Parallel processing

Chris Paciorek

2022-09-30

Table of contents

1. Some scenarios for parallelization **2**
Embarrassingly parallel (EP) problems . . . 3

2. Overview of parallel processing **3**
Computer architecture . . . 3
Some useful terminology: . . . 4
Distributed vs. shared memory . . . 4
Shared memory . . . 4
Distributed memory . . . 5
Some other approaches to parallel processing . . . 5
GPUs . . . 5
Spark and Hadoop . . . 6
Cloud computing . . . 6

3. Parallelization strategies **6**

4. Introduction to the future package **7**
Overview: Futures and the R future package . . . 7
Overview of parallel backends . . . 8
Accessing variables and workers in the worker processes . . . 8

5. Illustrating the principles in specific case studies **9**
Scenario 1: one model fit . . . 9
Scenario 1A: . . . 9
Scenario 1B: . . . 10
Scenario 2: three different prediction methods on your data . . . 11
Scenario 3: 10-fold CV and 10 or fewer cores . . . 12
Using a parallelized for loop with `foreach` . . . 13
Alternatively using parallel apply statements . . . 14
Scenario 4: parallelizing over prediction methods . . . 15
Scenario 5: 10-fold CV across multiple methods with many more than 10 cores . . . 17
Scenario 5A: nested parallelization . . . 17
Scenario 5B: Parallelizing across multiple nodes . . . 18

Scenario 6: Stratified analysis on a very large dataset . . . 19
Scenario 7: Simulation study with n=1000 replicates: parallel random number generation . . 20

6. Additional details and topics (optional) 21
Setting the number of threads (cores used) in threaded code (including parallel linear algebra in R) . . . 21
Important warnings about use of threaded BLAS . . . 21
Speed and threaded BLAS . . . 21
Conflicts between openBLAS and various R functionality . . . 22

7. Using Dask in Python 22
Scheduler . . . 22
Parallel map . . . 23
Futures / delayed evaluation . . . 23
Final notes . . . 24

PDF

References:

- Tutorial on parallel processing using Python's Dask and R's future packages

This unit will be fairly Linux-focused as most serious parallel computation is done on systems where some variant of Linux is running. The single-machine parallelization discussed here should work on Macs and Windows, but some of the details of what is happening under the hood are different for Windows.

1. Some scenarios for parallelization

- You need to fit a single statistical/machine learning model, such as a random forest or regression model, to your data.
- You need to fit three different statistical/machine learning models to your data.
- You are running a prediction method on 10 cross-validation folds, possibly using multiple statistical/machine learning models to do prediction.
- You are running an ensemble prediction method such as *SuperLearner* or *Bayesian model averaging* over 10 cross-validation folds, with 30 statistical/machine learning methods used for each fold.
- You are running stratified analyses on a very large dataset (e.g., running regression models once for each subgroup within a dataset).
- You are running a simulation study with n=1000 replicates. Each replicate involves fitting 10 statistical/machine learning methods.

Given you are in such a situation, can you do things in parallel? Can you do it on your laptop or a single computer? Will it be useful (i.e., faster or provide access to sufficient memory) to use multiple computers, such as multiple nodes in a Linux cluster?

All of the functionality discussed in this Unit applies ONLY if the iterations/loops of your calculations can be done completely separately and do not depend on one another; i.e., you can do the computa-

tion as separate processes without communication between the processes. This scenario is called an *embarrassingly parallel* computation.

Embarrassingly parallel (EP) problems

An EP problem is one that can be solved by doing independent computations in separate processes without communication between the processes. You can get the answer by doing separate tasks and then collecting the results. Examples in statistics include

1. simulations with many independent replicates
2. bootstrapping
3. stratified analyses
4. random forests
5. cross-validation.

The standard setup is that we have the same code running on different datasets. (Note that different processes may need different random number streams, as we will discuss in the Simulation Unit.)

To do parallel processing in this context, you need to have control of multiple processes. Note that on a shared system with queueing/scheduling software set up, this will generally mean requesting access to a certain number of processors and then running your job in such a way that you use multiple processors.

In general, except for some modest overhead, an EP problem can ideally be solved with $1/p$ the amount of time for the non-parallel implementation, given p CPUs. This gives us a speedup of p, which is called linear speedup (basically anytime the speedup is of the form kp for some constant k).

