**The problem**

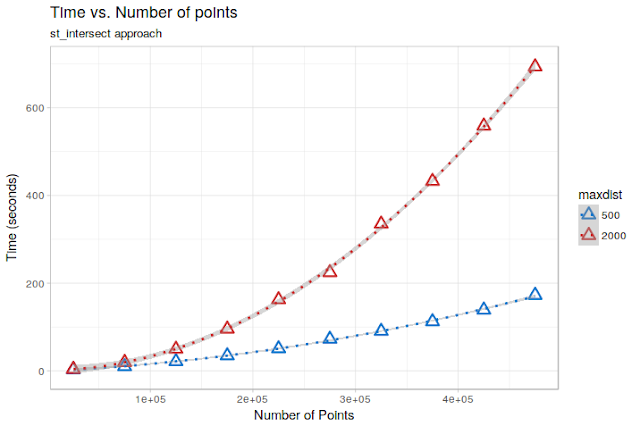
Last week, I replied to this [interesting question](https://stackoverflow.com/questions/48650274/spatial-efficient-way-of-finding-all-points-within-x-meters-of-a-point) posted by @Tim\_K over stackoverflow. He was seeking efficient solutions to **identify all points falling within a maximum distance of xx meters with respect to each single point in a spatial points dataset**.

If you have a look at the thread, you will see that a simple solution based on creating a “buffered” polygon dataset beforehand and then intersecting it with the original points is quite fast ***for “reasonably sized” datasets***, thanks to sf spatial indexing capabilities which reduce the number of the required comparisons to be done (See <http://r-spatial.org/r/2017/06/22/spatial-index.html>). In practice, something like this:

# create test data: 50000 uniformly distributed points on a "square" of 100000  
# metres  
maxdist <- 500  
pts <- data.frame(x = runif(50000, 0, 100000),  
 y = runif(50000, 0, 100000),  
 id = 1:50000) %>%  
 sf::st\_as\_sf(coords = c("x", "y"))  
# create buffered polygons  
pts\_buf <- sf::st\_buffer(pts, maxdist)  
# Find points within 500 meters wrt each point  
int <- sf::st\_intersects(pts\_buf, pts)  
int

## Sparse geometry binary predicate list of length 50000, where the predicate was `intersects'  
## first 10 elements:  
## 1: 1, 11046, 21668, 25417  
## 2: 2, 8720, 12595, 23620, 26926, 27169, 39484  
## 3: 3, 11782, 20058, 27869, 33151, 47864  
## 4: 4, 35665, 45691  
## 5: 5, 37093, 37989  
## 6: 6, 31487  
## 7: 7, 38433, 42009, 45597, 49806  
## 8: 8, 12129, 31486  
## 9: 9, 27840, 35577, 36797, 40906  
## 10: 10, 15482, 16615, 26103, 41417

However, this starts to have **problems over really large datasets**, because the total number of comparisons to be done still rapidly increase besides the use of spatial indexes. A **test** done by changing the number of points in the above example in the range 25000 – 475000 shows for example this kind of behavior, for two different values of maxdist (500 and 2000 m):

[](https://i2.wp.com/1.bp.blogspot.com/-KcqMy6mXBOo/WowAzjEyJSI/AAAAAAAANGE/_1-YSdF0gSgkAFwOg3hg0BqYUjMGTbWUACLcBGAs/s1600/Rplot01.png?ssl=1)

On the test dataset, the relationships are *almost perfectly quadratic* (due to the uniform distribution of points). Extrapolating them to the **12 Million points dataset of the OP**, we would get an execution time of about 14 hours for maxdist = 500, and a staggering **3.5 days** formaxdist = 2000. Still doable, but not ideal…

My suggestion to the OP was therefore to **“split” the points in chunks** based on the x-coordinate and then work on a per-split basis, eventually assigning each chunk to a different core within a parallellized cycle.

In the end, I got curious and decided to give it a go **to see what kind of performance improvement it was possible to obtain** with that kind of approach. You can find results of some tests below.

