Data Science for Economists

Lecture 12: Parallel programming

Grant R. McDermott

University of Oregon | EC 607

Contents

Note: This is the third of three lectures on programming. Please take a look at the first two lectures if you haven't yet. Nothing that we will cover here is critically dependent on these earlier lectures. However, I'm going to assume that you have a good understanding of how R functions and environments generally work. Our goal for today is to dramatically speed up our programming tasks by getting them to run in parallel.

Software requirements

R packages

- New: **parallel**, **future**, **future.apply**, **furrr**, **RhpcBLASctl**, **tictoc**
- Already used: **tidyverse**, **data.table**, **pbapply**, **memoise**, **here**, **hrbrthemes**

The code chunk below will install (if necessary) and load all of these packages for you. Note that the **parallel** package is bundled together with the base R installation and should already be on your system. I'm also going to call the `future::plan()` function and set the resolution to "multisession". Don't worry what this means right now — I'll explain in due course — just think of it as a convenient way to set our desired parallel programming behaviour for the rest of this document.

```r
## Load and install the packages that we'll be using today
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tictoc, parallel, pbapply, future, future.apply, tidyverse,
               hrbrthemes, furrr, RhpcBLASctl, memoise, here)
## My preferred ggplot2 plotting theme (optional)
theme_set(hrbrthemes::theme_ipsum())

## Set future::plan() resolution strategy
plan(multisession)
```

Prologue

A few lectures back, we talked about the huge difference that some relatively new packages have made to spatial analysis in R. Complex spatial operations that previously necessitated equally complex spatial objects have been superseded by much simpler and more intuitive tools. If that wasn't good enough, these new tools are also faster. We are going to see something very similar today. Parallel programming is a big and complex topic, with many potential pitfalls. However, software innovations and some amazing new(ish) packages have made it *much* easier and safer to program in parallel.[1]

With that in mind, I'm going to structure today's lecture back-to-front. In particular, I'm going to start with some motivating examples. My primary goal is to demonstrate both the ease and immediate payoff of "going parallel". Only after convincing you of these facts will we get into some of the technical details that were abstracted away behind the scenes. The latter part of the lecture will go over parallel programming in more general terms (i.e. not R-specific) and highlight potential pitfalls that you should be aware of.

Ready? Let's go.

Example 1

Our first motivating example is going to involve the same `slow_square()` function that we saw in the previous lecture:

```r
# library(tidyverse) ## Already loaded

## Emulate slow function
slow_square =
  function(x = 1) {
    x_sq = x^2
    d = tibble(value = x, value_squared = x_sq)
    Sys.sleep(2)
    return(d)
    }
```

Let's iterate over this function using the standard `lapply()` method that we're all familiar with by now. Note that this iteration will be executed in *serial*. I'll use the **tictoc** package (link) to record timing.

```r
# library(tictoc) ## Already loaded

tic()
serial_ex = lapply(1:12, slow_square) %>% bind_rows()
toc()
```

```
## 24.066 sec elapsed
```

As expected, the iteration took about 24 seconds to run because of the enforced break after every sequential iteration (i.e. `Sys.sleep(2)`). On the other hand, this means that we can easily speed things up by iterating in *parallel*.

Before continuing, it's worth pointing out that our abilty to go parallel hinges on the number of CPU cores available to us. The simplest way to obtain this information from R is with the `parallel::detectCores()` function:

```r
# future::availableCores() ## Another option
detectCores()
```

```
## [1] 12
```

So, I have 12 cores to play with on my laptop.[2] Adjust expectations for you own system accordingly.

Okay, back to our example. I'm going to implement the parallel iteration using the **future.apply** package (link) — more on this later. Note that the parameters of the problem are otherwise unchanged.

```r
# library(future.apply)  ## Already loaded
# plan(multisession)     ## Already set above
```

[1] I should emphasise that the R-core team has provided excellent support for parallel programming for over a decade. But there's no question in my mind that the barriers to entry have recently been lowered.

[2] A Dell Precision 5530 running Arch Linux, if you're interested.

```r
tic()
future_ex = future_lapply(1:12, slow_square) %>% bind_rows()
toc(log = TRUE)
```

```
## 4.309 sec elapsed
```

Look at at that: A 6× speedup! Even more impressively, consider how little the syntax changed. I basically just had to tell R that I wanted to implement the iteration in parallel (i.e. **plan(multisession)**) and slightly amend my `lapply()` call (i.e. **future**_lapply()).

Let's confirm that the output is the same.

```r
all_equal(serial_ex, future_ex)
```

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

For those of you who prefer the `purrr::map()` family of functions for iteration and are feeling left out; don't worry. The **furrr** package (link) has you covered. Once again, the syntax for these parallel functions will be very little changed from their serial versions. We simply have to tell R that we want to run things in parallel with `plan(multisession)` and then slightly amend our map call to **future**_map_df**r**().[3]

```r
# library(furrr)      ## Already loaded
# plan(multisession)  ## Already set above

tic()
furrr_ex = future_map_dfr(1:12, slow_square)
toc()
```

```
## 4.404 sec elapsed
```

How easy was that? We hardly had to change our original code and didn't have to pay a cent for all that extra performance.[4] Congratulate yourself on already being such an expert at parallel programming.

