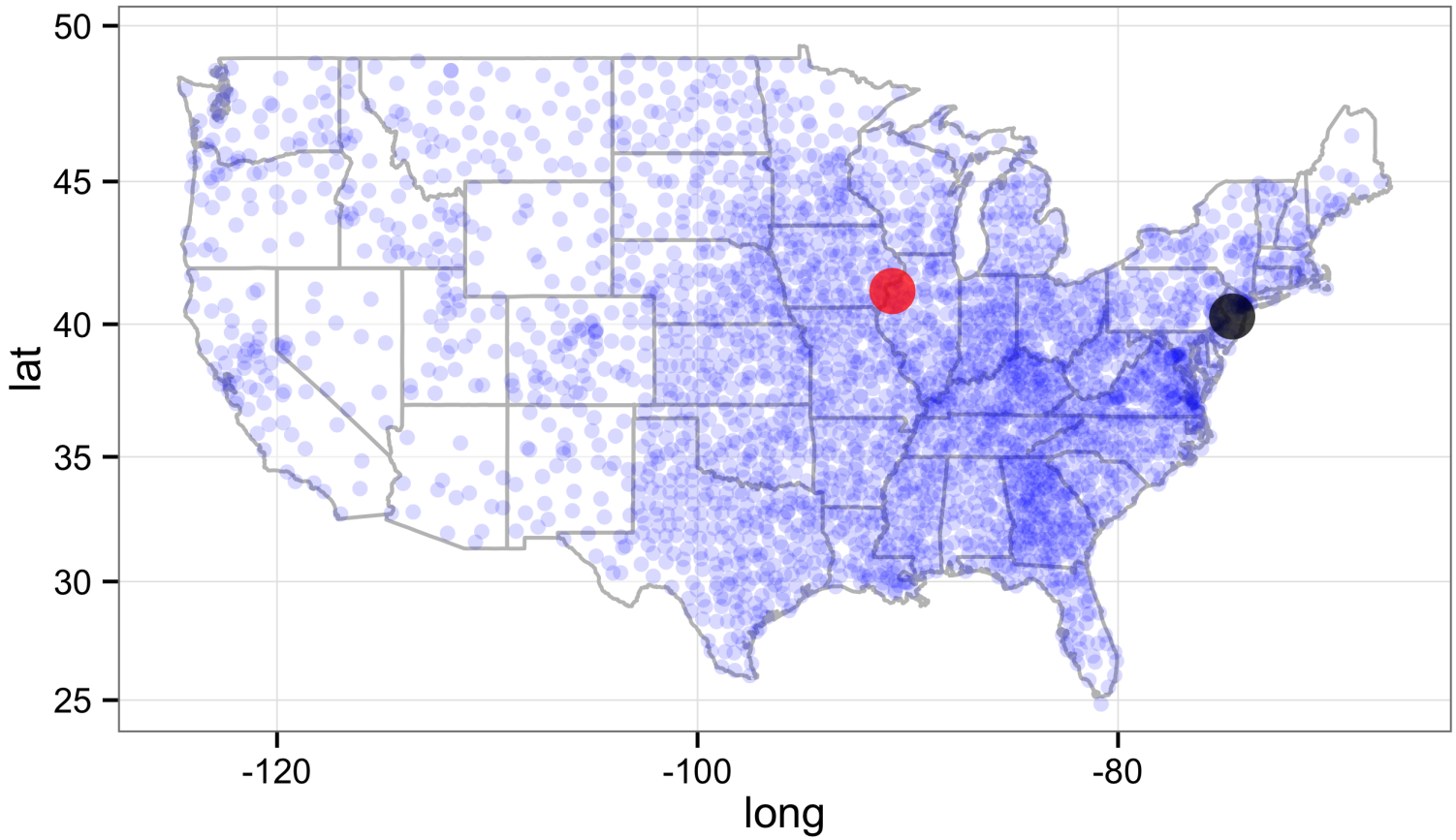
3,109 Counties in the United States — ignoring Puerto Rico, Hawaii, Alaska, American Somoa, the Virgin Islands, etc.