2. Overview of parallel processing

Computer architecture

Computers now come with multiple processors for doing computation. Basically, physical constraints have made it harder to keep increasing the speed of individual processors, so the chip industry is now putting multiple processing units in a given computer and trying/hoping to rely on implementing computations in a way that takes advantage of the multiple processors.

Everyday personal computers usually have more than one processor (more than one chip) and on a given processor, often have more than one core (multi-core). A multi-core processor has multiple processors on a single computer chip. On personal computers, all the processors and cores share the same memory.

Supercomputers and computer clusters generally have tens, hundreds, or thousands of 'nodes', linked by a fast local network. Each node is essentially a computer with its own processor(s) and memory. Memory is local to each node (distributed memory). One basic principle is that communication between a processor and its memory is much faster than communication between processors with different memory. An example of a modern supercomputer is the Cori supercomputer at Lawrence Berkeley National Lab, which has 12,076 nodes, and a total of 735,200 cores. Each node has either 96 or 128 GB of memory for a total of 1.3 PB of memory.

For our purposes, there is little practical distinction between multi-processor and multi-core situations. The main issue is whether processes share memory or not. In general, I won't distinguish between cores and processors. We'll just focus on the number of cores on given personal computer or a given node in a cluster.

Some useful terminology:

- *cores*: We'll use this term to mean the different processing units available on a single machine or node of a cluster.
- *nodes*: We'll use this term to mean the different computers, each with their own distinct memory, that make up a cluster or supercomputer.
- *processes*: instances of a program(s) executing on a machine; multiple processes may be executing at once. A given program may start up multiple processes at once. Ideally we have no more processes than cores on a node.
- *workers*: the individual processes that are carrying out the (parallelized) computation. We'll use *worker* and *process* interchangeably.
- *tasks*: individual units of computation; one or more tasks will be executed by a given process on a given core.
- *threads*: multiple paths of execution within a single process; the operating system sees the threads as a single process, but one can think of them as 'lightweight' processes. Ideally when considering the processes and their threads, we would the same number of cores as we have processes and threads combined.
- *forking*: child processes are spawned that are identical to the parent, but with different process IDs and their own memory. In some cases if objects are not changed, the objects in the child process may refer back to the original objects in the original process, avoiding making copies.
- *sockets*: some of R's parallel functionality involves creating new R processes (e.g., starting processes via `Rscript`) and communicating with them via a communication technology called sockets.
- *scheduler*: a program that manages users' jobs on a cluster. *Slurm* is a commonly used scheduler.
- *load-balanced*: when all the cores that are part of a computation are busy for the entire period of time the computation is running.

Distributed vs. shared memory

There are two basic flavors of parallel processing (leaving aside GPUs): distributed memory and shared memory. With shared memory, multiple processors (which I'll call cores for the rest of this document) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. You can think of each node as a separate computer connected by a fast network.

Shared memory

For shared memory parallelism, each core is accessing the same memory so there is no need to pass information (in the form of messages) between different machines. However, unless one is using threading (or in some cases when one has processes created by forking), objects will still be copied when creating new processes to do the work in parallel. With threaded computations, multiple threads can access object(s) without making explicit copies. But in some programming contexts one needs to be careful

that the threads on different cores doesn't mistakenly overwrite places in memory that are used by other cores (this is not an issue in R).

We'll cover two types of shared memory parallelism approaches in this unit:

- threaded linear algebra
- multicore functionality

Threading

Threads are multiple paths of execution within a single process. If you are monitoring CPU usage (such as with `top` in Linux or Mac) and watching a job that is executing threaded code, you'll see the process using more than 100% of CPU. When this occurs, the process is using multiple cores, although it appears as a single process rather than as multiple processes.

Note that this is a different notion than a processor that is hyperthreaded. With hyperthreading a single core appears as two cores to the operating system.

Distributed memory

Parallel programming for distributed memory parallelism requires passing messages between the different nodes. The standard protocol for doing this is MPI, of which there are various versions, including `openMPI`.

While there are various R (e.g., `Rmpi` and the `pbdR` packages) and Python packages that use MPI behind the scenes, we'll only cover distributed memory parallelization via the `future` package and `Dask`, which don't use MPI.

Some other approaches to parallel processing

GPUs

GPUs (Graphics Processing Units) are processing units originally designed for rendering graphics on a computer quickly. This is done by having a large number of simple processing units for massively parallel calculation. The idea of general purpose GPU (GPGPU) computing is to exploit this capability for general computation.