Example 2

Our second motivating example will involve a more realistic and slightly more computationally-intensive case: Bootstrapping coefficient values for hypothesis testing.[5] I'll also spend a bit more time talking about the packages we're using and what they're doing.

Start by creating a fake data set (`our_data`) and specifying a bootstrapping function (`bootstrp()`). This function will draw a sample of 10,000 observations from the the data set (with replacement), fit a regression, and then extract the coefficient on the x variable. Note that this coefficient should be around 2 according to our simulation setup.

```r
## Set seed (for reproducibility)
set.seed(1234)
# Set sample size
n = 1e6

## Generate a large data frame of fake data for a regression
our_data =
  tibble(x = rnorm(n), e = rnorm(n)) %>%
  mutate(y = 3 + 2*x + e)

## Function that draws a sample of 10,000 observations, runs a regression and
```

[3] In this particular case, the extra "r" at the end tells future to concatenate the data frames from each iteration by *rows*.

[4] Not to flog a dead horse, but as I pointed out in the very first lecture of this course: Have you seen the price of a Stata/MP license recently? Not to mention the fact that you effectively pay *per* core…

[5] It's an aside, but I'm strongly of the opinion that simulation methods like bootstrapping provide a far more intuitive way to understand statistics. Don't believe me? Watch John Rausser's fantastic talk: "*Statistics Without The Agonizing Pain*". Seriously, watch it.

```
## extracts the coefficient value on the x variable (should be around 2).
bootstrp =
  function(i) {
  ## Sample the data
  sample_data = sample_n(our_data, size = 1e4, replace = TRUE)
  ## Run the regression on our sampled data and extract the extract the x
  ## coefficient.
  x_coef = lm(y ~ x, data = sample_data)$coef[2]
  ## Return value
  return(tibble(x_coef = x_coef))
  }
```

Serial implementation (for comparison)

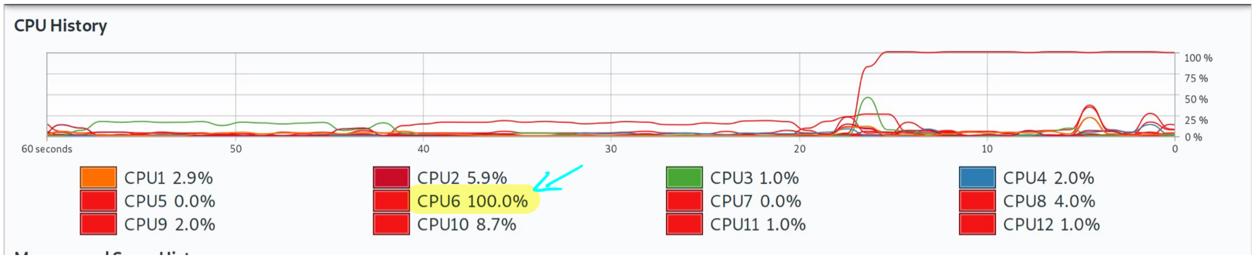
Let's implement the function in serial first to get a benchmark for comparison.

```
set.seed(123L) ## Optional to ensure that the results are the same

## 10,000-iteration simulation
tic()
sim_serial = lapply(1:1e4, bootstrp) %>% bind_rows()
toc(log = TRUE)
```

```
## 37.369 sec elapsed
```

So that took about 35 seconds on my system. Not a huge pain, but let's see if we can do better by switching to a parallel (multicore) implementation. For the record, though here is a screenshot of my system monitor, showing that only one core was being used during this serial version.



Parallel implemention using the future ecosystem

All of the parallel programming that we've been doing so far is built on top of Henrik Bengtsson's amazing **future** package (link). A "future" is basically a very flexible way of evaluating code and output. Among other things, this allows you to switch effortlessly between evaluating code in *serial* or *asynchronously* (i.e. in parallel). You simply have to set your resolution *plan* — "sequential", "multisession", "cluster", etc. — and let future handle the implementation for you.

Here's Henrik describing the core idea in more technical terms:

> In programming, a *future* is an abstraction for a *value* that may be available at some point in the future. The state of a future can either be unresolved or resolved… Exactly how and when futures are resolved depends on what strategy is used to evaluate them. For instance, a future can be resolved using a sequential strategy, which means it is resolved in the current R session. Other strategies may be to resolve futures asynchronously, for instance, by evaluating expressions in parallel on the current machine or concurrently on a compute cluster.