Pairwise Distances between US Counties



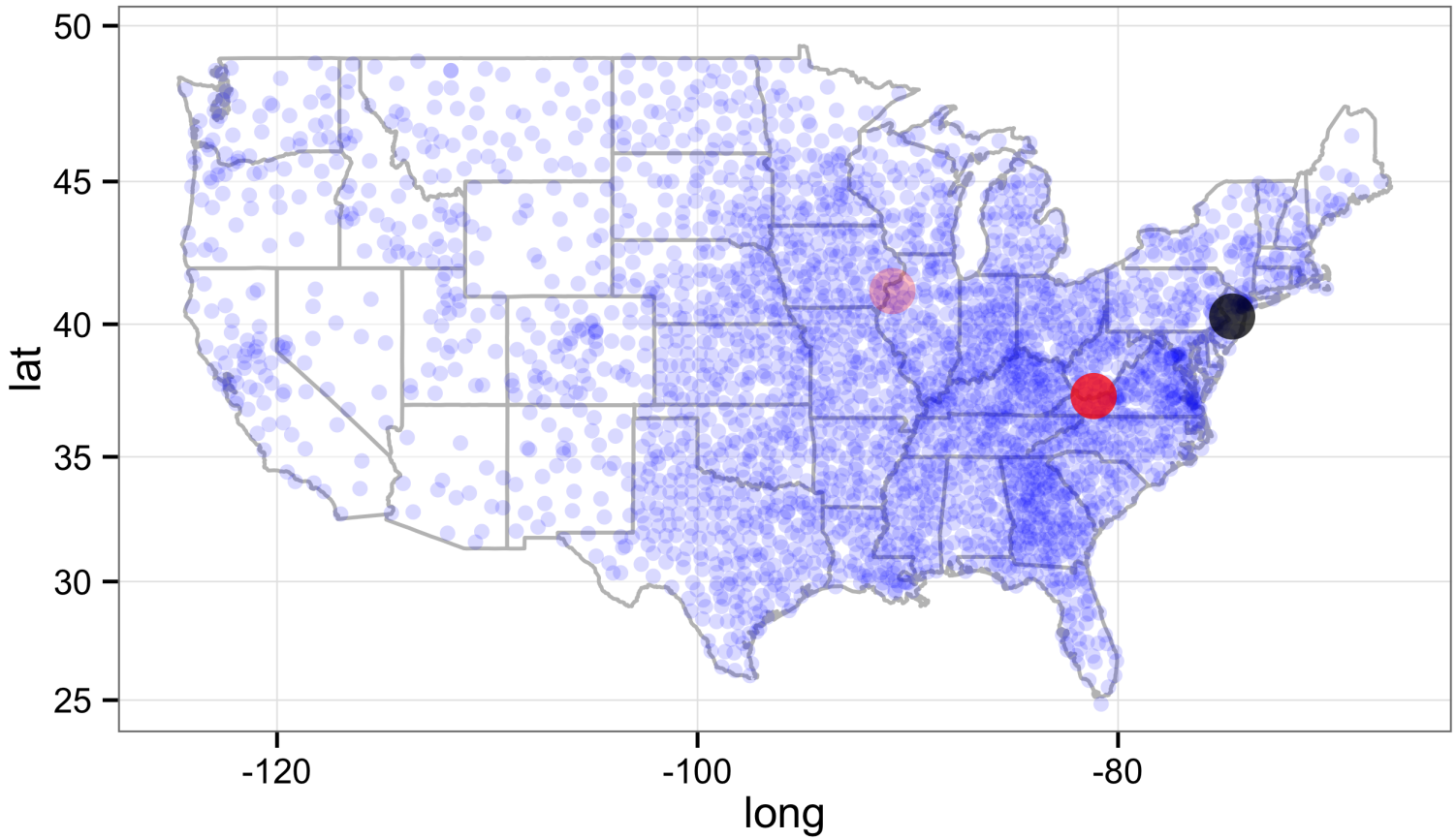
- Start with a county Mercer County, NJ

Pairwise Distances between US Counties



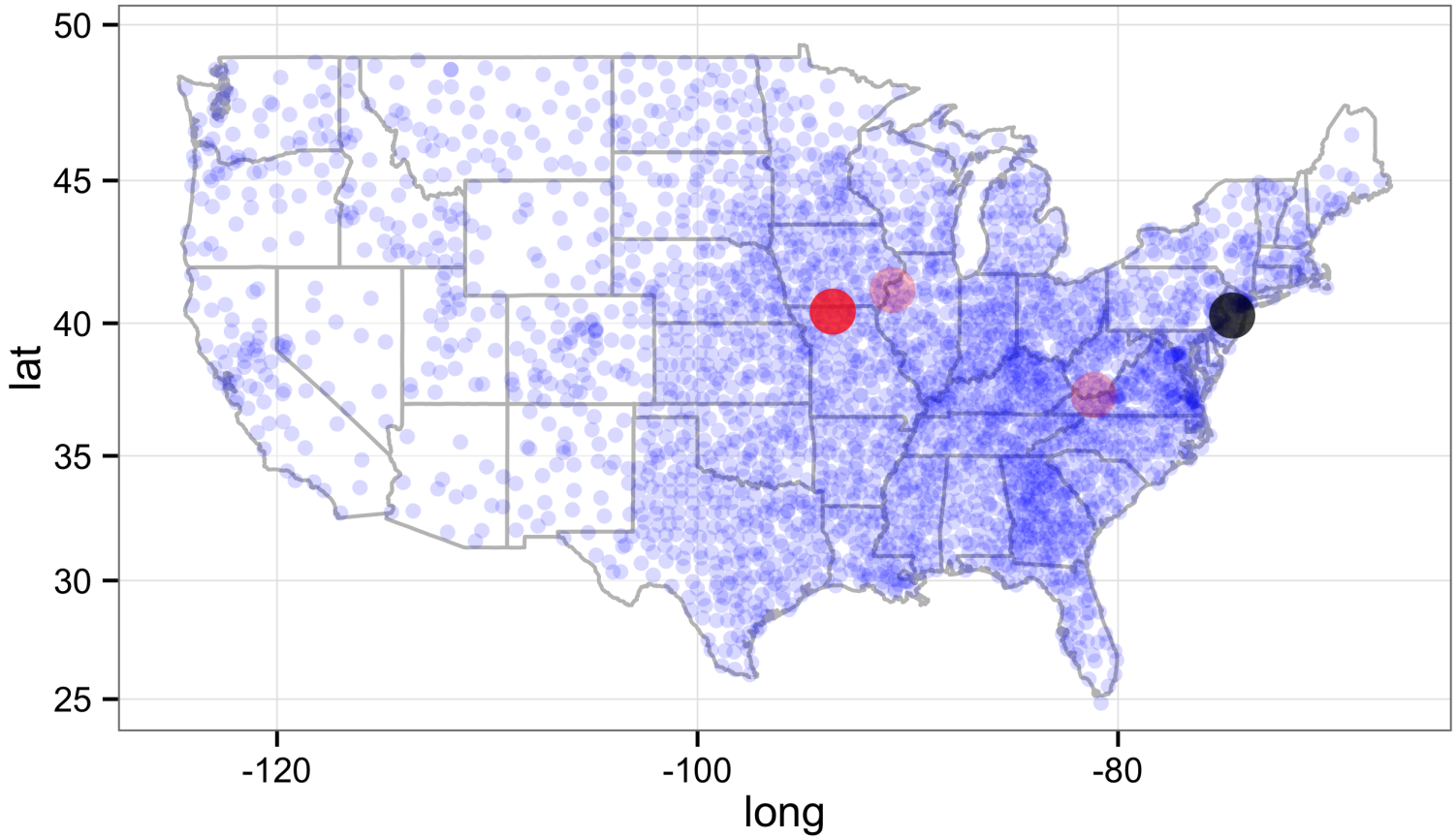
- Calculate the distance between Mercer County, NJ and Mercer County, IL.
- Store it.

Pairwise Distances between US Counties



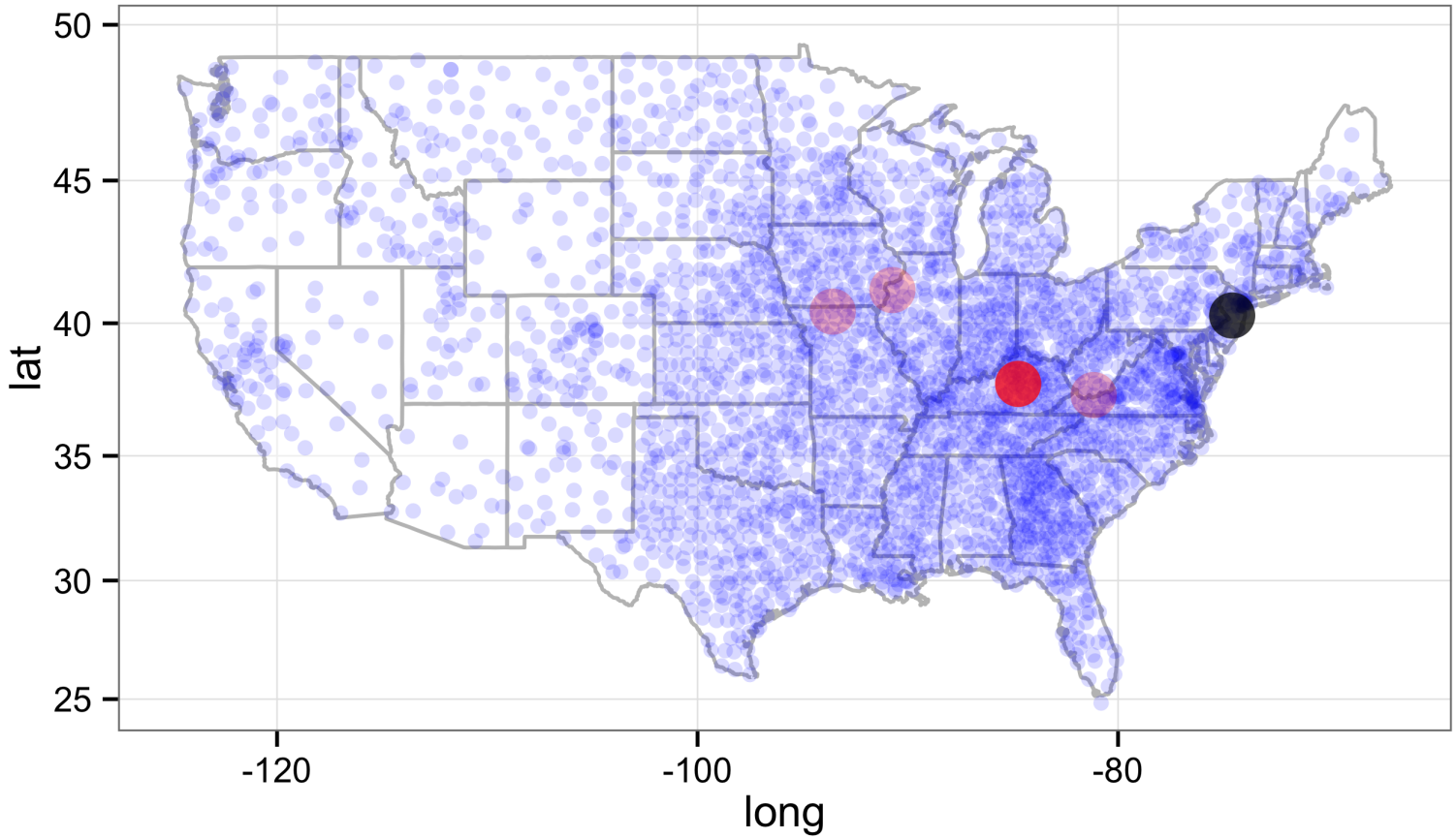
- Then, calculate the distance between Mercer County, NJ and Mercer County, WV.
- Store it.