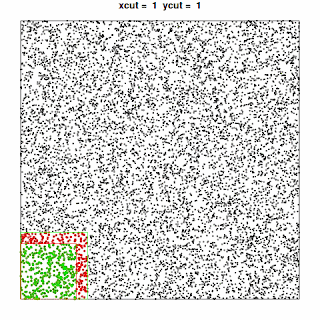
**A (possible) solution: Speeding up computation by combining data.table and sf\_intersect**

**The idea here is to use a simple divide-and-conquer approach.**

**We first split the total spatial extent of the dataset in a certain number of regular quadrants. We then iterate over the quadrants and for each one we:**

1. **Extract the points contained into the quadrant and apply a buffer to them;**
2. **Extract the points contained in a slightly larger area, computed by expanding the quadrant by an amount equal to the *maximum distance for which we want to identify the “neighbors”;***
3. **Compute and save the intersection between the buffered points and the points contained in the “expanded” quadrant**

**“Graphically”, this translates to exploring the dataset like this:**

**[](https://i0.wp.com/1.bp.blogspot.com/-AG71hBW1vBw/Wov9Qxlo_8I/AAAAAAAANF4/OjpSHuc8Nx8vtRWiSSGHbgq_QCV8YraSQCLcBGAs/s1600/animation2.gif?ssl=1)**

**, where the points included in the current “quadrant” are shown in green and the additional points needed to perform the analysis for that quadrant are shown in red.**

**Provided that the subsetting operations do not introduce an excessive overhead (i.e., they are fast enough…) this should provide a performance boost, because it should consistently reduce the total number of comparisons to be done.**

**Now, every “R” expert will tell you that if you need to perform fast subsetting over large datasets the way to go is to use properly indexeddata.tables, which provide lightning-speed subsetting capabilities.**

**So, let’s see how we could code this in a functions:**

**points\_in\_distance <- function(in\_pts,  
 maxdist,  
 ncuts = 10) {  
  
 require(data.table)  
 require(sf)  
 # convert points to data.table and create a unique identifier  
 pts <- data.table(in\_pts)  
 pts <- pts[, or\_id := 1:dim(in\_pts)[1]]  
  
 # divide the extent in quadrants in ncuts\*ncuts quadrants and assign each  
 # point to a quadrant, then create the index over "x" to speed-up  
 # the subsetting  
 range\_x <- range(pts$x)  
 limits\_x <-(range\_x[1] + (0:ncuts)\*(range\_x[2] - range\_x[1])/ncuts)  
 range\_y <- range(pts$y)  
 limits\_y <- range\_y[1] + (0:ncuts)\*(range\_y[2] - range\_y[1])/ncuts  
 pts[, `:=`(xcut = as.integer(cut(x, ncuts, labels = 1:ncuts)),  
 ycut = as.integer(cut(y, ncuts, labels = 1:ncuts)))] %>%  
 setkey(x)  
  
 results <- list()  
 count <- 0  
 # start cycling over quadrants  
 for (cutx in seq\_len(ncuts)) {  
  
 # get the points included in a x-slice extended by `maxdist`, and build  
 # an index over y to speed-up subsetting in the inner cycle  
 min\_x\_comp <- ifelse(cutx == 1,  
 limits\_x[cutx],  
 (limits\_x[cutx] - maxdist))  
 max\_x\_comp <- ifelse(cutx == ncuts,  
 limits\_x[cutx + 1],  
 (limits\_x[cutx + 1] + maxdist))  
 subpts\_x <- pts[x >= min\_x\_comp & x < max\_x\_comp] %>%  
 setkey(y)  
  
 for (cuty in seq\_len(ncuts)) {  
 count <- count + 1  
  
 # subset over subpts\_x to find the final set of points needed for the  
 # comparisons  
 min\_y\_comp <- ifelse(cuty == 1,  
 limits\_y[cuty],  
 (limits\_x[cuty] - maxdist))  
 max\_y\_comp <- ifelse(cuty == ncuts,  
 limits\_x[cuty + 1],  
 (limits\_x[cuty + 1] + maxdist))  
 subpts\_comp <- subpts\_x[y >= min\_y\_comp & y < max\_y\_comp]  
  
 # subset over subpts\_comp to get the points included in a x/y chunk,  
 # which "neighbours" we want to find. Then buffer them by maxdist.  
 subpts\_buf <- subpts\_comp[ycut == cuty & xcut == cutx] %>%  
 sf::st\_as\_sf() %>%   
 sf::st\_buffer(maxdist)  
  