As I've tried to emphasise, **future** is relatively new on the scene. It is certainly not the first or only way to implement parallel processes in R. However, I think that it provides a simple and unified framework that makes it the preeminent choice. What's more, the same commands that we use here will carry over very neatly to more complicated settings involving high-performance computing clusters. We'll experience this first hand when we get to the big data section of the course.

You've probably also noted that keep referring to the "future ecosystem". This is because **future** provides the framework for other packages to implement parallel versions of their functions. The two that I am focusing on today are

1. the **future.apply** package (link), also by Henrik, and
2. the **furrr** package (link), an implementation for **purrr** by Davis Vaughan.

In both cases, we start by setting the plan for resolving the future evaluation (here: `plan(multisession)`). We then call our functions — which involve minor modifications of their serial equivalents — and let future magic take care of everything else.

1) future.apply Here's the `future.apply::future_lapply()` parallel implementation. Note that I'm adding the `future.seed=123L` option to ensure that the results are the same. While not strictly necessary, it's always a good idea to set a random seed with simulations for the sake of reproducibility.

```
# library(future.apply)  ## Already loaded
# plan(multisession)     ## Already set above

## 10,000-iteration simulation
tic()
sim_future = future_lapply(1:1e4, bootstrp, future.seed=123L) %>% bind_rows()
toc()
```

```
## 10.635 sec elapsed
```

2) furrr And here's the `furrr::future_map_dfr()` implementation. Similar to the above, note that I'm only adding the `.options=future_options(seed=123L)` option to ensure that the output is exactly the same.

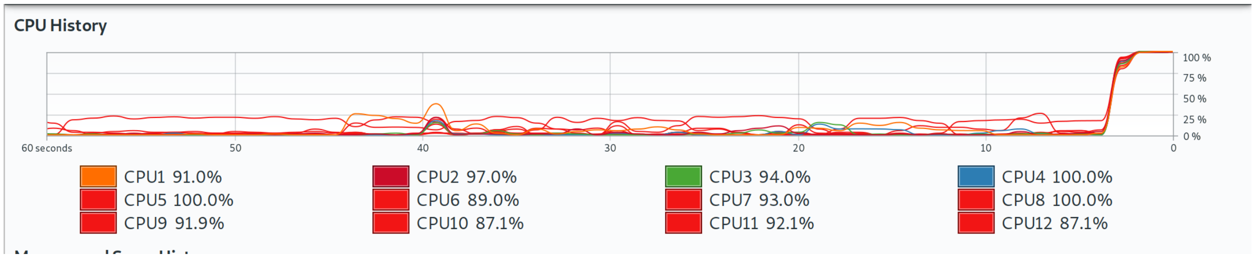
```
# library(furrr)      ## Already loaded
# plan(multisession)  ## Already set above

## 10,000-iteration simulation
tic()
sim_furrr = future_map_dfr(1:1e4, bootstrp, .options = furrr_options(seed=123L))
toc()
```

```
## 9.775 sec elapsed
```

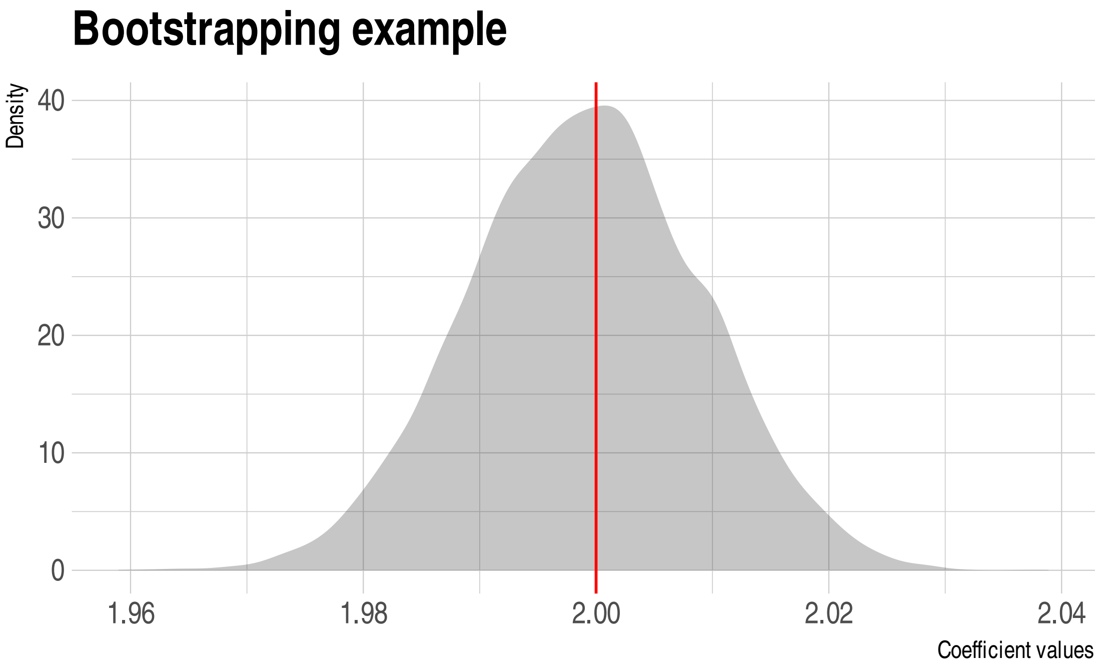
Results

As expected, we dramatically cut down on total computation time by going parallel. Note, however, that the parallel improvements for this example didn't scale linearly with the number of cores on my system (i.e. 12). The reason has to do with the *overhead* of running the parallel implementations — a topic that I cover in more depth toward the bottom of this document. Again, for the record, here is a screenshot showing that all of my cores were now being used during these parallel implementations.