Pairwise Distances between US Counties



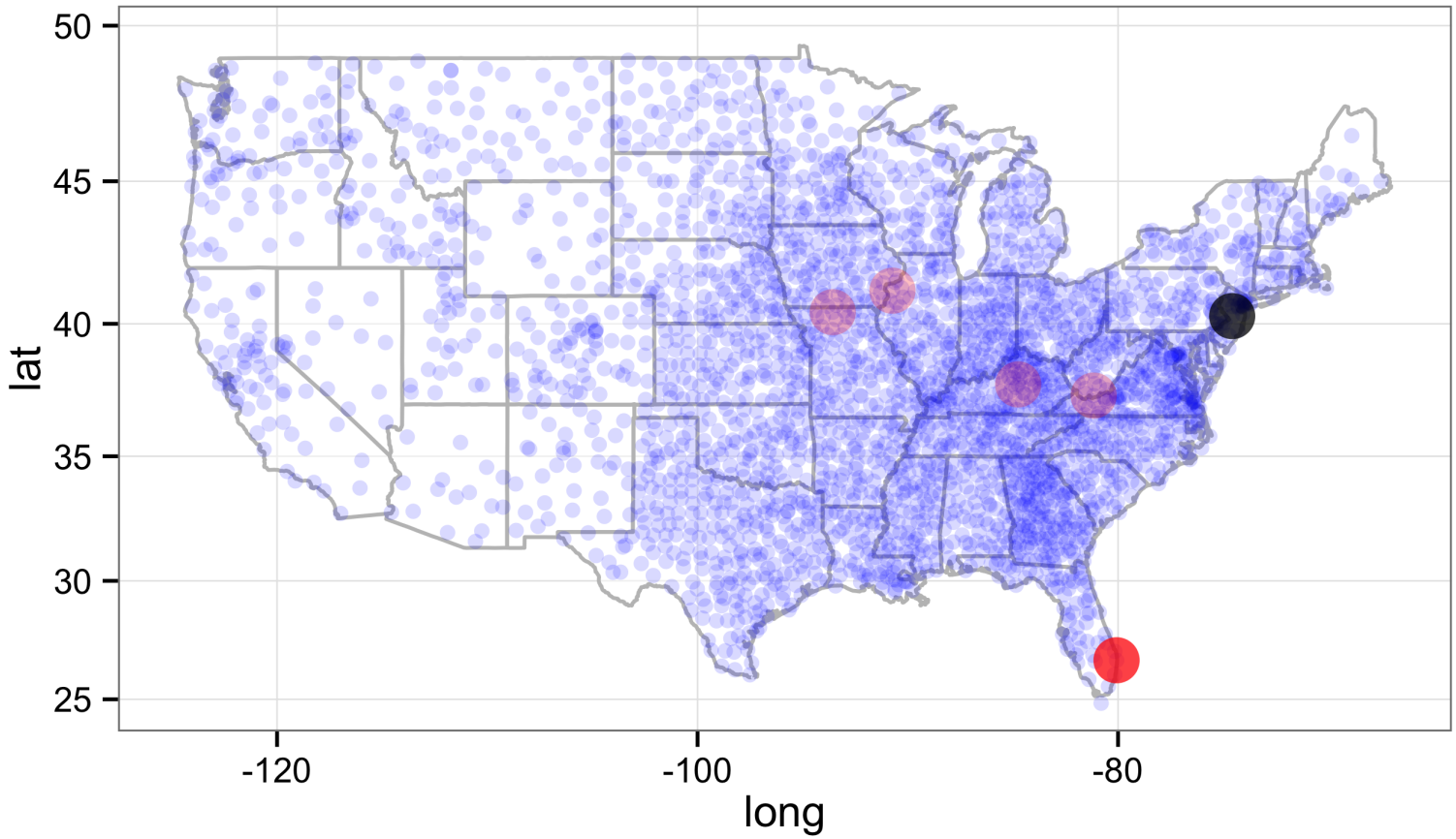
- Then, Mercer County, MO and store it.

