

Parallel Processing

Data Wrangling, Session 7d

Kieran Healy

Code Horizons

October 2024

Load the packages, as always

```
library(here)      # manage file paths
library(socviz)    # data and some useful functions
library(tidyverse) # your friend and mine

## Magic new package
# install.packages("furrr")
library(furrr) # Also loads `future`
```

```
Loading required package: future
```

Split, Apply, Combine

A lot of analysis has this pattern

We start with a dataset

We **split** it into pieces, usually according to some feature or categorical variable, or by file or something.

We do something—the *same* thing—to each of those pieces. That is we **apply** a function or procedure to the pieces. That procedure returns some result for each piece. The result will be of the same form: a number, a vector of counts, summary statistics, a model, a list, whatever.

Finally we **combine** those results into a final piece of output, usually a tibble or somesuch.

For example

`dplyr` is all about this.

```
gss_sm %>  
  count(bigregion)  
  
# A tibble: 4 × 2  
  bigregion     n  
  <fct>       <int>  
1 Northeast    488  
2 Midwest     695  
3 South        1052  
4 West         632
```

We *split* into groups, *apply* the `sum()` function within the groups, and *combine* the results into a new tibble showing the resulting sum per group. The various `dplyr` functions are oriented to doing this in a way that gives you a consistent set of outputs.

For example: **split**

We can split, apply, combine in various ways.

Base R has the **split()** function:

```
out ← mtcars ▷  
  split(mtcars$cyl)  
summary(out) # mtcars split into a list of data frames by the `cyl` variable
```

	Length	Class	Mode
4	11	data.frame	list
6	11	data.frame	list
8	11	data.frame	list

For example: split

Tidyverse has `group_split()`:

For example: **apply**

The application step is “I want to fit a linear model to each piece”

```
out ← mtcars ▷  
  group_split(cyl) ▷  
  map(\(df) lm(mpg ~ wt + hp + gear, data = df))
```

For example: **apply**

The application step is “I want to fit a linear model to each piece” and get a summary

```
mtcars >  
group_split(cyl) >  
map(\(df) lm(mpg ~ wt + hp + gear, data = df)) >  
map(summary) >  
map_dbl("r.squared")
```

```
[1] 0.7301860 0.6597413 0.4995237
```

For example: **combine**

In this case the “combine” step is implicitly at the end: we get a vector of R squareds back, and it’s as long as the number of groups.

```
mtcars %>  
  group_split(cyl) %>  
  map(\(df) lm(mpg ~ wt + hp + gear, data = df)) %>  
  map(summary) %>  
  map_dbl("r.squared")
```

```
[1] 0.7301860 0.6597413 0.4995237
```

For example: `apply`

This is also what we're doing more elegantly (staying within a tibble structure) if we `nest()` and use `broom` to get a summary out.

```
mtcars %>
  group_by(cyl) %>
  nest() %>
  mutate(model = map(data, ~lm(mpg ~ wt + hp + gear, data = df)),
         perf = map(model, broom::glance)) %>
  unnest(perf)

# A tibble: 3 × 15
# Groups:   cyl [3]
  cyl data     model  r.squared adj.r.squared sigma statistic p.value    df
  <dbl> <tibble> <lm>    <dbl>        <dbl>    <dbl>    <dbl>    <dbl> <dbl>
1     6 <tibble> <lm>    0.660      0.319    1.20    1.94    0.300    3
2     4 <tibble> <lm>    0.730      0.615    2.80    6.31    0.0211   3
3     8 <tibble> <lm>    0.500      0.349    2.06    3.33    0.0648   3
# i 6 more variables: logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
#   df.residual <int>, nobs <int>
```

How this happens

In each of these cases, the data is processed *sequentially* or *serially*. R splits the data according to your instructions, applies the function or procedure to each one in turn, and combines the outputs in order out the other side. Your computer's processor is handed each piece in turn.

How this happens

For small tasks that's fine. But for bigger tasks it gets inefficient quickly.

```
## From Henrik Bengtsson's documentation for future/furr
slow_sum <- function(x) {
  sum <- 0
  for (value in x) {
    Sys.sleep(1.0) ## one-second slowdown per value
    sum <- sum + value
  }
  sum
}

# This takes > ten seconds to run.
tic toc::tic()
slow_sum(1:10)
```

```
[1] 55
```

```
tic toc::toc()
```

```
10.036 sec elapsed
```

If *this* is the sort of task we have to apply to a bunch of things, it's going to take ages.

That's Embarrassing

A feature of many split-apply-combine activities is that it *does not matter* what order the “apply” part happens to the groups. All that matters is that we can combine the results at the end, no matter what order they come back in. E.g. in the `mtcars` example the model being fit to the 4-cylinder cars group doesn’t depend in any way on the results of the model being fit to the 8-cylinder group.

This sort of situation is *embarrassingly parallel*.

That's Embarrassing

When a task is embarrassingly parallel, and the number of pieces or groups is large enough or complex enough, then if we can do them at the same time then we should. There is some overhead—we have to keep track of where each piece was sent and when the results come back in—but if that's low enough in comparison to doing things serially, then we should parallelize the task.