While it wasn't exactly hard work, I think we deserve to see the results of our bootstrapping exercise in nice plot form. I'll use the `sim_furrr` results data frame for this, although it doesn't matter since they're all the same thanks to our our random seed. As you can see, the estimated coefficient values are tightly clustered around our simulated mean of 2.

```
sim_furrr %>%
  ggplot(aes(x_coef)) +
  geom_density(col=NA, fill="gray25", alpha=0.3) +
  geom_vline(xintercept=2, col="red") +
  labs(
    title = "Bootstrapping example",
    x="Coefficient values", y="Density",
    caption = "Notes: Density based on 10,000 draws with sample size of 10,000 each."
    )
```



Bootstrapping example

Notes: Density based on 10,000 draws with sample size of 10,000 each.

Other parallel options

Futures are not the only game in town for parallel programming in R. For example, I'm going to talk about the base R `mclapply` function further below. However, one particular option that I want to mention very briefly is the **pbapply** package (link). As we saw during the first programming lecture, this package provides a lightweight wrapper on the *apply functions that adds a progress bar. However, the package also adds a very convenient option for multicore implementation. You basically just have to add cl=CORES to the call. While it doesn't rely on futures, **pbapply** also takes care of all the OS-specific overhead for you. See here for an interesting discussion on what's happening behind the scenes.

You will need to run this next chunk interactively to see the progress bar.

```
set.seed(123) ## Optional to ensure results are exactly the same.

# library(pbapply) ## Already loaded

## 10,000-iteration simulation
tic()
sim_pblapply = pblapply(1:1e4, bootstrp, cl = parallel::detectCores()) %>% bind_rows()
toc()
```

```
## 8.219 sec elapsed
```

> **Aside:** On the subject of progress bars, check out the **progressr** package (link) for a unified framework that works with all kinds of functions and (a)syncronous backends.

General parallel programming topics

Motivating examples out of the way, let's take a look underneath the hood. I want to emphasise that this section is more "good to know" than "need to know". Even if you take nothing else away from rest of this lecture, you are already well placed to begin implementing parallel functions at a much larger scale.

And yet… while you don't *need* to know the next section in order to program in parallel in R, getting a solid grasp of the basics is valuable. It will give you a better understanding of how parallel programming works in general and help you to appreciate how much **future** and co. are doing behind the scenes for you. It will also help you to understand why the same code runs faster on some systems than others, and avoid some common pitfalls.

Terminology

I'll start by clearing up some terminology.

- **Socket:** The physical connection on your computer that houses the processor. Most work and home computers — even very high-end ones — only have one socket and, thus, one processor. However, they can have multiple cores. Speaking of which…
- **Core:** The part of the processor that actually performs the computation. Back in the day, processors were limited to a single core. However, most modern processors now house multiple cores. Each of these cores can perform entirely separate and independent computational processes.
- **Process:** A single instance of a running task or program (R, Dropbox, etc). A single core can only run one process at a time. However, it may give the appearance of doing more than that by efficiently scheduling between them. Speaking of which…
- **Thread:** A component or subset of a process that can, inter alia, share memory and resources with other threads. We'll return to this idea as it applies to *hyperthreading* in a few paragraphs.
- **Cluster:** A collection of objects that are capable of hosting cores. This could range from a single socket (on your home computer) to an array of servers (on a high-performance computing network).

You may wondering where the much-referenced **CPU** (i.e. central processing unit) fits into all of this. Truth be told, the meaning of CPU has evolved with the advent of new technology like multicore processors. For the purposes of this lecture I will use the following definition:

$$\text{No. of CPUs} = \text{No. of sockets} \times \text{No. of physcial cores} \times \text{No. of threads per core}$$

If nothing else, this is consistent with the way that my Linux system records information about CPU architecure via the lscpu shell command:

```
$ ## Only works on Linux
$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
```

```
## CPU(s):                          12
## Thread(s) per core:              2
## Core(s) per socket:              6
## Socket(s):                       1
```

Note that the headline "CPU(s)" number is the same that I got from running `parallel::detectCores()` earlier (i.e. 12).