Pairwise Distances between US Counties



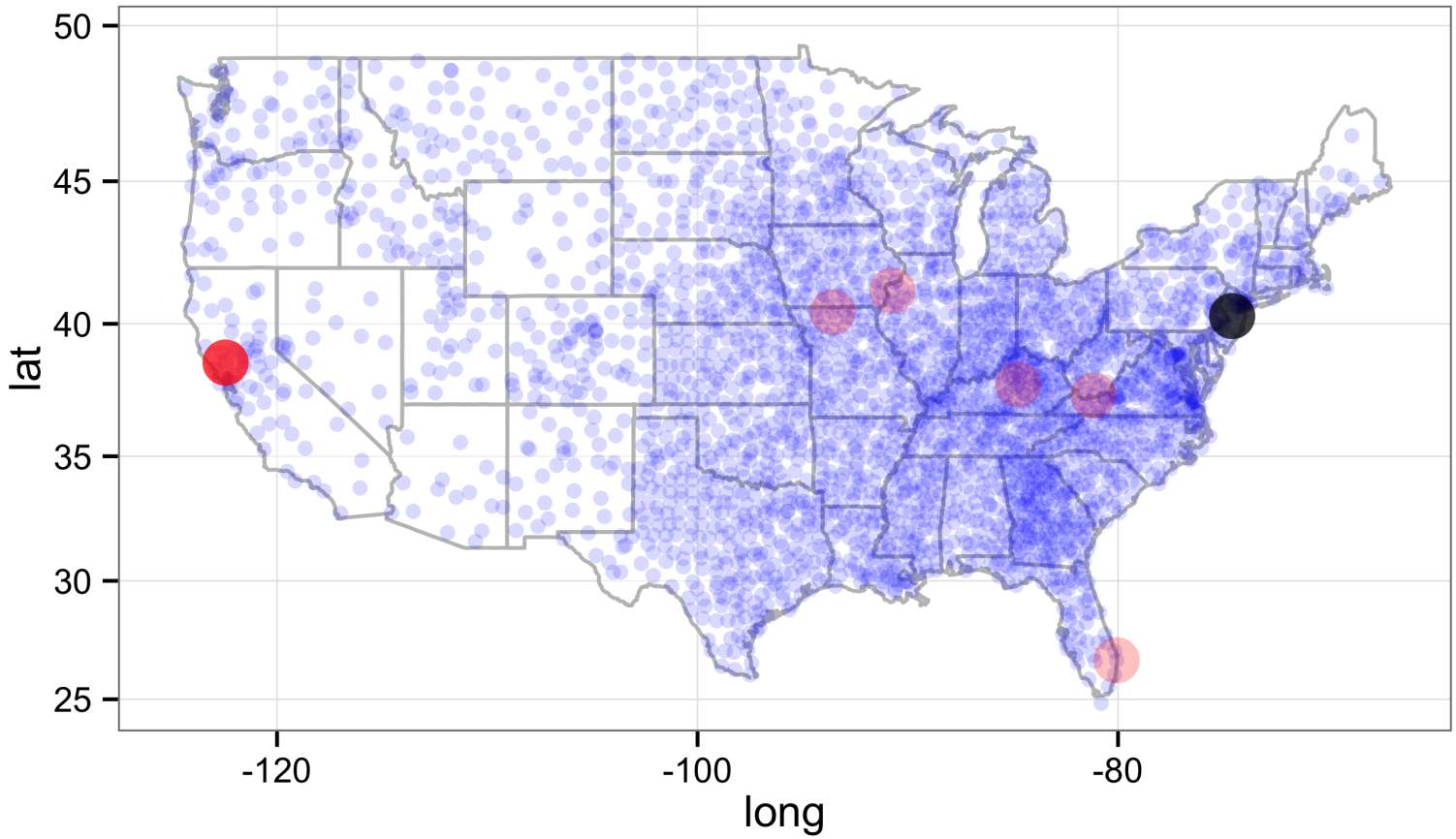
- Then, Mercer County, KY and store it.

Pairwise Distances between US Counties



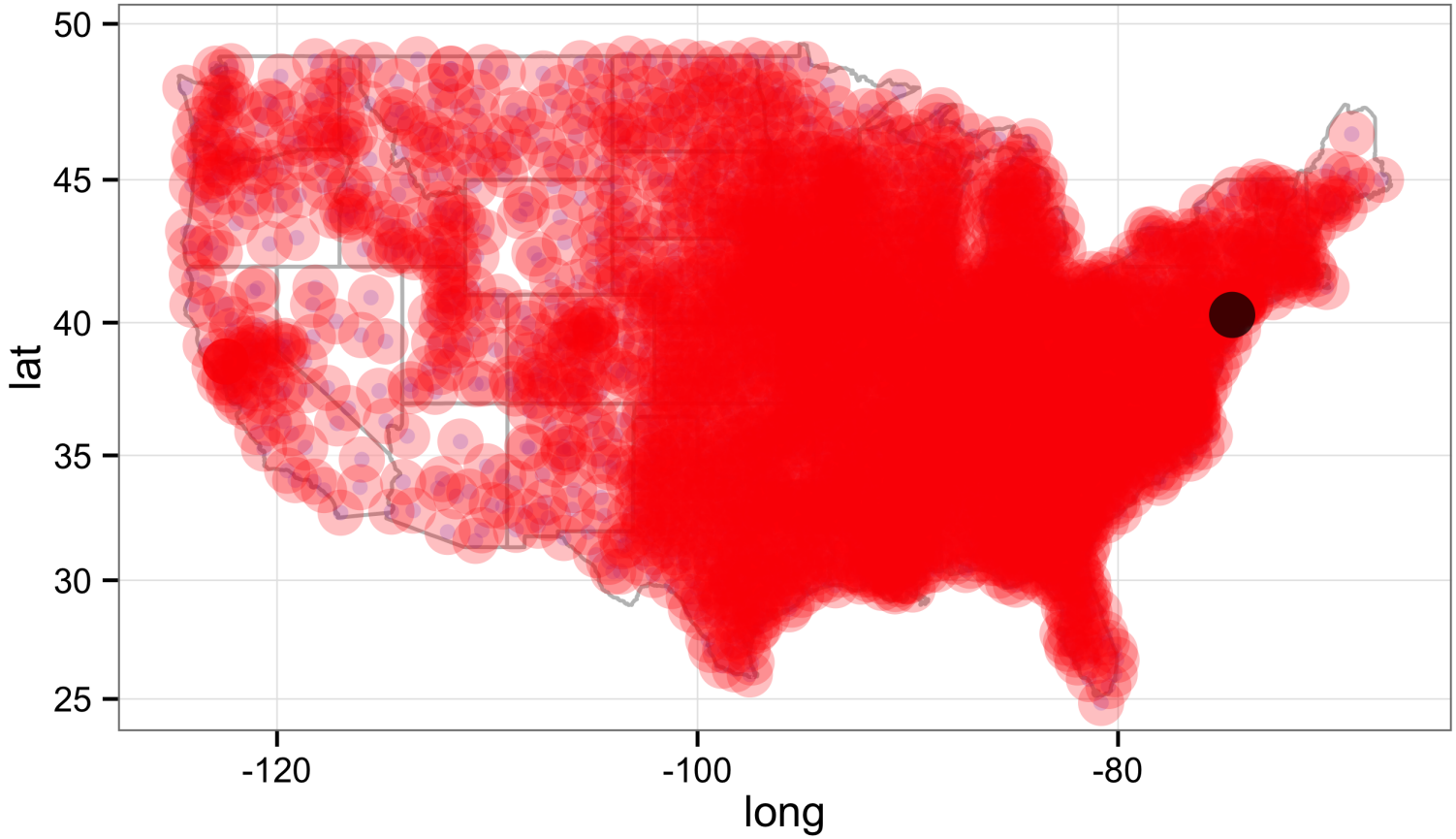
- Eventually, you would calculate the distance between Mercer County, NJ and Naples County, FL.
- Then store that.

Pairwise Distances between US Counties



- And, then, you would calculate the distance between Mercer County, NJ and Napa County, CA.
- Then store that.

Pairwise Distances between US Counties



- For Mercer County alone, thats 3,108 distance calculations.
- Without duplicate calculations: $\frac{3{,}109 \cdot 3{,}108}{2} = 4,831,386$ distances.
- With duplicate calculations: $9,665,881$ distances.