Multicore Computing

Parallel computing used to mean “I have a cluster of computers at my disposal”. But modern CPUs are constructed in a semi-modular fashion. They have some number of “cores”, each one of which is like a small processor in its own right. In effect you have a computer cluster already.

Some Terms

Socket: The physical connection on your computer that houses the processor. These days, mostly there's just one.

Core: The part of the processor that actually performs the computation. Most modern processors have multiple cores. Each one can do wholly independent work.

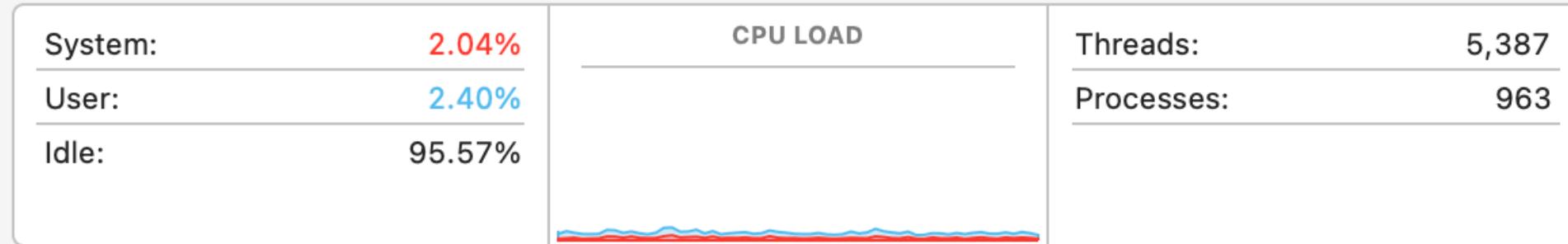
Process: A single instance of a running task or program (R, Slack, Chrome, etc). A core can only run one process at a time. But, cores are *fast*. And so, they can run many *threads*

Thread: A piece of a process that can share memory and resources with other threads. If you have enough power you can do something Intel called **hyperthreading**, which is a way of dividing up a physical core into (usually) two *logical* cores that work on the same clock cycle and share some resources on the physical core.

Cluster: A collection of things that are capable of hosting cores. Might be a single socket (on your laptop) or an old-fashioned room full of many physical computers that can be made to act as if they were a single machine.

Multicore Computing

Most of the time, even when we are using them, our computers sit around doing PRETTY MUCH NOTHING.



```
## How many cores do we have?  
parallelly :: availableCores()
```

```
system  
16
```

Remember, processor clock cycles are *really fast*. They're measured in billions of cycles per second.

We need to put those cores to work!

Future and furrr

These packages make parallel computing *way* more straightforward than it used to be. In particular, for Tidyverse-centric workflows, **furrr** provides a set of **future_** functions that are drop-in replacements for **map** and friends. So **map()** becomes **future_map()** and so on.

We set things up like this:

```
# library(furrr) # We did this already
plan(multisession) # Default will use the available resources
## Note difference between multisession and multicore
## [tl;dr: the latter is faster, but you can't use it on Windows]
```

Toy Example

```
# Another slow function (from
# Grant McDermott this time)
slow_square ← function(x = 1) {
  x_sq ← x^2
  out ← tibble(value = x, value_sq = x_sq)
  Sys.sleep(2) # ZZZZ
  out
}

tictoc::tic("Serially")
## This is of course way slower than just writing
## slow_square(1:15) but nvm that for now
serial_out ← map(1:15, slow_square) ▷
  list_rbind()
tictoc::toc()
```

Serially: 30.099 sec elapsed

Toy Example

```
tictoc::tic("Parallelized")
parallel_out ← future_map_dfr(1:15, slow_square)
tictoc::toc()
```

```
Parallelized: 3.825 sec elapsed
```

```
identical(serial_out, parallel_out)
[1] TRUE
```

NOAA Data

Data obtained with:

```
mkdir raw
cd raw
wget --no-parent -r -l inf --wait 5 --random-wait 'https://www.ncei.noaa.gov/data/sea-surface-temperature-optimum-interpolation/v2.1/access/avhrr/'
```

This tries to be polite with the NOAA: it enforces a wait time and in addition randomizes it to make it variably longer. Note also no boundary on depth of folder recursion. There are a lot of files (>15,000). Doing it this way will take several *days* in real time (though much much less in actual transfer time of course).

NOAA Temperature Data

Raw data directories:

```
basename(fs :: dir_ls(here :: here("avhrr")))
```

```
[1] "198109" "198110" "198111" "198112" "198201" "198202" "198203" "198204"
[9] "198205" "198206" "198207" "198208" "198209" "198210" "198211" "198212"
[17] "198301" "198302" "198303" "198304" "198305" "198306" "198307" "198308"
[25] "198309" "198310" "198311" "198312" "198401" "198402" "198403" "198404"
[33] "198405" "198406" "198407" "198408" "198409" "198410" "198411" "198412"
[41] "198501" "198502" "198503" "198504" "198505" "198506" "198507" "198508"
[49] "198509" "198510" "198511" "198512" "198601" "198602" "198603" "198604"
[57] "198605" "198606" "198607" "198608" "198609" "198610" "198611" "198612"
[65] "198701" "198702" "198703" "198704" "198705" "198706" "198707" "198708"
[73] "198709" "198710" "198711" "198712" "198801" "198802" "198803" "198804"
[81] "198805" "198806" "198807" "198808" "198809" "198810" "198811" "198812"
[89] "198901" "198902" "198903" "198904" "198905" "198906" "198907" "198908"
[97] "198909" "198910" "198911" "198912"
[ reached getOption("max.print") -- omitted 421 entries ]
```