A bit more about logical cores and hyperthreading

Logical cores extend or emulate the ability of physical cores to perform additional tasks. The most famous example is Intel's **hyperthreading** technology, which allows a single core to switch very rapidly between two different tasks. This mimics the appearance and performance (albeit to a lesser extent) of an extra physical core. You may find this YouTube video helpful for understanding the difference in more depth, including a nice analogy involving airport security lines.

Taking a step back, you don't have to worry too much about the difference between physical and logical (hyperthreaded) cores for the purpose of this lecture. R doesn't care whether you run a function on a physical core or a logical one. Both will work equally well. (Okay, the latter will be a little slower.) Still, if you are interested in determining the number of physical cores versus logical cores on your system, then there are several ways to this from R. For example, you can use the **RhpcBLASctl** package (link).

```r
# library(RhpcBLASctl) ## Already loaded

get_num_procs() ## No. of all cores (including logical/hyperthreaded)
```

```
## [1] 12
```

```r
get_num_cores() ## No. of physical cores only
```

```
## [1] 6
```

Forking vs Sockets

As I keep saying, it's now incredibly easy to run parallel programs in R. The truth is that it has actually been easy to do so for a long time, but the implementation used to vary by operating system. In particular, simple parallel implementations that worked perfectly well on Linux or Mac didn't work on Windows (which required a lot more overhead). For example, take a look at the help documentation for the `parallel::mclapply()` function, which has been around since 2011. If you did so, you would see a warning that `mclapply()` *"relies on forking and hence is not available on Windows"*.

Now, we clearly didn't encounter any OS-specific problems when we ran the parallel versions of our motivating examples above. The same code worked for everyone, including anyone using Windows. ~~*Loud booing.*~~ What was happening behind the scenes is that the **future** packages automatically handled any complications for us. The parallel functions were being executed in a way that was optimised for each person's OS and R environment.

But what is "forking" and why does it matter what OS I am using anyway? Those are good questions that relate to the method of parallelization (i.e. type of cluster) that your system supports. The short version is that there are basically two ways that code can be parallelized:

- **Forking** works by cloning your entire R environment to each separate core. This includes your data, loaded packages, functions, and any other objects in your current session. This is very efficient because you don't have to worry about reproducing your "master" environment in each "worker" node. Everything is already linked, which means that you aren't duplicating objects in memory. However, forking is not supported on Windows and can also cause problems in an IDE or GUI like RStudio.[6]
- **Parallel sockets** (aka "PSOCKs") work by launching a new R session in each core. This means that your master environment has to be copied over and instantiated separately in each parallel node. This requires greater overhead and causes everything to run slower, since objects will be duplicated across each core. Technically, a PSOCK works by establishing a network (e.g. as if you were connected to a remote cluster), but everything is self-contained on your computer. This approach can be implemented on every system, including Windows, and doesn't create problems for IDEs like RStudio.

I've summarised the differences between the two approaches in the table below.

Forking	PSOCKs
□ Fast and memory efficient.	× Slower and more memory-intensive (than forking).
× Only available for Unix-based systems.	□ Works on every operating system, incl. Windows.
× Potentially unstable in an IDE like RStudio.	□ Fine to use in an IDE like RStudio.

[6] The reason is that shared GUI elements are being shared across child processes. (See the "GUI/embedded environments" section here.) Combined with the fact that the shared memory is *read only*, there is a risk of "cross-contamination" where any modification made to a shared object by one of the worker nodes can propagates to the system.

Forking	PSOCKs

The general rule of thumb is PSOCKs are safer and more universal than forking. This makes them a good default, even if they do come with a potential performance/memory penalty. And, indeed, this is exactly what we did by selecting the **plan(multisession)** resolution strategy. So, now you know where that choice came from.

At the same time, however, I recommend that you consider forking if it is available to you (i.e. you're on Linux or Mac) and you want to maximise performance. In these cases, the solution requires two simple tweaks:

1. Change your resolution plan to **plan(multicore)**, and
2. Run your R script from the terminal using, say, `$ Rscript -e 'rmarkdown::render("mydoc.Rmd", "all")'` or `$ Rscript myfile.R`.

Here's a very simple illustration, using the same setup as Example 1 from earlier.

```r
plan(multicore) ## NB: Only works on Unix!

tic()
future_ex_mc = future_lapply(1:12, slow_square) %>% bind_rows()
toc(log = TRUE)
```

```
## 2.518 sec elapsed
```

The upshot is that the forking version (i.e. `plan(multicore)`) is about 1.6× faster than the PSOCK equivalent (i.e. `plan(multisession)`). But remember: *Caveat emptor*.

For more on forks vs PSOCKS, take a look at the relevant section of the **future** README.

Explicit vs implicit parallelization

Thus far we have only been concerned with *explicit* parallelization. As in, we explicitly told R to run a particular set of commands in parallel. But there is another form of *implicit* parallelization that is equally important to be aware of. In this case, certain low-level functions and operations are automatically run in parallel regardless of whether we "told" R to do so or not. Implicit parallelization can make a big difference to performance, but is not the default behaviour in R. So it has to enabled first (for example, by the package that you're using). Moreover, combining explicit and implicit parallelization can cause problems if you don't take certain precautions. Let's take a look at where implicit parallelization enters the fray.