Most researchers don't program for a GPU directly but rather use software (often machine learning software such as Tensorflow or PyTorch, or other software that automatically uses the GPU such as JAX) that has been programmed to take advantage of a GPU if one is available. The computations that run on the GPU are run in GPU *kernels*, which are functions that are launched on the GPU. The overall workflow runs on the CPU and then particular (usually computationally-intensive tasks for which parallelization is helpful) tasks are handed off to the GPU. GPUs and similar devices (e.g., TPUs) are often called "co-processors" in recognition of this style of workflow.

The memory on a GPU is distinct from main memory on the computer, so when writing code that will use the GPU, one generally wants to avoid having large amounts of data needing to be transferred back and forth between main (CPU) memory and GPU memory. Also, since there is overhead in launching a GPU kernel, one wants to avoid launching a lot of kernels relative to the amount of work being done by each kernel.

Spark and Hadoop

Spark and Hadoop are systems for implementing computations in a distributed memory environment, using the MapReduce approach, as discussed in Unit 7.

Cloud computing

Amazon (Amazon Web Services' EC2 service), Google (Google Cloud Platform's Compute Engine service) and Microsoft (Azure) offer computing through the cloud. The basic idea is that they rent out their servers on a pay-as-you-go basis. You get access to a virtual machine that can run various versions of Linux or Microsoft Windows server and where you choose the number of processing cores you want. You configure the virtual machine with the applications, libraries, and data you need and then treat the virtual machine as if it were a physical machine that you log into as usual. You can also assemble multiple virtual machines into your own virtual cluster and use platforms such as Spark on the cloud provider's virtual machines.

3. Parallelization strategies

Some of the considerations that apply when thinking about how effective a given parallelization approach will be include:

- the amount of memory that will be used by the various processes,
- the amount of communication that needs to happen – how much data will need to be passed between processes,
- the latency of any communication - how much delay/lag is there in sending data between processes or starting up a worker process, and
- to what extent do processes have to wait for other processes to finish before they can do their next step.

The following are some basic principles/suggestions for how to parallelize your computation.

- Should I use one machine/node or many machines/nodes?
  - If you can do your computation on the cores of a single node using shared memory, that will be faster than using the same number of cores (or even somewhat more cores) across multiple nodes. Similarly, jobs with a lot of data/high memory requirements that one might think of as requiring Spark or Hadoop may in some cases be much faster if you can find a single machine with a lot of memory.
  - That said, if you would run out of memory on a single node, then you'll need to use distributed memory.
- What level or dimension should I parallelize over?
  - If you have nested loops, you generally only want to parallelize at one level of the code. That said, in this unit we'll see some tools for parallelizing at multiple levels. Keep in mind whether your linear algebra is being threaded. Often you will want to parallelize over a loop and not use threaded linear algebra within the iterations of the loop.
  - Often it makes sense to parallelize the outer loop when you have nested loops.
  - You generally want to parallelize in such a way that your code is load-balanced and does not involve too much communication.
- How do I balance communication overhead with keeping my cores busy?

  - If you have very few tasks, particularly if the tasks take different amounts of time, often some processors will be idle and your code poorly load-balanced.
  - If you have very many tasks and each one takes little time, the overhead of starting and stopping the tasks will reduce efficiency.
- Should multiple tasks be pre-assigned (statically assigned) to a process (i.e., a worker) (sometimes called *prescheduling*) or should tasks be assigned dynamically as previous tasks finish?
  - To illustrate the difference, suppose you have 6 tasks and 3 workers. If the tasks are pre-assigned, worker 1 might be assigned tasks 1 and 4 at the start, worker 2 assigned tasks 2 and 5, and worker 3 assigned tasks 3 and 6. If the tasks are dynamically assigned, worker 1 would be assigned task 1, worker 2 task 2, and worker 3 task 3. Then whichever worker finishes their task first (it woudn't necessarily be worker 1) would be assigned task 4 and so on.
  - Basically if you have many tasks that each take similar time, you want to preschedule the tasks to reduce communication. If you have few tasks or tasks with highly variable completion times, you don't want to preschedule, to improve load-balancing.
  - For R in particular, some of R's parallel functions allow you to say whether the tasks should be prescheduled. In the future package, `future_lapply` has arguments `future.scheduling` and `future.chunk.size`. Similarly, there is the `mc.preschedule` argument in `mclapply()`.

4. Introduction to the future package

Before we illustrate implementation of various kinds of parallelization, I'll give an overview of the `future` package, which we'll use for many of the implementations. The future package has been developed over the last few years and provides some nice functionality that is easier to use and more cohesive than the various other approaches to parallelization in R.