NOAA Temperature Data

Raw data files, in netCDF (Version 4) format

```
basename(fs :: dir_ls(here :: here("avhrr", "202402")))
```

```
[1] "oisst-avhrr-v02r01.20240201.nc" "oisst-avhrr-v02r01.20240202.nc"
[3] "oisst-avhrr-v02r01.20240203.nc" "oisst-avhrr-v02r01.20240204.nc"
[5] "oisst-avhrr-v02r01.20240205.nc" "oisst-avhrr-v02r01.20240206.nc"
[7] "oisst-avhrr-v02r01.20240207.nc" "oisst-avhrr-v02r01.20240208.nc"
[9] "oisst-avhrr-v02r01.20240209.nc" "oisst-avhrr-v02r01.20240210.nc"
[11] "oisst-avhrr-v02r01.20240211.nc" "oisst-avhrr-v02r01.20240212.nc"
[13] "oisst-avhrr-v02r01.20240213.nc" "oisst-avhrr-v02r01.20240214.nc"
[15] "oisst-avhrr-v02r01.20240215.nc" "oisst-avhrr-v02r01.20240216.nc"
[17] "oisst-avhrr-v02r01.20240217.nc" "oisst-avhrr-v02r01.20240218.nc"
[19] "oisst-avhrr-v02r01.20240219.nc" "oisst-avhrr-v02r01.20240220.nc"
[21] "oisst-avhrr-v02r01.20240221.nc" "oisst-avhrr-v02r01.20240222.nc"
[23] "oisst-avhrr-v02r01.20240223.nc" "oisst-avhrr-v02r01.20240224.nc"
[25] "oisst-avhrr-v02r01.20240225.nc" "oisst-avhrr-v02r01.20240226.nc"
[27] "oisst-avhrr-v02r01.20240227.nc" "oisst-avhrr-v02r01.20240228.nc"
[29] "oisst-avhrr-v02r01.20240229.nc"
```

Some Prep

```
## Seasons for plotting
season <- function(in_date){
  br = yday(as.Date(c("2019-03-01",
                      "2019-06-01",
                      "2019-09-01",
                      "2019-12-01"))))
  x = yday(in_date)
  x = cut(x, breaks = c(0, br, 366))
  levels(x) = c("Winter", "Spring", "Summer", "Autumn", "Winter")
  x
}

season_lab <- tibble(yrday = yday(as.Date(c("2019-03-01",
                                              "2019-06-01",
                                              "2019-09-01",
                                              "2019-12-01"))),
                     lab = c("Spring", "Summer", "Autumn", "Winter"))
```

NOAA Temperature Data

Raw data files

```
#install.packages("ncdf4")
#install.packages("terra")
library(terra)

## For the filename processing
## This one gives you an unknown number of chunks each with approx n elements
chunk ← function(x, n) split(x, ceiling(seq_along(x)/n))

## This one gives you n chunks each with an approx equal but unknown number of elements
chunk2 ← function(x, n) split(x, cut(seq_along(x), n, labels = FALSE))

## All the daily .nc files:
all_fnames ← as.character(fs::dir_ls(here("avhrr"), recurse = TRUE, glob = "*.nc"))
chunked_fnames ← chunk(all_fnames, 25)

length(all_fnames)
```

```
[1] 15549
```

```
length(chunked_fnames)
```

```
[1] 622
```

NOAA Temperature Data

The data is in netCDF (Version 4) format. An interesting self-documenting format. Read one in with the netCDF reader.

```
tmp ← ncdf4::nc_open(all_fnames[10000])
tmp
```

```
File /Users/kjhealy/Documents/courses/data_wrangling/avhrr/200901/oisst-avhrr-v02r01.20090120.nc
(NC_FORMAT_NETCDF4):
```

```
4 variables (excluding dimension variables):
  short anom[lon,lat,zlev,time]  (Chunking: [1440,720,1,1])  (Compression: shuffle,level 4)
    long_name: Daily sea surface temperature anomalies
    _FillValue: -999
    add_offset: 0
    scale_factor: 0.00999999977648258
    valid_min: -1200
    valid_max: 1200
    units: Celsius
  short err[lon,lat,zlev,time]  (Chunking: [1440,720,1,1])  (Compression: shuffle,level 4)
    long_name: Estimated error standard deviation of analysed_sst
    units: Celsius
    _FillValue: -999
    add_offset: 0
    scale_factor: 0.00999999977648258
    valid_min: 0
```