 # retransform to sf since data.tables lost the geometric attrributes  
 subpts\_comp <- sf::st\_as\_sf(subpts\_comp)  
  
 # compute the intersection and save results in a element of "results".  
 # For each point, save its "or\_id" and the "or\_ids" of the points within "dist"  
 inters <- sf::st\_intersects(subpts\_buf, subpts\_comp)  
  
 # save results  
 results[[count]] <- data.table(  
 id = subpts\_buf$or\_id,  
 int\_ids = lapply(inters, FUN = function(x) subpts\_comp$or\_id[x]))  
 }  
 }  
 data.table::rbindlist(results)  
}**

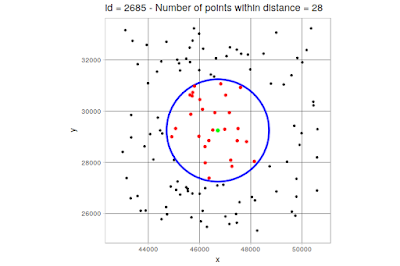
**The function takes as input a points sf object, a target distance and a number of “cuts” to use to divide the extent in quadrants, and provides in output a data frame in which, for each original point, the “ids” of the points within maxdist are reported in the int\_ids list column.**

Now, **let’s see if this works**:

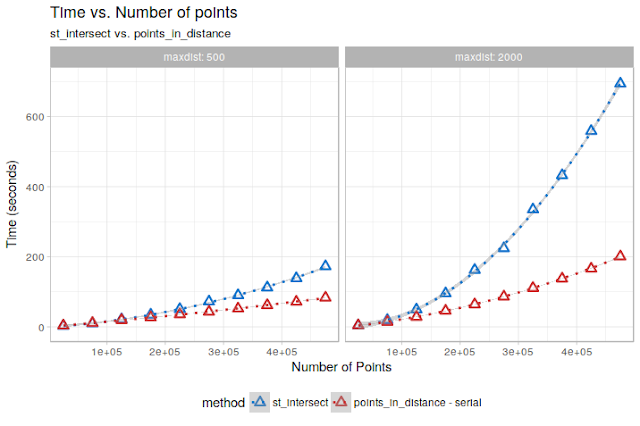
**pts <- data.frame(x = runif(20000, 0, 100000),  
 y = runif(20000, 0, 100000),  
 id = 1:20000) %>%  
 st\_as\_sf(coords = c("x", "y"), remove = FALSE)  
maxdist <- 2000  
out <- points\_in\_distance(pts, maxdist = maxdist, ncut = 10)  
out**

**## id int\_ids  
## 1: 15119 2054,18031, 9802, 8524, 4107, 7412,  
## 2: 14392 12213,10696, 6399,14392, 4610, 1200,  
## 3: 14675 2054,18031, 9802, 8524, 4107, 7412,  
## 4: 3089 12293,18031, 8524, 4107,12727, 2726,  
## 5: 17282 9802,8524,4107,7412,2726,1275,  
## ---   
## 19995: 248 16610, 7643, 8059,15998,16680, 1348,  
## 19996: 8433 16821,15638,16680, 3876,13851, 1348,  
## 19997: 17770 11060, 7643, 8059,19868, 7776,10146,  
## 19998: 11963 9948, 9136,15956,18512, 9219, 8925,  
## 19999: 15750 5291,18093,14462,15362,12575, 5189,**

**# get a random point  
sel\_id <- sample(pts$id,1)  
pt\_sel <- pts[sel\_id, ]  
pt\_buff <- pt\_sel %>% sf::st\_buffer(maxdist)  
# get ids of points within maxdist  
id\_inters <- unlist(out[id == sel\_id, ]$int\_ids)  
pt\_inters <- pts[id\_inters,]  
  
#plot results  
ggplot(pt\_buff) + theme\_light() +  
 geom\_point(data = pts, aes(x = x, y = y), size = 1) +  
 geom\_sf(col = "blue", size = 1.2, fill = "transparent") +  
 geom\_sf(data = pt\_inters, col = "red", size = 1.5) +  
 geom\_point(data = pt\_sel, aes(x = x, y = y), size = 2, col = "green") +  
 xlim(st\_bbox(pt\_buff)[1] - maxdist, st\_bbox(pt\_buff)[3] + maxdist) +  
 ylim(st\_bbox(pt\_buff)[2] - maxdist, st\_bbox(pt\_buff)[4] + maxdist) +   
 ggtitle(paste0("id = ", sel\_id, " - Number of points within distance = ", length(id\_inters)))**