Geo-Coded US County Data

```
head(dfCounties)

##   latitude longitude state    county
## 1    34.22    -82.45    SC Abbeville
## 2    30.34    -92.35    LA    Acadia
## 3    37.71    -75.66    VA  Accomack
## 4    43.74   -116.41    ID       Ada
## 5    41.44    -94.63    IA     Adair
## 6    36.96    -85.40    KY     Adair

mCounties <- as.matrix(dfCounties[, 1:2])
mCountiesSmall <- mCounties[1:400, ]
```

- Main matrix with coordinates of 3,109 counties: `mCounties`
- Small matrix with coordinates of just 400 counties: `mCountiesSmall`

Element-wise Calculation of Distances

- For the true distance across the surface of the Earth between two points: use **law of cosines** or **haversine formula**.
- The Euclidean distance **is not** the true distance.
- But, it **does approximate** the structure of the computational task of calculating the true distances.

- Goal: Populate an N by N matrix with element (i, j) corresponding to the Euclidean distance (d_{ij}) between element i and element j.
- Where $x_i = latitude_i$ and $y_i = longitude_i$:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- Avoiding duplicate calculations $\leftrightarrow$ recognizing $d_{ij} = d_{ji}$.

Strategy for Comparing Different Implementations

- We will compare the relative performance (speed) of different implementations run in R.
- For each implementation, we will create an R `function` that takes a `matrix` of coordinates as input and returns an *N* by *N* distance `matrix` with pair-wise distances.
- We will make duplicate calculations.
- **Every implementation will perform the same computational task**.
- We will use the **microbenchmark** package to compare computation time.

Explicit Element-wise Calculation in R

- The most immediate approach is to calculate each element of the pair-wise distance matrix individually.

```
calcPWDe <- function(mat) {
    out <- matrix(data = NA, nrow = nrow(mat), ncol = nrow(mat))

    for (row in 1:nrow(out)) {
        for (col in 1:ncol(out)) {
            out[row, col] <- sqrt(((mat[row, 1] - mat[col, 1]) ^ 2 +
                                   (mat[row, 2] - mat[col, 2]) ^ 2
                                   )
                                  )
        }
    }
    return(out)
}
```

Performance

- Note: small `matrix` of coordinates.
- 2 `for()` Loops in R.

```
library("microbenchmark")
microbenchmark(loploop = calcPWDe(mCountiesSmall),
               times = 5
               )

## Unit: seconds
##       expr   min  lq median    uq   max neval
##  loploop 1.498 1.5  1.511 1.521 1.533     5
```

Is **1.5 seconds** fast or is it slow?

Implicit Element-wise Calculations via Vectors

- In R, it is much faster to work with vectors or vector-like objects whenever possible.
- Operating on vectors of length > 1 is faster than **repeated** operations on vectors of length 1.

```
system.time({
    (1:200000)^2
})

##    user  system elapsed
##   0.001   0.000   0.001
```

```
system.time({
    for(i in 1:200000) i^2
})

##    user  system elapsed
##   0.051   0.004   0.056
```

Where can we **vectorize** the code in `calcPWDe()`?

Implicit Element-wise Calculations via Vectors

```
calcPWDv <- function(mat) {
    out <- matrix(data = NA,
                  nrow = nrow(mat),
                  ncol = nrow(mat)
                  )

    for (row in 1:nrow(out)) {
        ## no loop over columns
        ## one row elem vs *all* col elems
        out[row, ] <- sqrt(((mat[row, 1] - mat[, 1]) ^ 2 +
                            ##                  !^!
                            (mat[row, 2] - mat[, 2]) ^ 2
                            ##                  !^!
                            )
                           )
    }
    return(out)
}
```

Performance

- Note: small `matrix` of coordinates.
- 2 `for()` Loops in R.
- 1 `for()` Loop in R.

```
microbenchmark(
    looploop = calcPWDe(mCountiesSmall),
    loop = calcPWDv(mCountiesSmall),
    times = 5
    )

## Unit: milliseconds
##       expr     min       lq  median       uq     max neval
##  looploop 1471.50 1489.41 1491.34 1495.34 1531.86     5
##       loop   19.29   19.44   19.57   19.61   20.53     5
```

Removing an unnecessary loop in R makes the code about **70 times** faster.

Seemingly Vectorized Code: The apply Family

```
calc1row <- function(row, mat) {
    return(sqrt(((row[1] - mat[, 1]) ^ 2 +
                 (row[2] - mat[, 2]) ^ 2
                )
               )
           )
}

calcPWDv2 <- function(mat) {
    out <- apply(mat,
                 MARGIN = 1,
                 FUN = calc1row,
                 mat = mat
                 )
    return(out)
}
```

Performance

- Note: small `matrix` of coordinates.
- 2 `for()` Loops in R.
- 1 `for()` Loop in R.
- 1 `apply()` in R.

```
## Unit: milliseconds
##       expr     min       lq  median       uq     max neval
##  looploop 1489.21 1489.74 1501.98 1506.33 1516.98     5
##       loop   17.81   18.50   18.87   19.15   62.75     5
##      apply   32.57   33.21   33.50   34.94   78.46     5
```

- Code can become more modular and easier to comprehend.
- Avoids explicit looping.
- But, **performance cost**.

Parallel Execution with foreach

- The **foreach** package provides a new looping construct: `foreach()`.
- Some similarities with the standard `for()` loop.
- Dramatically simplifies parallelization across multiple cores on a single computer of or multiple cores on a mutli-node HPC system (e.g., Tukey, Della, Adroit).

```
library("foreach")
library("doParallel")

## Loading required package:  iterators
## Loading required package:  parallel

nProcs <- 8
cl <- makeCluster(spec = nProcs, type = "PSOCK")
registerDoParallel(cl)
```

Parallel Execution with foreach

- In this application, we get better performance if we pre-assign multiple rows to each worker process.
- Split the *N* total row-specific calculations across 8 workers.

```
mRanges <- matrix(NA, nrow = nProcs, ncol = 3)
## initial #/per worker
mRanges[, 3] <- floor(nrow(dfCounties)/nProcs)
## number short
nShort <- nrow(dfCounties) - sum(mRanges[, 3])
## adj #/per worker
mRanges[, 3] <- mRanges[, 3] +
    c(rep(1, nShort), rep(0, nProcs - nShort))
## worker end points
mRanges[, 2] <- cumsum(mRanges[, 3])
## worker start points
mRanges[, 1] <- mRanges[, 2] - mRanges[, 3] + 1
```

Parallel Execution with foreach

- Each worker computes their part of the domain $(1, \ldots, N)$.

```
mRanges

##      [,1] [,2] [,3]
## [1,]    1  389  389
## [2,]  390  778  389
## [3,]  779 1167  389
## [4,] 1168 1556  389
## [5,] 1557 1945  389
## [6,] 1946 2333  388
## [7,] 2334 2721  388
## [8,] 2722 3109  388
```

Parallel Execution with foreach

```
calcPWDfe <- function(mat) {
    mOut <- foreach(i = 1:nProcs,
                    .combine = rbind, .noexport = "dfCounties",
                    .export = c("mRanges", "mCounties")) %dopar% {
                        mTmp <- matrix(NA,
                                       nrow = mRanges[, 3],
                                       ncol = nrow(mat)
                                       )
                        for(j in mRanges[i, 1]:mRanges[i, 2]) {
                            mTmp[i, ] <- sqrt((mat[i, 1]  - mat[, 1]) ^ 2 +
                                              (mat[i, 2]  - mat[, 2]) ^ 2
                                              )
                        }
                        return(mTmp)
                    }
    return(mOut)
}
```

Performance

- Note: full `matrix` of coordinates.
- 1 `for()` Loop in R.
- 1 `for()` Loop in R + Parallel Execution via `foreach()`.

```
microbenchmark("loop" = calcPWDv(mCounties),
               "parloop" = calcPWDfe(mCounties),
               times = 5
               )

## Unit: milliseconds
##     expr    min     lq median     uq  max neval
##     loop 1063.5 1113.9 1119.5 1154.2 1168     5
##  parloop  805.5  806.8  820.7  858.8 1543     5
```

The parallel R is **1.40 times** faster than the fastest sequential R code (and this is a problem ill-suited for `foreach` parallelization).