Other approaches include `parallel::parLapply`, `parallel::mclapply`, the use of `foreach` without `future`, and the `partools` package. The `partools` package is interesting. It tries to take the parts of Spark/Hadoop most relevant for statistics-related work – a distributed file system and distributed data objects – and discard the parts that are a pain/not useful – fault tolerance when using many, many nodes/machines.

Overview: Futures and the R future package

What is a *future*? It's basically a flag used to tag a given operation such that when and where that operation is carried out is controlled at a higher level. If there are multiple operations tagged then this allows for parallelization across those operations.

According to Henrik Bengtsson (the `future` package developer) and those who developed the concept:

- a future is an abstraction for a value that will be available later
- the value is the result of an evaluated expression
- the state of a future is either unresolved or resolved

Why use futures? The `future` package allows one to write one's computational code without hard-coding whether or how parallelization would be done. Instead one writes the code in a generic way and at the beginning of one's code sets the 'plan' for how the parallel computation should be done

given the computational resources available. Simply changing the 'plan' changes how parallelization is done for any given run of the code.

More concisely, the key ideas are:

- Separate what to parallelize from how and where the parallelization is actually carried out.
- Different users can run the same code on different computational resources (without touching the actual code that does the computation).

Overview of parallel backends

One uses `plan()` to control how parallelization is done, including what machine(s) to use and how many cores on each machine to use.

For example,

```
plan(multiprocess)
## spreads work across multiple cores
# alternatively, one can also control number of workers
plan(multiprocess, workers = 4)
```

This table gives an overview of the different plans.

Type	Description	Multi-node	Copies of objects made?
multisession	uses additional R sessions as the workers	no	yes
multicore	uses forked R processes as the workers	no	not if object not modified
remote	uses an R sessions on another machine	yes	yes
cluster	uses R sessions on other machine(s)	yes	yes

Accessing variables and workers in the worker processes

The future package usually does a good job of identifying the packages and (global) variables you use in your parallelized code and loading those packages on the workers and copying necessary variables to the workers. It uses the `globals` package to do this.

Here's a toy example that shows that `n` and `MASS::geyser` are automatically available in the worker processes.

```
library(future)
library(future.apply)

plan(multisession)

library(MASS)
n <- nrow(geyser)
```

```
myfun <- function(idx) {
   # geyser is in MASS package
   return(sum(geyser$duration) / n)
}

future_sapply(1:5, myfun)
```

```
[1] 3.460814 3.460814 3.460814 3.460814 3.460814
```

In other contexts in R (or other languages) you may need to explicitly copy objects to the workers (or load packages on the workers). This is sometimes called *exporting* variables.

5. Illustrating the principles in specific case studies

Scenario 1: one model fit

Scenario: You need to fit a single statistical/machine learning model, such as a random forest or regression model, to your data.

Scenario 1A:

A given method may have been written to use parallelization and you simply need to figure out how to invoke the method for it to use multiple cores.

For example the documentation for the `randomForest` package doesn't indicate it can use multiple cores, but the `ranger` package can – note the `num.threads` argument.

```
args(ranger::ranger)
```

```
function (formula = NULL, data = NULL, num.trees = 500, mtry = NULL,
    importance = "none", write.forest = TRUE, probability = FALSE,
    min.node.size = NULL, max.depth = NULL, replace = TRUE, sample.fraction = ifelse(replace,
        1, 0.632), case.weights = NULL, class.weights = NULL,
    splitrule = NULL, num.random.splits = 1, alpha = 0.5, minprop = 0.1,
    split.select.weights = NULL, always.split.variables = NULL,
    respect.unordered.factors = NULL, scale.permutation.importance = FALSE,
    local.importance = FALSE, regularization.factor = 1, regularization.usedepth = FALSE,
    keep.inbag = FALSE, inbag = NULL, holdout = FALSE, quantreg = FALSE,
    oob.error = TRUE, num.threads = NULL, save.memory = FALSE,
    verbose = TRUE, seed = NULL, dependent.variable.name = NULL,
    status.variable.name = NULL, classification = NULL, x = NULL,
    y = NULL, ...)
NULL
```

Scenario 1B:

If a method does linear algebra computations on large matrices/vectors, R can call out to parallelized linear algebra packages (the BLAS and LAPACK).