BLAS/LAPACK

Did you ever wonder how R and other programming languages perform their calculations? For example, how does R actually do things like vector addition, or scalar and matrix multiplication? The answer is **BLAS** (**B**asic **L**inear **A**lgebra **S**uprograms). BLAS are a collection of low-level routines that provide standard building blocks for performing basic vector and matrix operations. These routines are then incoporated in related libraries like **LAPACK** (**L**inear **A**lgebra **Pack**age), which provide their own routines for solving systems of linear equations and linear least squares, calculating eigenvalues, etc. In other words, BLAS and LAPACK provide the linear algebra framework that supports virtually all of statistical and computational programming

R ships with its own BLAS/LAPACK libraries by default. These libraries place a premium on stablility (e.g. common user experience across operating systems). While the default works well enough, you can get *significant* speedups by switching to more optimized libraries such as the Intel Math Kernel Library (MKL) or OpenBLAS. Among other things, these optimised BLAS libraries support multi-threading. So now you are using all your available computer power to, say, solve a matrix.

You can use the `sessionInfo()` command to see which BLAS/LAPACK library you are using. For example, I am using OpenBLAS on this computer:

```
sessionInfo()[c("BLAS", "LAPACK")]
```

```
## $BLAS
## [1] "/usr/lib/libopenblas_haswellp-r0.3.13.so"
##
## $LAPACK
## [1] "/usr/lib/libopenblas_haswellp-r0.3.13.so"
```

Beware resource competition

While this all sounds great — and I certainly recommend taking a look at MKL or OpenBLAS — there is a potential downside. In particular, you risk competing with yourself for computational resources (i.e. memory) if you mix explicit and implicit parallel calls. For instance, if you run explicit multicore functions from within R on a system that has been configured with an optimised BLAS. As Dirk Eddelbuettel succintly puts it in this Stack Overflow thread:

> There is one situation you want to avoid: (1) spreading a task over all *N* cores and (2) having each core work on the task using something like OpenBLAS or MKL with all cores. Because now you have an *N* by *N* contention: each of the N task wants to farm its linear algebra work out to all *N* cores.

Now, I want to emphasise that this conflict rarely matters in my own experience. I use optimised BLAS libraries and run explicit parallel calls all the time in my R scripts. Despite this, I have hardly ever run into a problem. Moreover, when these slowdowns have occured, I've found the effect to be relatively modest.[7] Still, I have read of cases where the effect can be quite dramatic (e.g. here) and so I wanted you to be aware of it all the same.

Luckily, there's also an easy and relatively costless solution: Simply turn off BLAS multi-threading. It turns out this has a negligible impact on performance, since most of the gains from optimised BLAS are actually coming from improved math vectorisation, not multi-threading. (See this post for a detailed discussion.) You can turn off BLAS multi-threading for the current R session via the `RhpcBLASctl::blas_set_num_threads()` function. For example, I sometimes include the following line at the top of an R script:

```
# blas_get_num_procs() ## If you want to find the existing number of BLAS threads
RhpcBLASctl::blas_set_num_threads(1) ## Set BLAS threads to 1 (i.e. turn off multithreading)
```

Since this is only in effect for the current R session, BLAS multithreading will be restored when I restart R.[8]

Library source code

Very briefly, I want to point out that implicit parallelization is automatically invoked by many of the external libraries that we use in R. The good news is that package developers normally take pains to avoid potential resource competition. For instance, consider the message that **data.table** greets us with at load time.

```
library(data.table, warn.conflicts = FALSE)
```

```
## data.table 1.13.6 using 6 threads (see ?getDTthreads).  Latest news: r-datatable.com
```

If you follow the suggestion and look at the `?getDTthreads` help documentation, you'll find an informative (and reassuring) discussion of its approach here:

> `data.table` automatically switches to single threaded mode upon fork (the mechanism used by `parallel::mclapply` and the `foreach` package). Otherwise, nested parallelism would very likely overload your CPUs and result in much slower execution. As data.table becomes more parallel internally, we expect explicit user parallelism to be needed less often…

A final point on this topic, riffing off the quoted text, is that packages like **data.table** implement their parallel operations at the source-code level, i.e. in C(++) and other compiled languages. So they are likely to be more efficient than the equivalent

[7] The major cost appears to be the unnecessary duplication of objects in memory.

[8] I could also reinstate the original behaviour in the same session by running `blas_set_num_threads(parallel::detectCores())`. You can turn off multithreading as the default mode by altering the configuration file when you first build/install your preferred BLAS library. However, that's both complicated and unnecessarily restrictive in my view.

explicit parallel calls that you might make. It's not that you can't combine, say, **future** and **data.table** (I do this often). But you should know that trying to do better than the latter's default operations may be a fool's errand.