NOAA Temperature Data

For analysis we are going to use the `terra` package, which understands many GIS type formats. Fundamentally we have a grid or *raster* of data that's 1440 x 720 in size. The data has several *layers*, each one a specific measure, such as sea-surface temperature, sea ice coverage, and so on.

```
tmp ← terra::rast(all_fnames[10000])
tmp

class      : SpatRaster
dimensions : 720, 1440, 4  (nrow, ncol, nlyr)
resolution : 0.25, 0.25  (x, y)
extent     : 0, 360, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84
sources    : oisst-avhrr-v02r01.20090120.nc:anom
              oisst-avhrr-v02r01.20090120.nc:err
              oisst-avhrr-v02r01.20090120.nc:ice
              oisst-avhrr-v02r01.20090120.nc:sst
varnames   : anom (Daily sea surface temperature anomalies)
              err (Estimated error standard deviation of analysed_sst)
              ice (Sea ice concentration)
              ...
names      : anom_zlev=0, err_zlev=0, ice_zlev=0, sst_zlev=0
unit       : Celsius,    Celsius,      %,    Celsius
time (days) : 2009-01-20
```

Terra can understand rasters that are aggregated into slabs or “bricks” covering

NOAA Temperature Data

Read one in with `terra`. Get the `sst` (Sea Surface Temperature) layer only.

```
tmp ← terra::rast(all_fnames[10000])["sst"]
tmp

class      : SpatRaster
dimensions  : 720, 1440, 1  (nrow, ncol, nlyr)
resolution  : 0.25, 0.25  (x, y)
extent     : 0, 360, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84
source      : oisst-avhrr-v02r01.20090120.nc:sst
varname     : sst (Daily sea surface temperature)
name        : sst_zlev=0
unit        : Celsius
time (days) : 2009-01-20
```

NOAA Temperature Data

What reading a *chunk* of filenames (with all their layers) does:

```
tmp2 ← terra::rast(chunked_fnames[[10]])  
tmp2  
  
class      : SpatRaster  
dimensions  : 720, 1440, 100  (nrow, ncol, nlyr)  
resolution  : 0.25, 0.25  (x, y)  
extent     : 0, 360, -90, 90  (xmin, xmax, ymin, ymax)  
coord. ref. : lon/lat WGS 84  
sources     : oisst-avhrr-v02r01.19820414.nc:anom  
              oisst-avhrr-v02r01.19820414.nc:err  
              oisst-avhrr-v02r01.19820414.nc:ice  
              ... and 97 more source(s)  
varnames    : anom (Daily sea surface temperature anomalies)  
              err (Estimated error standard deviation of analysed_sst)  
              ice (Sea ice concentration)  
              ...  
names       : anom_zlev=0, err_zlev=0, ice_zlev=0, sst_zlev=0, anom_zlev=0, err_zlev=0, ...  
unit        : Celsius, Celsius, %, Celsius, Celsius, Celsius, ...  
time (days) : 1982-04-14 to 1982-05-08
```

NOAA Temperature Data

Write a function to get a file, read in the SST raster, and get its area-weighted mean, for the North Atlantic region only.

```
process_raster ← function(fnames, crop_area = c(-80, 0, 0, 60), var = "sst") {  
  
  tdf ← terra::rast(as.character(fnames))[var] ▷  
    terra::rotate() ▷ # Fix 0-360 lon  
    terra::crop(crop_area) # Manually crop to a defined box. Default is Atlantic lat/lon box  
  
  wts ← terra::cellSize(tdf, unit = "km") # For scaling  
  
  data.frame(  
    date = terra::time(tdf),  
    # global() calculates a quantity for the whole grid on a particular SpatRaster  
    # so we get one weighted mean per file that comes in  
    wt_mean_sst = terra::global(tdf, "mean", weights = wts, na.rm=TRUE)$weighted_mean  
  )  
}
```

NOAA Temperature Data

Try it on one data file:

```
process_raster(all_fnames[1000]) %>  
  as_tibble()  
  
# A tibble: 1 × 2  
date      wt_mean_sst  
<date>        <dbl>  
1 1984-05-27     20.6
```

Try it on one *chunk* of data files:

```
process_raster(chunked_fnames[[500]]) ▷  
  as_tibble()
```

```
# A tibble: 25 × 2  
  date      wt_mean_sst  
  <date>        <dbl>  
1 2015-11-02     22.8  
2 2015-11-03     22.8  
3 2015-11-04     22.8  
4 2015-11-05     22.7  
5 2015-11-06     22.7  
6 2015-11-07     22.7  
7 2015-11-08     22.6  
8 2015-11-09     22.5  
9 2015-11-10     22.5  
10 2015-11-11    22.4  
# i 15 more rows
```

NOAA Temperature Data

Do it in parallel for all files:

```
# library(furrr) # We did this already
# plan(multisession)

tic toc::tic("Terra Method")
df ← future_map(chunked_fnames, process_raster) ▷
  list_rbind() ▷
  as_tibble() ▷
  mutate(date = ymd(date),
        year = lubridate::year(date),
        month = lubridate::month(date),
        day = lubridate::day(date),
        yrday = lubridate::yday(date),
        season = season(date))
tic toc::toc()
```