The BLAS is the library of basic linear algebra operations (written in Fortran or C). A fast BLAS can greatly speed up linear algebra in R relative to the default BLAS that comes with R. Some fast BLAS libraries are

- Intel's *MKL*; available for educational use for free
- *OpenBLAS*; open source and free
- *vecLib* for Macs; provided with your Mac

In addition to being fast when used on a single core, all of these BLAS libraries are threaded - if your computer has multiple cores and there are free resources, your linear algebra will use multiple cores, provided your program is linked against the threaded BLAS installed on your machine and provided the environment variable OMP_NUM_THREADS is not set to one. (Macs make use of VECLIB_MAXIMUM_THREADS rather than OMP_NUM_THREADS.)

Threading in R is limited to linear algebra, provided R is linked against a threaded BLAS.

Here's some code that illustrates the speed of using a threaded BLAS:

```
library(RhpcBLASctl)
x <- matrix(rnorm(5000^2), 5000)

blas_set_num_threads(4)
system.time({
   x <- crossprod(x)
   U <- chol(x)
})

##    user  system elapsed
##   8.316   2.260   2.692

blas_set_num_threads(1)
system.time({
   x <- crossprod(x)
   U <- chol(x)
})

##    user  system elapsed
##   6.360   0.036   6.399
```

Here the elapsed time indicates that using four threads gave us a two-three times (2-3x) speedup in terms of real time, while the user time indicates that the threaded calculation took a bit more total processing time (combining time across all processors) because of the overhead of using multiple threads.

Note that the code also illustrates use of an R package that can control the number of threads from within R, but you could also have set OMP_NUM_THREADS before starting R.

To use an optimized BLAS with R, talk to your systems administrator, see Section A.3 of the R Installation and Administration Manual, or see these instructions to use vecLib BLAS from Apple's Accelerate framework on your own Mac.

It's also possible to use an optimized BLAS with Python's `numpy` and `scipy` packages, on either Linux or using the Mac's *vecLib* BLAS. Details will depend on how you install Python, numpy, and scipy.

Scenario 2: three different prediction methods on your data

Scenario: You need to fit three different statistical/machine learning models to your data.

What are some options?

- use one core per model
- if you have rather more than three cores, apply the ideas here combined with Scenario 1 above - with access to a cluster and parallelized implementations of each model, you might use one node per model

```
library(future)
ntasks <- 3
plan(multisession, workers = ntasks)

n <- 10000000
system.time({
    fut_p <- future(mean(rnorm(n)), seed = TRUE)
    fut_q <- future(mean(rgamma(n, shape = 1)), seed = TRUE)
    fut_s <- future(mean(rt(n, df = 3)), seed = TRUE)
        p <- value(fut_p)
        q <- value(fut_q)
        s <- value(fut_s)
})
```

```
   user  system elapsed
  0.013   0.002   2.197
```

```
system.time({
    p <- mean(rnorm(n))
    q <- mean(rgamma(n, shape = 1))
    s <- mean(rt(n, df = 3))
})
```

```
   user  system elapsed
  3.190   0.034   3.226
```

Question: Why might this not have shown a perfect three-fold speedup?

If we look at the future object (e.g., `fut_p`), we see that by default lazy evaluation is off and that the code executes *asynchronously*. Let's consider these ideas in more detail. The future will start being evaluated right away – this is non-lazy evaluation. The future will execute asynchronously, which means that the worker process will evaluate the future (in the background from the perspective of the main process) while the main process can continue doing other things, in particular interacting with the user. This asynchronous evaluation is also called a *non-blocking* call because execution of the task in the worker process does not block things from happening in the main process. However the call to `value()` is *synchronous* (and is a *blocking* call) because it needs to returns a result, so control of the session does not return to the user until the value is available (i.e., once the future is done being evaluated).

One can change the future to use lazy evaluation.

You could also have used tools like `foreach` and `future_lapply` here as well, as we'll discuss next.

Scenario 3: 10-fold CV and 10 or fewer cores

Scenario: You are running a prediction method on 10 cross-validation folds.

This illustrates the idea of running some number of tasks using the cores available on a single machine.