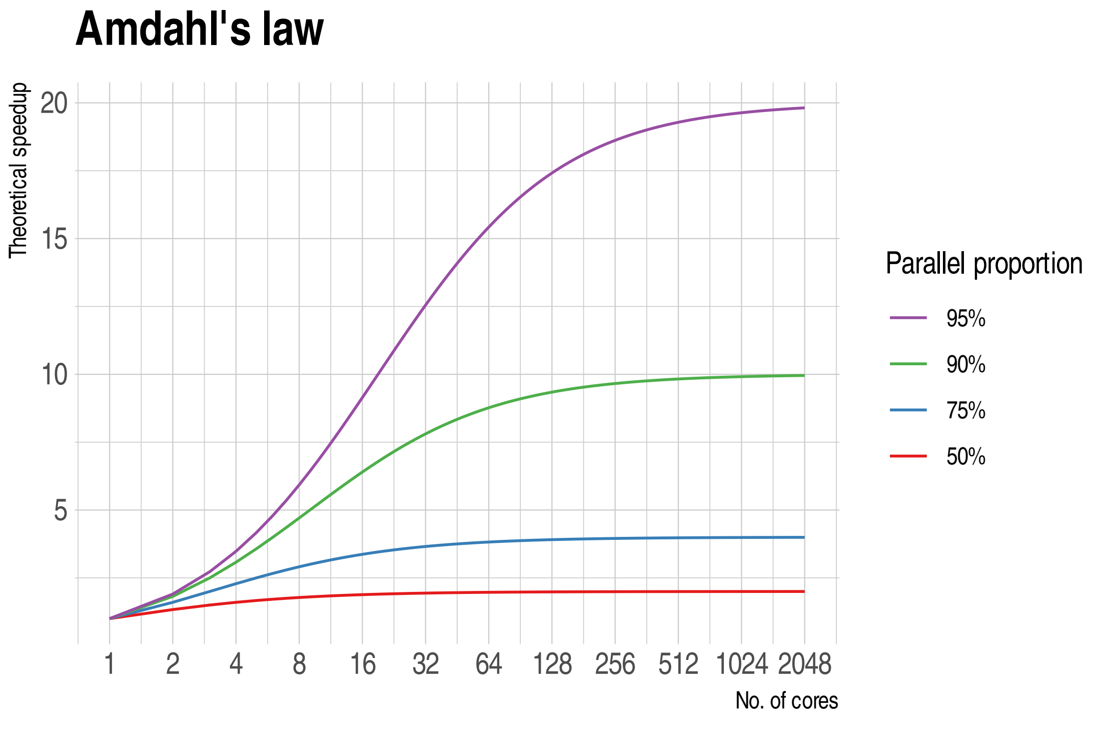
Miscellaneous

When should I go parallel?

The short answer is that you want to invoke the multicore option whenever you are faced with a so-called "embarrassingly parallel" problem. You can click on that link for a longer description, but the key idea is that these computational problems are easy to break up into smaller chunks. You likely have such a case if the potential code chunks are independent and do not need to communicate in any way. Classic examples include bootstrapping (since each regression or resampling iteration is drawn independently) and Markov chain Monte Carlo (i.e. MCMC).

Having said that, there are limitations to the gains that can be had from parallelization. Most obviously, there is the computational overhead associated with splitting up the problem, tracking the individual nodes, and then bringing everything back into a single result. This can be regarded as an issue largely affecting shorter and smaller computations. In other words, the overhead component of the problem tends to diminish in relative size as the overall computation time increases.

On the opposite end of the spectrum, there is Amdahl's law (generalised as Gustafson's law). This formalises the intuitive idea that there are diminishing returns to parallelization, depending on the proportion of your code that can be run in parallel. A case in point is Bayesian MCMC routines, which typically include a fixed "burn-in" period regardless of how many parallel chains are being run in parallel.



How many cores should I use?

If you look this question up online, you'll find that most people recommend using `detectCores()-1`. This advice stems from the idea that you probably want to reserve one core for other tasks, such as running your web browser or word processor. While I don't disagree, I typically use all available cores for my parallel computations. For one thing, I do most of my heavy computational work in the cloud (i.e. on a server or virtual machine). So keeping some computational power in reserve doesn't make sense. Second, when I am working locally, I've gotten into the habit of closing all other applications while a parallel function is running. Your mileage may vary, though. (And remember the possible diminishing returns brought on by Amdahl's law). FWIW, calling `plan(multisession)` or `plan(multicore)` automatically default to using all your cores. You can change that by running, say, `plan(multisession(workers = detectCores()-1))`.

Fault tolerance (error catching, caching, etc.)