Terra Method: 30.309 sec elapsed

```
dim(df)
```

```
[1] 15549    7
```

NOAA Temperature Data

```
df %>  
  slice_sample(n = 10)
```

A tibble: 10 × 7

	date	wt_mean_sst	year	month	day	yrday	season
	<date>	<dbl>	<dbl>	<dbl>	<int>	<dbl>	<fct>
1	2000-07-12	22.8	2000	7	12	194	Summer
2	1991-11-12	21.6	1991	11	12	316	Autumn
3	2016-06-01	21.6	2016	6	1	153	Summer
4	2016-08-04	24.1	2016	8	4	217	Summer
5	2005-04-23	20.3	2005	4	23	113	Spring
6	2012-01-25	19.7	2012	1	25	25	Winter
7	2006-11-18	22.3	2006	11	18	322	Autumn
8	2001-12-22	20.7	2001	12	22	356	Winter
9	1986-09-20	23.2	1986	9	20	263	Autumn
10	2016-04-01	19.6	2016	4	1	92	Spring

NOAA Temperature Data

Make our cores work even harder

```
## Seas of the world polygons
seas ← sf::read_sf(here("seas"))

seas
```

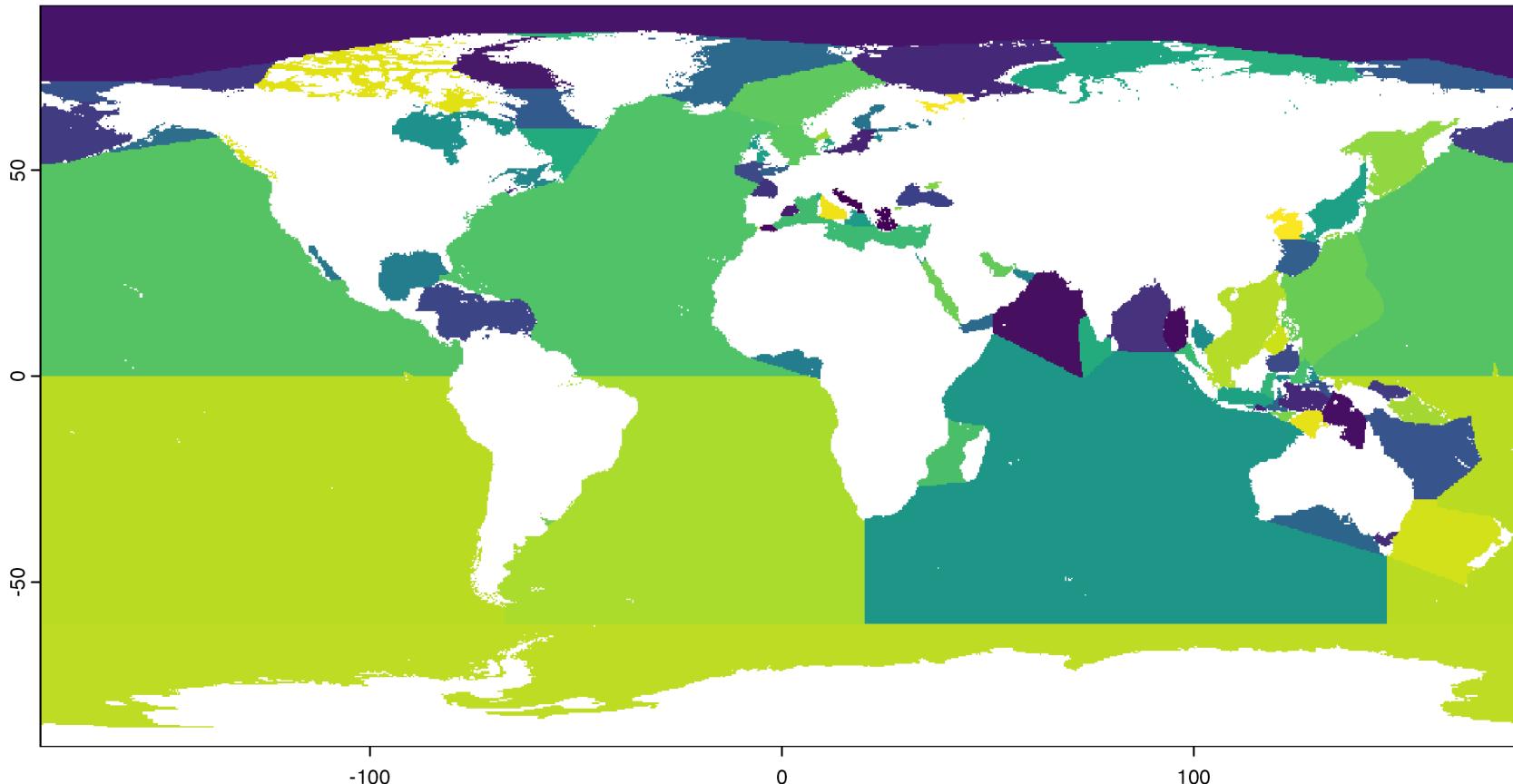
```
Simple feature collection with 101 features and 10 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -180 ymin: -85.5625 xmax: 180 ymax: 90
Geodetic CRS:  WGS 84
# A tibble: 101 × 11
  NAME        ID   Longitude Latitude min_X  min_Y max_X  max_Y    area MRGID
  <chr>      <chr>    <dbl>    <dbl> <dbl> <dbl> <dbl>    <dbl> <dbl>
1 Rio de La Pl... 33     -56.8   -35.1  -59.8  -36.4  -54.9  -31.5  3.18e4  4325
2 Bass Strait   62A    146.     -39.5   144.   -41.4   150.   -37.5  1.13e5  4366
3 Great Austra... 62     133.     -36.7   118.   -43.6   146.   -31.5  1.33e6  4276
4 Tasman Sea    63     161.     -39.7   147.   -50.9   175.   -30    3.34e6  4365
5 Mozambique C... 45A    40.9    -19.3   32.4   -26.8   49.2   -10.5  1.39e6  4261
6 Savu Sea       480    122.     -9.48  119.   -10.9   125.   -8.21  1.06e5  4343
7 Timor Sea      48i    128.     -11.2   123.   -15.8   133.   -8.18  4.34e5  4344
8 Bali Sea       481    116.     -7.93  114.   -9.00  117.   -7.01  3.99e4  4340
9 Coral Sea      64     157.     -18.2   141.   -30.0   170.   -6.79  4.13e6  4364
10 Flores Sea     48j    120.     -7.51  117.   -8.74  123.   -5.51  1.03e5 4341
# i 91 more rows
```