Here I'll illustrate parallel looping, using this simulated dataset and basic use of `randomForest()`.

```
randomForest 4.7-1
```

```
Type rfNews() to see new features/changes/bug fixes.
```

```
cvFit <- function(foldIdx, folds, Y, X, loadLib = FALSE) {
    if(loadLib)
        library(randomForest)
    out <- randomForest(y = Y[folds != foldIdx],
                        x = X[folds != foldIdx, ],
                        xtest = X[folds == foldIdx, ])
    return(out$test$predicted)
}

set.seed(23432)
## training set
n <- 1000
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)
nFolds <- 10
folds <- sample(rep(seq_len(nFolds), each = n/nFolds), replace = FALSE)
```

Using a parallelized for loop with `foreach`

The foreach package provides a `foreach` command that allows you to do this easily. foreach can use a variety of parallel "back-ends", of which the future package is one back-end (via the `doFuture` package) that provides a lot of flexibility in what computational resources are used via `plan()`. For our purposes here, we'll focus on using shared memory cores.

Note that `foreach` also provides functionality for collecting and managing the results to avoid some of the bookkeeping you would need to do if writing your own standard for loop. The result of `foreach` will generally be a list, unless we request the results be combined in different way, using the `.combine` argument.

```
library(doFuture)
library(doRNG)
nCores <- 2
plan(multisession, workers = nCores)
registerDoFuture()

## Use of %dorng% from doRNG relates to parallel random number generation.
## We'll see more in Unit 10 (Simulation)
## If not using random number generation, people usually use %dopar%.

result <- foreach(i = seq_len(nFolds)) %dorng% {
    cat('Starting ', i, 'th job.\n', sep = '')
    output <- cvFit(i, folds, Y, X)
    cat('Finishing ', i, 'th job.\n', sep = '')
    output # this will become part of the out object
}
```

```
Starting 1th job.
Finishing 1th job.
Starting 2th job.
Finishing 2th job.
Starting 3th job.
Finishing 3th job.
Starting 4th job.
Finishing 4th job.
Starting 5th job.
Finishing 5th job.
Starting 6th job.
Finishing 6th job.
Starting 7th job.
Finishing 7th job.
Starting 8th job.
Finishing 8th job.
Starting 9th job.
Finishing 9th job.
```

```
Starting 10th job.
Finishing 10th job.
```

```
length(list)
```

```
[1] 1
```

```
result[[1]][1:5]
```

```
       20          26          29          30          69
2.6714476 -0.1169513  0.8114822 -1.1478375  0.5954467
```

You can debug by running serially using `%do%` rather than `%dopar%` or `%dorng%`. Note that you may need to load packages within the `foreach` construct to ensure a package is available to all of the calculations.

Alternatively using parallel apply statements

The `future.apply` package also has the ability to parallelize the various `apply` functions (`apply`, `lapply`, `sapply`, etc.).

We'll consider parallel `future_lapply` and `future_sapply`.

```
library(future.apply)
nCores <- 2
plan(multisession, workers = nCores)

input <- seq_len(nFolds)
input
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
system.time(
    res <- future_sapply(input, cvFit, folds, Y, X, future.seed = TRUE)
)
```

```
   user  system elapsed
  1.110   0.030  20.401
```

```
system.time(
    res2 <- sapply(input, cvFit, folds, Y, X)
)
```

```
   user  system elapsed
 37.998   0.003  38.022
```

Question: why are the user time (and system time) miniscule when using `future_sapply`?

Now suppose you have 4 cores (and therefore won't have an equal number of tasks per core). The approach in the next scenario should work better.

Scenario 4: parallelizing over prediction methods

Scenario: parallelizing over prediction methods or other cases where execution time varies

If you need to parallelize over prediction methods or in other contexts in which the computation time for the different tasks varies widely, you want to avoid having the parallelization tool group the tasks in advance, because some cores may finish a lot more quickly than others. However, in many cases, this sort of grouping in advance (called prescheduling or 'static' allocation of tasks to workers) is the default. This is also the case with the future package – the default is to group the tasks in advance into "chunks", so that each worker processes one future (one chunk), containing multiple tasks.