Explicit Element-wise Calculation through Rcpp

- R has the ability to interface with instructions from compiled code.
- Although it executes faster, writing correct code in a compiled language like C++ is more nuanced and less forgiving.
- The R package Rcpp does several things:
  1. Creates complete C++ files from user-provided C++ snippets.
  2. Provides handy object classes to make C++–level work seem R–like.
  3. Automates compilation of these files and creation of corresponding R functions.

Explicit Element-wise Calculation through Rcpp

```
library("Rcpp")
cppFunction("
NumericMatrix calcPWDcpp (NumericMatrix x) {
  int nrows =  x.nrow() ;
  int ncols = x.nrow() ;
  NumericMatrix out(nrows, ncols) ;

  for(int arow = 0; arow < nrows; arow++) {
    for(int acol = 0; acol < ncols; acol++) {
      double LatDS =  pow(x(arow, 0) - x(acol, 0), 2.0) ;
      double LongDS = pow(x(arow, 1) - x(acol, 1), 2.0) ;
       out(arow, acol) = sqrt(LatDS + LongDS) ;
     }
   }
  return(out) ;
 }
 "
            )
```

Explicit Element-wise Calculation through Rcpp

- Rcpp creates the R–level function for us.
- Instead of seeing the source, the R function points to a piece of the computer's memory.
- This is where the instructions from our compiled code are stored.

```
calcPWDcpp

## function (x)
## .Primitive(".Call")(<pointer: 0x10d5e0b50>, x)
```

Performance

- 1 `for()` Loop in R.
- 2 `for()` Loops in C++.

```
microbenchmark(
    loop = calcPWDv(mCounties),
    cpp = calcPWDcpp(mCounties),
    times = 5
    )

## Unit: milliseconds
##  expr    min     lq median     uq    max neval
##  loop 1052.0 1057.3 1057.9 1063.5 1160.0     5
##   cpp  117.7  129.9  187.3  187.5  191.4     5
```

The C++ code is **6 times** faster than the single R loop (sequential).

Parallel Execution through Rcpp and OpenMP

- OpenMP is an **API** for parallel execution.
- **API**: a standard way of interacting with software that is not your work
- Parallelizing your C++ code with OpenMP requires little work beyond the original C++ snippet.
- With a compatible environment and OpenMP installed:
  1. Instruct the compiler and linker to find the OpenMP API
  2. Insert the OpenMP **pragma** for the compiler to handle
- **pragma**: an instruction for the compiler, not part of the language itself.

```
Sys.setenv("PKG_CXXFLAGS" = "-fopenmp")
Sys.setenv("PKG_LIBS" = "-fopenmp")
```

Parallel Execution through Rcpp and OpenMP

```
cppFunction("
NumericMatrix calcPWDcppOMP (NumericMatrix x) {
  int nrows =  x.nrow() ;
  int ncols = nrows ;
  NumericMatrix out(nrows, ncols) ;
  omp_set_num_threads(8) ;
      // just include the pragma
      #pragma omp parallel for
      for(int arow = 0; arow < nrows; arow++) {
         for(int acol = 0; acol < ncols; acol++) {
            double LatDS =  pow(x(arow, 0) - x(acol, 0), 2.0) ;
            double LongDS = pow(x(arow, 1) - x(acol, 1), 2.0) ;
            out(arow, acol) = sqrt(LatDS + LongDS) ;
     }
    }

  return out ;
}
",
            include = "#include <omp.h>"
            )
```

Performance

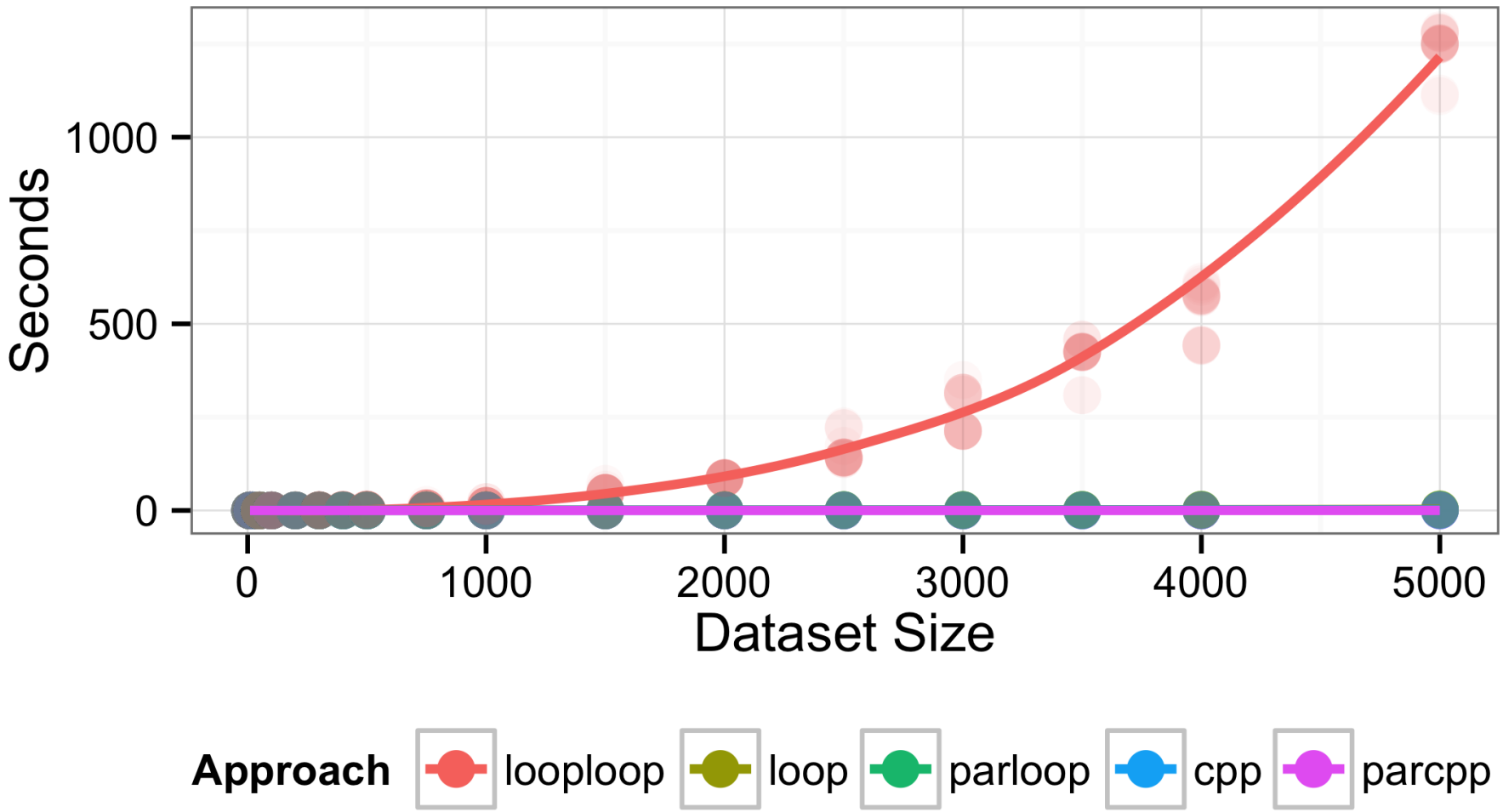
- 1 `for()` Loop in R.
- 2 `for()` Loops in C++.
- 2 `for()` Loops in C++ + Parallel Execution with OpenMP.

```
microbenchmark(
    loop = calcPWDv(mCounties),
    cpp = calcPWDcpp(mCounties),
    parcpp = calcPWDcppOMP(mCounties),
    times = 5
    )

## Unit: milliseconds
##    expr     min      lq  median     uq    max neval
##    loop 1055.43 1060.11 1079.77 1153.0 1185.5     5
##     cpp  131.43  137.42  138.68  187.3  192.6     5
##  parcpp   71.41   77.37   84.36  123.7  124.6     5
```