NOAA Temperature Data

```
## Rasterize the seas polygons using one of the nc files
## as a reference grid for the rasterization process
## They're all on the same grid so it doesn't matter which one
one_raster ← all_fnames[1]
seas_vect ← terra::vect(seas)
tmp_tdf_seas ← terra::rast(one_raster)["sst"] ▷
  terra::rotate()
seas_zonal ← terra::rasterize(seas_vect, tmp_tdf_seas, "NAME")
```

NOAA Temperature Data

```
plot(seas_zonal, legend = FALSE)
```



Pointers and wrapping

We'll need to apply this grid of seas and oceans repeatedly — once for each `.nc` file — which means it has to get passed to all our worker cores. But it is stored as a pointer, so we can't do that directly. We need to wrap it:

```
# If we don't do this it can't be passed around
# across the processes that future_map() will spawn
seas_zonal_wrapped ← terra::wrap(seas_zonal)
```

NOAA Temperature Data

Write a function very similar to the other one.

```
process_raster_zonal ← function(fnames, var = "sst") {  
  
  tdf_seas ← terra::rast(as.character(fnames))[var] ▷  
  terra::rotate() ▷  
  terra::zonal(terra::unwrap(seas_zonal_wrapped), mean, na.rm = TRUE)  
  
  colnames(tdf_seas) ← c("ID", paste0("d_", terra::time(terra::rast(as.character(fnames))[var])))  
  tdf_seas ▷  
  pivot_longer(-ID, names_to = "date", values_to = "sst_wt_mean")  
}
```

NOAA Temperature Data

Try it on one record:

```
process_raster_zonal(all_fnames[10000])  
  
# A tibble: 101 × 3  
  ID            date      sst_wt_mean  
  <chr>        <chr>          <dbl>  
1 Adriatic Sea d_2009-01-20    13.5  
2 Aegean Sea   d_2009-01-20    15.9  
3 Alboran Sea  d_2009-01-20    14.8  
4 Andaman or Burma Sea d_2009-01-20    27.1  
5 Arabian Sea  d_2009-01-20    26.6  
6 Arafura Sea  d_2009-01-20    29.6  
7 Arctic Ocean d_2009-01-20   -1.74  
8 Baffin Bay   d_2009-01-20   -1.58  
9 Balearic (Iberian Sea) d_2009-01-20    14.0  
10 Bali Sea    d_2009-01-20    28.7  
# i 91 more rows
```

We'll tidy the date later. You can see we get 101 summary records per day.

NOAA Temperature Data

Try it on one *chunk* of records:

```
process_raster_zonal(chunked_fnames[[1]])  
  
# A tibble: 2,525 × 3  
  ID      date    sst_wt_mean  
  <chr>   <chr>     <dbl>  
1 Adriatic Sea d_1981-09-01  23.0  
2 Adriatic Sea d_1981-09-02  23.1  
3 Adriatic Sea d_1981-09-03  22.9  
4 Adriatic Sea d_1981-09-04  22.9  
5 Adriatic Sea d_1981-09-05  22.5  
6 Adriatic Sea d_1981-09-06  22.4  
7 Adriatic Sea d_1981-09-07  22.4  
8 Adriatic Sea d_1981-09-08  22.5  
9 Adriatic Sea d_1981-09-09  22.6  
10 Adriatic Sea d_1981-09-10  22.5  
# i 2,515 more rows
```

NOAA Temperature Data

Now `future_map()` it:

```
tictoc::tic("Big op")
seameans_df ← future_map(chunked_fnames, process_raster_zonal) ▷
  list_rbind() ▷
  mutate(date = str_remove(date, "d_"),
        date = ymd(date),
        year = lubridate::year(date),
        month = lubridate::month(date),
        day = lubridate::day(date),
        yrday = lubridate::yday(date),
        season = season(date)) ▷
  rename(sea = ID)
tictoc::toc()
```