First we'll set up an artificial example with four slow tasks and 12 fast tasks and see the speed of running with the default of prescheduling. Whether to preschedule or not is controlled by either the `future.chunk.size` or `future.scheduling` arguments.

```
## @knitr parallel-lapply-preschedule

library(future.apply)
nCores <- 4
plan(multisession, workers = nCores)

## specifically designed to be slow when have four cores and
## and use prescheduling, because
## the slow tasks all assigned to one worker
n <- rep(c(1e7, 1e5, 1e5, 1e5), each = 4)


fun <- function(i) {
    cat("working on ", i, "; ")
    mean(lgamma(exp(rnorm(n[i]))))
}

system.time(fun(1))
```

```
working on  1 ;
```

```
   user  system elapsed
  1.434   0.007   1.442
```

```
system.time(fun(5))
```

```
working on  5 ;
```

```
   user  system elapsed
  0.017   0.000   0.017
```

```
## Static allocation ##

## default - should do static allocation
system.time(
    res <- future_sapply(seq_along(n), fun, future.seed = TRUE)
)
```

```
working on  1 ; working on  2 ; working on  3 ; working on  4 ; working on  5 ; working on  6 ; worki
```

```
   user  system elapsed
  0.286   0.004   6.800
```

```
## this is the default: 1 future (and therefore 4 tasks) per worker
system.time(
    res <- future_sapply(seq_along(n), fun, future.scheduling = 1,
                         future.seed = TRUE)
)
```

```
working on  1 ; working on  2 ; working on  3 ; working on  4 ; working on  5 ; working on  6 ; worki
```

```
   user  system elapsed
  0.302   0.006   6.824
```

```
## equivalently, 4 tasks per chunk, 1 chunk (1 future) per worker
system.time(
    res <- future_sapply(seq_along(n), fun, future.chunk.size = 4,
                         future.seed = TRUE)
)
```

```
working on  1 ; working on  2 ; working on  3 ; working on  4 ; working on  5 ; working on  6 ; worki
```

```
   user  system elapsed
  0.269   0.004   6.612
```

And here we prevent prescheduling. I find the `future.chunk.size` argument easier to understand than the `future.scheduling` argument. `future.chunk.size` says how many tasks to group together. So setting equal to 1 means no grouping and therefore not using static allocation.

```
## Dynamic allocation ##

## 1 task per chunk, 4 chunks (4 futures) per worker
system.time(
    res <- future_sapply(seq_along(n), fun, future.chunk.size = 1,
                         future.seed = TRUE)
)
```

```
working on  1 ; working on  2 ; working on  3 ; working on  4 ; working on  5 ; working on  6 ; worki
```

```
   user  system elapsed
  0.285   0.001   2.550
```

```
## or, equivalently, we could specify future.scheduling = 4
```

Scenario 5: 10-fold CV across multiple methods with many more than 10 cores

Scenario: You are running an ensemble prediction method such as SuperLearner or Bayesian model averaging on 10 cross-validation folds, with many statistical/machine learning methods.

Here you want to take advantage of all the cores you have available, so you can't just parallelize over folds.

First we'll discuss how to deal with the nestedness of the problem and then we'll talk about how to make use of many cores across multiple nodes to parallelize over a large number of tasks.

Scenario 5A: nested parallelization

One can always flatten the looping, either in a for loop or in similar ways when using apply-style statements.

```
## original code: multiple loops
for(fold in 1:n) {
  for(method in 1:M) {
     ### code here
  }
}
## revised code: flatten the loops
output <- foreach(idx = 1:(n*M)) %dopar% {
   fold <- idx %/% M + 1
   method <- idx %% M + 1
   ### code here
}
```

Alternatively, `foreach` supports nested parallelization as follows:

```
output <- foreach(fold = 1:n) %:%
  foreach(method = 1:M) %dopar% {
     ## code here
  }
```

The `%:%` basically causes the nesting to be flattened, with `n*M` total tasks run in parallel.

One can also use nested futures and the future package will just take care of parallelizing across all the individual tasks. I won't go into that here, but there is information in the tutorial.

Scenario 5B: Parallelizing across multiple nodes

If you have access to multiple machines networked together, including a Linux cluster, you can use the tools in the future package across multiple nodes (either in a nested parallelization situation with many total tasks or just when you have lots of unnested tasks to parallelize over). Here we'll just illustrate how to use multiple nodes, but if you had a nested parallelization case you can combine the ideas just above with the use of multiple nodes.

Simply start R as you usually would.