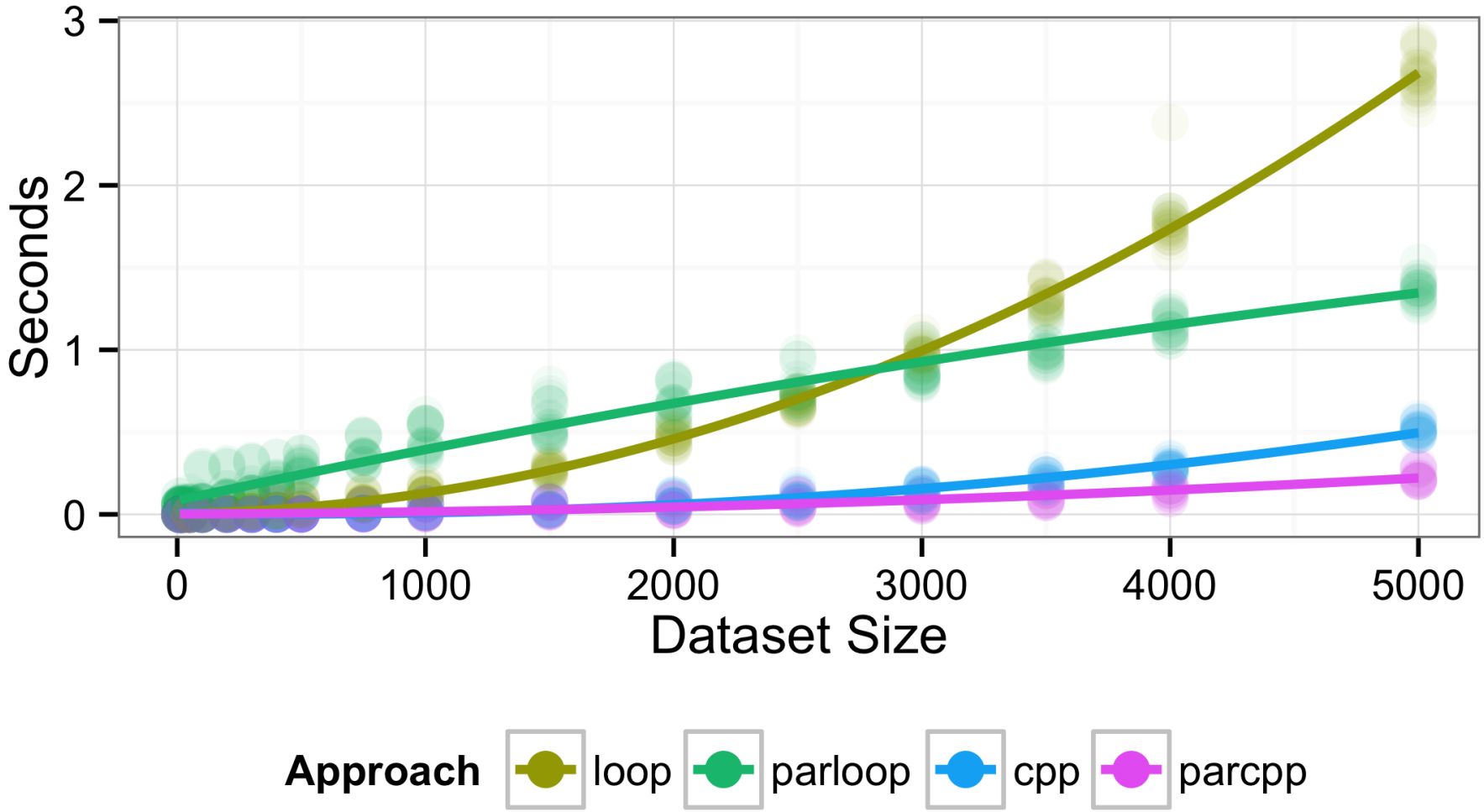
The parallel C++ code is over **15 times** faster than the fastest R code and is **2 times** faster than the sequential C++ code.