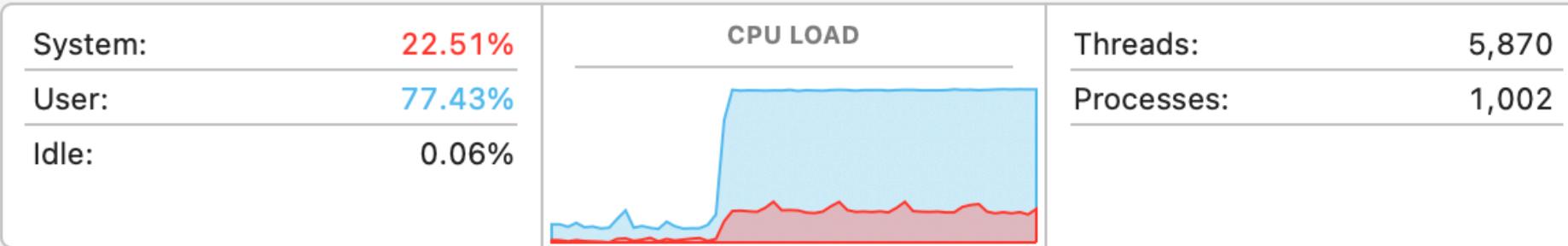
Big op: 48.553 sec elapsed

NOAA Temperature Data

```
seameans_df
```

```
# A tibble: 1,570,449 × 8
  sea      date    sst_wt_mean year month   day yrday season
  <chr>    <date>     <dbl> <dbl> <dbl> <int> <dbl> <fct>
1 Adriatic Sea 1981-09-01     23.0 1981     9     1    244 Summer
2 Adriatic Sea 1981-09-02     23.1 1981     9     2    245 Autumn
3 Adriatic Sea 1981-09-03     22.9 1981     9     3    246 Autumn
4 Adriatic Sea 1981-09-04     22.9 1981     9     4    247 Autumn
5 Adriatic Sea 1981-09-05     22.5 1981     9     5    248 Autumn
6 Adriatic Sea 1981-09-06     22.4 1981     9     6    249 Autumn
7 Adriatic Sea 1981-09-07     22.4 1981     9     7    250 Autumn
8 Adriatic Sea 1981-09-08     22.5 1981     9     8    251 Autumn
9 Adriatic Sea 1981-09-09     22.6 1981     9     9    252 Autumn
10 Adriatic Sea 1981-09-10    22.5 1981     9    10    253 Autumn
# i 1,570,439 more rows
```