**[](https://i1.wp.com/3.bp.blogspot.com/-S2NspQ46veo/WowBskp4_wI/AAAAAAAANGM/BdjMp99hg3QWexWxoPNo8xBLIpREsF5mACLcBGAs/s1600/Rplot2.png?ssl=1)**

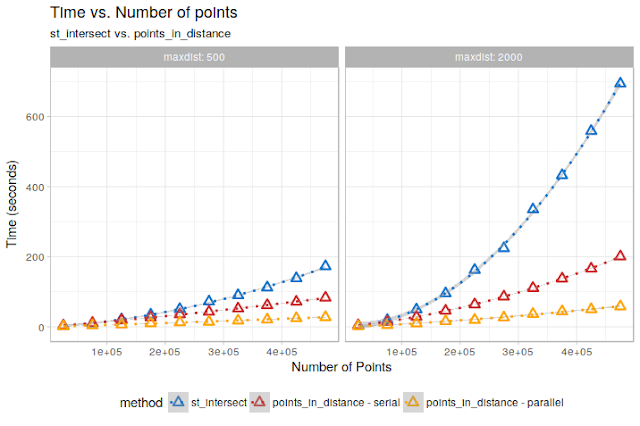
**So far, so good. Now, let’s do the same exercise with varying number of points to see how it behaves in term of speed:**

**[](https://i2.wp.com/2.bp.blogspot.com/-jirwQtpYwuY/WowBureJrGI/AAAAAAAANGQ/bN94b1sanqg0KuWVtpn3USNPQjCmdC2ugCEwYBhgL/s1600/Rplot3.png?ssl=1)**

**Already not bad! In particular for the maxdist = 2000 case, we get a quite large speed improvement!**

**However, a nice thing about the points\_in\_distance approach is that it is easily parallelizable. All is needed is to change some lines of the function so that the outer loop over the x “chunks” exploits a parallel backend of some kind. (You can find an example implementation exploiting foreachin**[**this gist**](https://gist.github.com/lbusett/247dc9b0b6bed04ac1b45c03999be348)**)**

**On a not-particularly-fast PC, using a 6-cores parallelization leads to this:**

**[](https://i1.wp.com/4.bp.blogspot.com/-PcjVQP4Yx_E/WowBzAV6vnI/AAAAAAAANGU/ZhXsCjd4RTgVRhksNymc3KU84m2xH0H4ACEwYBhgL/s1600/Rplot4.png?ssl=1)**

**Looking good! Some more skilled programmer could probably squeeze out even more speed from it by some additional data.table magic, but the improvement is very noticeable.**

**In terms of execution time, extrapolating again to the “infamous” 12 Million points dataset, this would be what we get:**

| Method | **Maxdist** | **Expected completion time (hours)** |
| --- | --- | --- |
| st\_intersect | 500 | 15.00 |
| points\_in\_distance – serial | 500 | 2.50 |
| points\_in\_distance – parallel | 500 | 0.57 |
| st\_intersect | 2000 | **85.00** |
| points\_in\_distance – serial | 2000 | 15.20 |
| points\_in\_distance – parallel | 2000 | **3.18** |

So, we get **a 5-6X speed improvement** already on the “serial” implementation, and **another 5X** thanks to parallelization over 6 cores! On themaxdist = 2000 case, this means going **from more than 3 days to about 3 hours**. And if we had more cores and RAM to throw at it, it would finish in minutes!

**Nice!**

**Final Notes**

* **The timings shown here are merely indicative, and related to the particular test-dataset we built. On a less uniformly distributed dataset I would expect a lower speed improvement.**
* Some time is “wasted” because **sf does not (yet) extend data.tables**, making it necessary to recreate sf objects from thedata.table subsets.
* The parallel implementation is quick-and-dirty, and it is a bit of a **memory-hog**! Be careful before throwing at it 25 processors!
* Speed is **influenced in a non-trivial way by the number of “cuts”** used to subdivide the spatial extent. There may be a sweet-spot related to points distribution and maxdist allowing reaching maximum speed.
* A similar approach for parallelization could exploit **repeatedly “cropping” the original sf points object** over the extent of the chunk/extended chunk. The data.table approach seems however to be faster.