The Big Picture



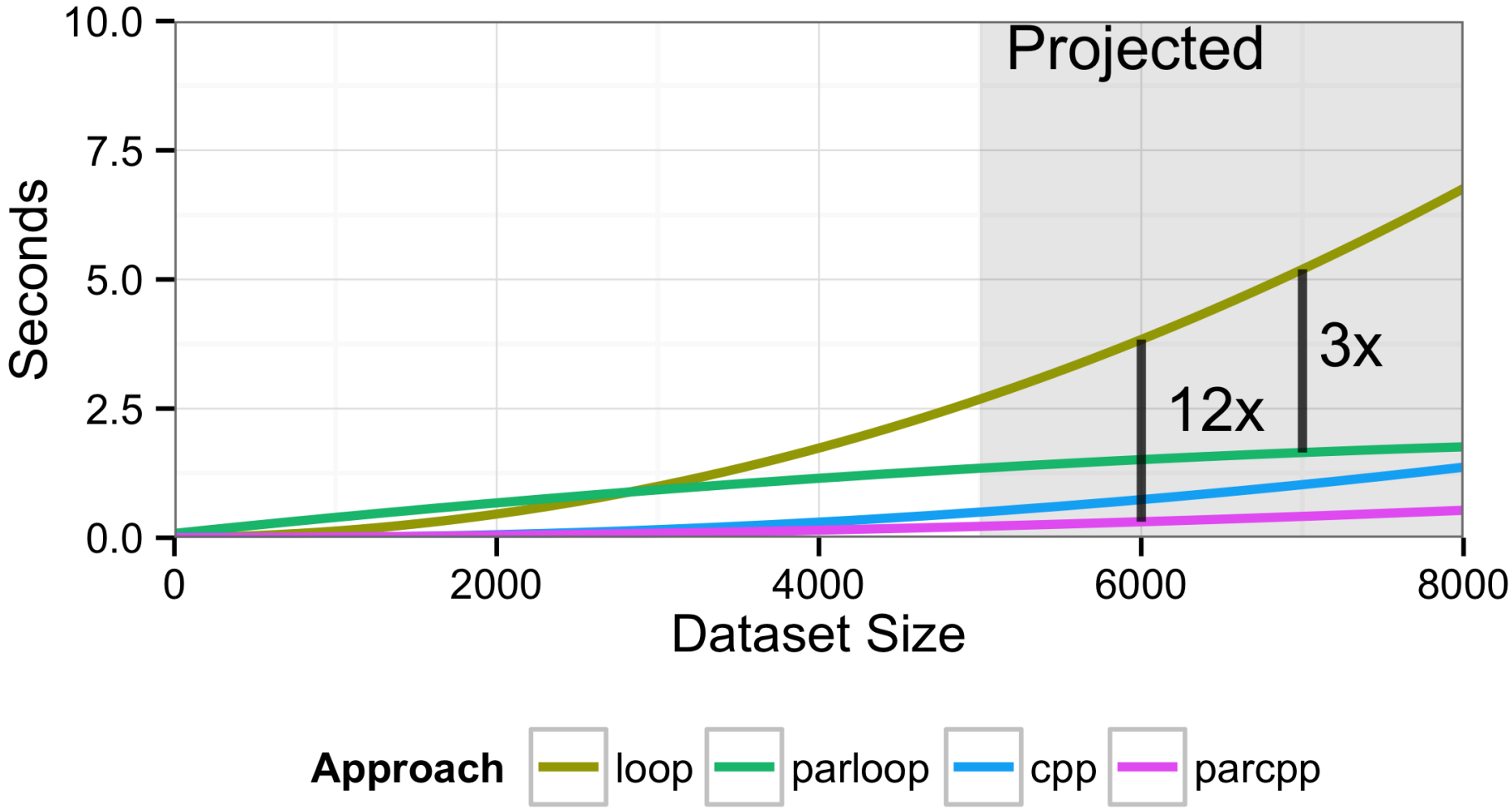
Scalability matters. The *looploop* (2 R loops) approach clearly **doesn't scale**. How do the others compare?

The Big Picture

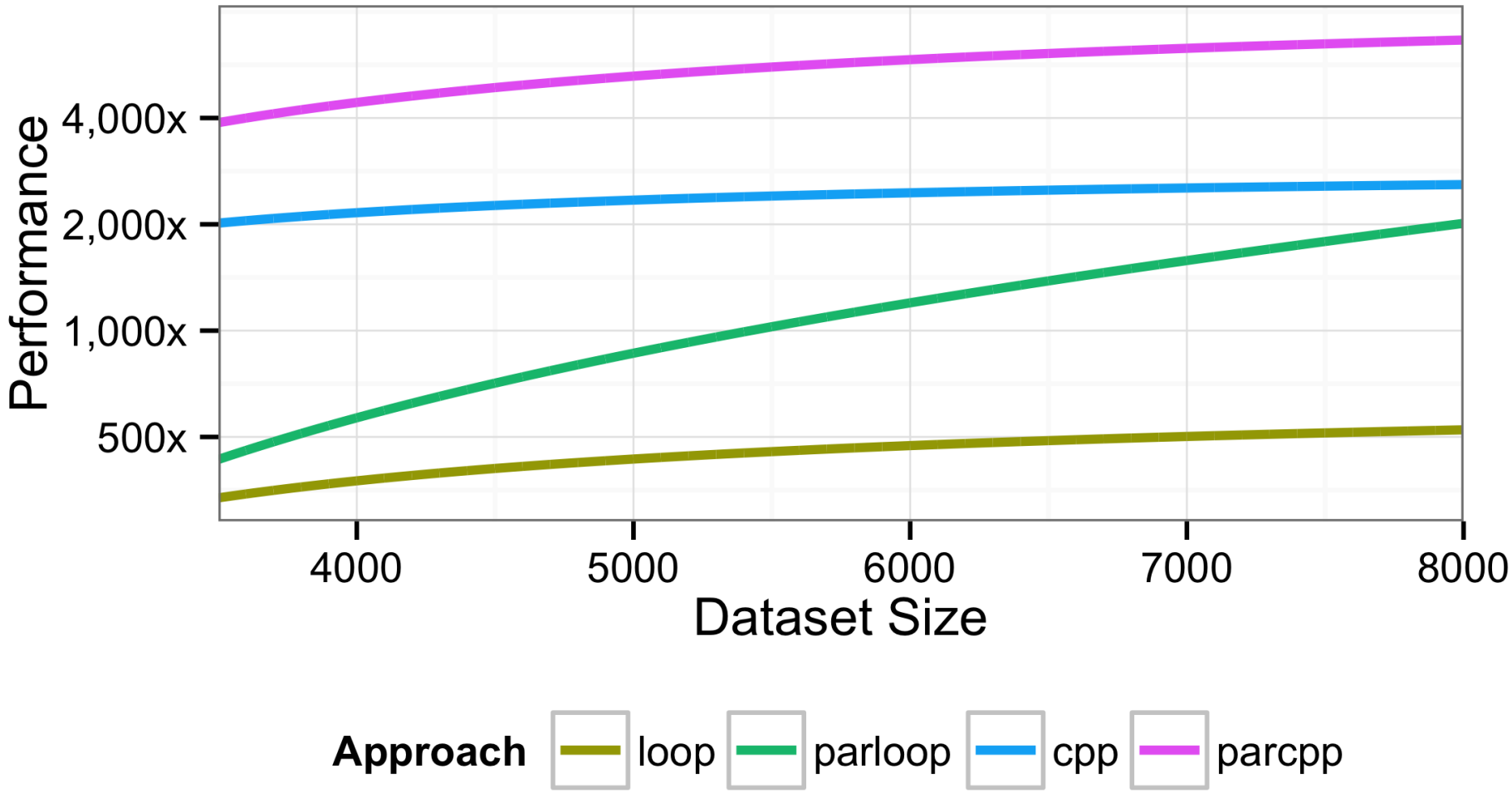


Both C++ approaches **scale very well**. Choosing between parallel `foreach()` and a sequential standard loop is **problem–specific**.

The Bigger Picture



The Bigger Picture



Relative to runtime of slowest method, "looploop".

This Morning

Case Study:

Performance of calculating (Euclidean) pairwise distances between all US counties across under different approaches

- Overview of how methods apply to a conceptually straightforward problem
- Narrow demonstration of possible performance gains
- Using R and freely available R packages

Coming Up...

- Dig deeper into each topic:
  1. Monitoring performance
  2. Simple performance improvements
  3. Parallel computing on local and remote machines
  4. Fast compiled code through Rcpp
  5. Complete Rcpp-based packages
- Apply this suite of programming approaches to additional examples:
  1. parametric, non-parametric bootstrap
  2. cross validation
  3. MC analysis of statistical properties
  4. surface of the Earth pair-wise distances
  5. importance-sampling
  6. sparse and dense linear regression
  7. EM and Bayesian Probit regression