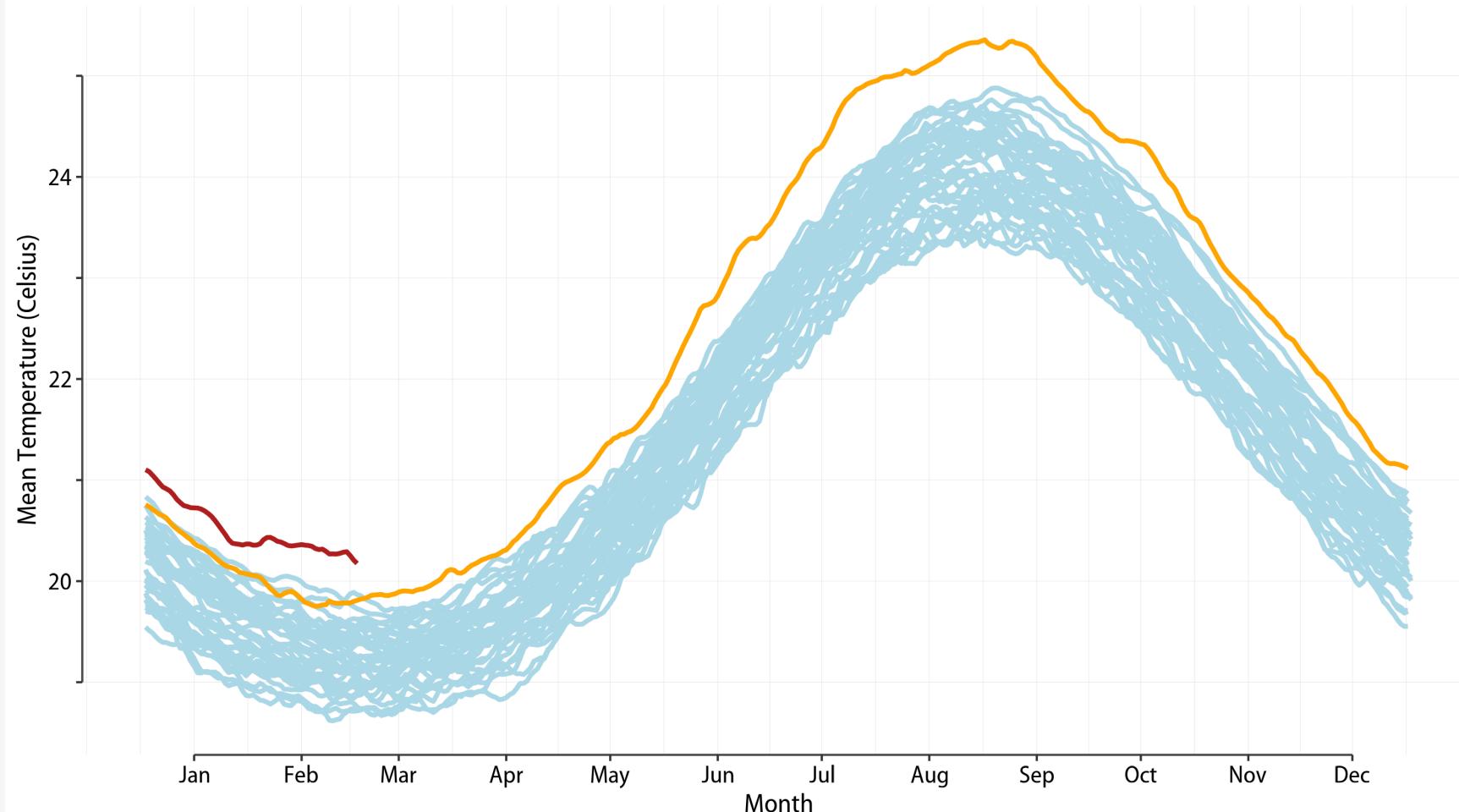
NOAA that's more like it



Mean Daily Sea Surface Temperature, North Atlantic Ocean, 1981-2024

Gridded and weighted NOAA OISST v2.1 estimates

Year 2023 2024 All other years



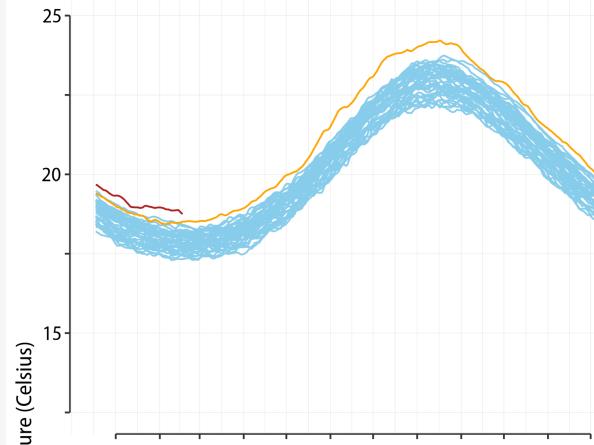
Kieran Healy / @kjhealy

Mean Daily Sea Surface Temperatures, 1981-2024

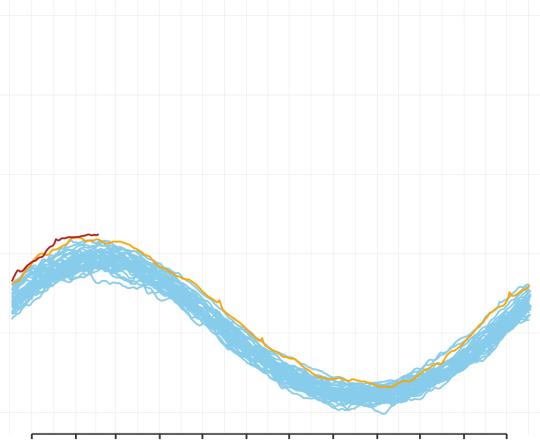
Area-weighted 0.25° grid estimates; NOAA OISST v2.1; IHO Sea Boundaries

Year — 2023 — 2024 — All other years

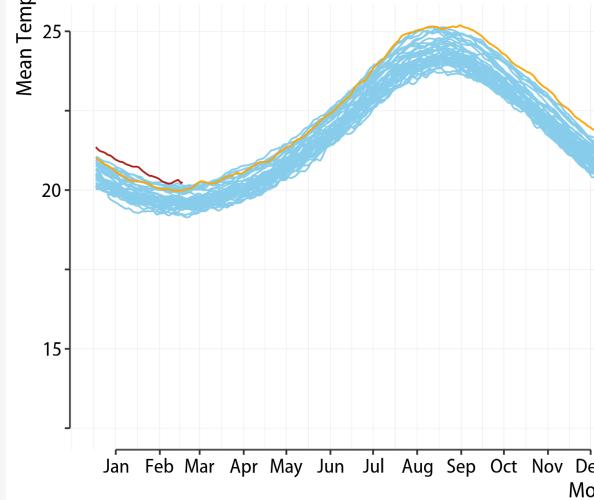
North Atlantic Ocean



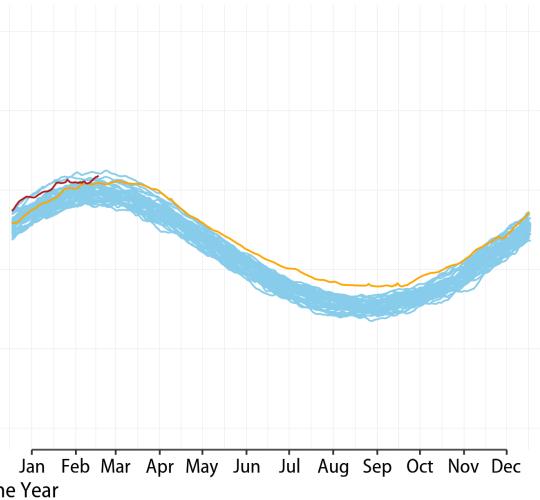
South Atlantic Ocean



North Pacific Ocean



South Pacific Ocean



Data processed with R; Figure made with ggplot by Kieran Healy / @kjhealy