In my experience, the worst thing about parallel computation is that it is very sensitive to failure in any one of its nodes. An especially frustrating example is the tendency of parallel functions to ignore/hide critical errors up until the very end when they are supposed to return output. ("Oh, so you encountered a critical error several hours ago, but just decided to continue for fun anyway? Thanks!") Luckily, all of the defensive programming tools that we practiced in the previous lecture — catching user errors and caching intermediate results — carry over perfectly to their parallel equivalents. Just make sure that you use a persistent cache.

Challenge: Prove this to yourself by running a parallel version of the cached iteration that we practiced last time. Specifically, you should recreate the `mem_square_verbose()` function, which in turn relies on the `mem_square_persistent()` function.[9] You should then be able to run `future_map_dfr(1:10, mem_square_verbose)` and it will automatically return the previously cached results. After that, try `future_map_dfr(1:24, mem_square_verbose)` and see what happens.

Random number generation

Random number generation (RNG) can become problematic in parallel computations (whether trying to ensure the same of different RNG across processes). R has various safeguards against this and future automatically handles RNG via the `future.seed` argument. We saw an explicit example of this in Example 2 above.

Parallel regression

A number of regression packages in R are optimised to run in parallel. For example, the superb **fixest** package (link) that we saw in the lecture on regression analysis will automatically invoke multicore capabilities when fitting high dimensional fixed effects models. The many Bayesian packages in R are also all capable of — and, indeed, expected to — fit regression models by running their MCMC chains in parallel. For example, **RStan** (link). Finally, you may be interested in the **partools** package (link), which provides convenient aliases for running a variety of statistical models and algorithms in parallel.

CPUs vs GPUs

Graphical Processing Units, or GPUs, are specialised chipsets that were originaly built to perform the heavy lifting associated with rendering graphics. It's important to realise that not all computers have GPUs. Most laptops come with so-called integrated graphics, which basically means that the same processor is performing both regular and graphic-rendering tasks. However, gaming and other high-end laptops (and many desktop computers) include a dedicated GPU card. For example, the Dell Precision 5530 that I'm writing these lecture notes on has a hybrid graphics setup with two cards: 1) an integrated Intel GPU (UHD 630) and 2) a discrete NVIDIA Quadro P2000.

So why am I telling you this? Well, it turns out that GPUs also excel at non-graphic computation tasks. The same processing power needed to perform the millions of parallel calculations for rendering 3-D games or architectural software, can be put to use on scientific problems. How exactly this was discovered involves an interesting backstory of supercomputers being built with Playstations. (Google it.) But the short version is that modern GPUs comprise *thousands* of cores that can be run in parallel. Or, as my colleague David Evans once memorably described it to me: "GPUs are basically just really, really good at doing linear algebra."

Still, that's about as much as I want to say about GPUs for now. Installing and maintaining a working GPU setup for scientific purposes is a much more complex task. (And, frankly, overkill for the vast majority of econometric or data science needs.) We may revisit the topic when we get to the machine learning section of the course in a few weeks.[10] Thus, and while the general concepts carry over, everything that we've covered today is limited to CPUs.

[9] To clarify: The verbose option simply provides helpful real-time feedback to us. However, the underlying persistent cache location — provided in this case by `mem_square_persistent()` — is necessary whenever you want to use a memoised function in the futures framework.

[10] Advanced machine learning techniques like deep learning are particularly performance-dependent on GPUs.

Monitoring multicore performance

Bash-compatible shells should come with the built-in `top` command, which provides a real-time view of running processes and resource consumption. (Pro-tip: Hit "1" to view processes across individual cores and "q" to quit.) An enhanced alternative that I really like and use all the time is **htop**, which is available on both Linux and Mac. (Windows users can install `htop` on the WSL that we covered way back in the shell lecture.). It's entirely up to you whether you want to install it. Your operating system almost certainly provides built-in tools for monitoring processes and resource usage (e.g. System Monitor). However, I wanted to flag `htop` before we get to the big data section of the course. We'll all be connecting to remote Linux servers at that point and a shell-based (i.e. non-GUI) process monitor will prove very handy for tracking resource use.

Further resources

- Dirk Eddelbuettel provides the authoritative reference on this topic in his review paper, *Parallel Computing With R: A Brief Review* (pre-print).
- Beyond Dirk's article, I'd argue that the starting point for further reading should be the **future** vignettes (one, two, three, four, five). There's a lot in there, so feel free to pick and choose.
- Similarly, the furrr package vignette is very informative (and concise).
- The parallel package vignette provides a very good overview, not only its own purpose, but of parallel programming in general. Particular attention is paid to the steps needed to ensure a stable R environment (e.g. across operating systems).
- Finally, there a number of resources online that detail older parallel programming methods in R (`foreach`, `mclapply`, `parLapply` `snow`, etc.). While these methods have clearly been superseded by the future package ecosystem in my mind, there is still a lot of valuable information to be gleaned from understanding them. Two of my favourite resources in this regard are: How-to go parallel in R (Max Gordon) and Beyond Single-Core R (Jonathan Dursi).