Here we'll use `foreach` with the future-based `doFuture` backend.

```
library(doFuture)
```

```
Loading required package: foreach
```

```
library(doRNG)
```

```
Loading required package: rngtools
```

```
## Specify the machines you have access to and
##    number of cores to use on each:
machines = c(rep("radagast.berkeley.edu", 1),
    rep("gandalf.berkeley.edu", 1),
    rep("arwen.berkeley.edu", 2))

## On the SCF, Savio and other clusters using the SLURM scheduler,
## you can figure out the machine names and set up the input to
## the 'workers' argument of 'plan' like this:
## machines <- system('srun hostname', intern = TRUE)

plan(cluster, workers = machines)

registerDoFuture()

fun = function(i, n = 1e6)
  out = mean(rnorm(n))

nTasks <- 120

print(system.time(out <- foreach(i = 1:nTasks) %dorng% {
    outSub <- fun(i)
    outSub # this will become part of the out object
}))
```

```
   user  system elapsed
  0.544   0.016   5.467
```

To use `future_lapply`, set up the plan in similar fashion to above. You can then do:

```
system.time(
    res <- future_sapply(input, cvFit, folds, Y, X)
)

## And just to check we are actually using the various machines:
future_sapply(seq_along(workers), function(i) Sys.getenv('HOST'))
```

Scenario 6: Stratified analysis on a very large dataset

Scenario: You are doing stratified analysis on a very large dataset and want to avoid unnecessary copies.

In many of R's parallelization tools, if you try to parallelize this case on a single node, you end up making copies of the original dataset, which both takes up time and eats up memory.

Here when we use the `multisession` plan, we make copies for each worker. And it's even worse if we force each task to be sent separately so that there is one copy per task.

```
do_analysis <- function(i) {
    return(mean(x))
}
x <- rnorm(5e7)  # our big "dataset"

options(future.globals.maxSize = 1e9)

plan(multisession, workers = 4) # new processes - copying!
system.time(tmp <- future_sapply(1:100, do_analysis))  # 9 sec.

## even worse if we dynamically allocate the tasks
system.time(tmp <- future_sapply(1:100, do_analysis,
                                 future.chunk.size = 1)) # 23 sec.
```

However, if you are working on a single machine (i.e., with shared memory) you can avoid this by using parallelization strategies that fork the original R process (i.e., make a copy of the process) and use the big data objects in the global environment (yes, this violates the usual programming best practices of not using global variables). The `multicore` plan (not available on Windows) allows you to do this.

This creates R worker processes with the same state as the original R process. Interestingly, this means that global variables in the forked worker processes are just references to the objects in memory in the original R process. So the additional processes do not use additional memory for those objects (despite what is shown in `top`) and there is no time involved in making copies. However, if you modify objects in the worker processes then copies are made.

So here we avoid copying the original dataset.

```
plan(multicore, workers = 4)  # forks (where supported, not Windows); no copying!
system.time(tmp <- future_sapply(1:100, do_analysis))  # 6.5 sec.
```

And here is code you can run to demonstrate that when using multicore, no copies are made.

```
x <- c(3.1, 2.5, 7.3)
lobstr::obj_addr(x)

future_sapply(1:2, function(i) {
    ## First, use the global 'x' just in case anything funny going on
    ## before object is used.
    y <- x[1]
    print(lobstr::obj_addr(x))
})
```

Scenario 7: Simulation study with n=1000 replicates: parallel random number generation

We won't cover this in class and you don't need to worry about this at the moment. Instead, I will mention the issue in the simulation unit when we talk about random number generation.

In Section 5, we set the random number seed to different values for random sample. One danger in setting the seed like that is that the random numbers in the different samples could overlap somewhat. This is probably somewhat unlikely if you are not generating a huge number of random numbers, but it's unclear how safe it is.

The key thing when thinking about random numbers in a parallel context is that you want to avoid having the same 'random' numbers occur on multiple processes. On a computer, random numbers are not actually random but are generated as a sequence of pseudo-random numbers designed to mimic true random numbers. The sequence is finite (but very long) and eventually repeats itself. When one sets a seed, one is choosing a position in that sequence to start from. Subsequent random numbers are based on that subsequence. All random numbers can be generated from one or more random uniform numbers, so we can just think about a sequence of values between 0 and 1.

Scenario: You are running a simulation study with n=1000 replicates.

Each replicate involves fitting two statistical/machine learning methods.

Here, unless you really have access to multiple hundreds of cores, you might as well just parallelize across replicates.

However, you need to think about random number generation. If you have overlap in the random numbers the replications may not be fully independent.

In R, the `rlecuyer` package deals with this. The L'Ecuyer algorithm has a period of 2^{191}, which it divides into subsequences of length 2^{127}.

Here's how you initialize independent sequences on different processes when using the `future_lapply`. All you need to do is set the argument `future.seed`.

```
library(future.apply)

fun <- function(i) {
    mean(lgamma(exp(rnorm(100))))
}

nCores <- 4
plan(multisession, workers = nCores)

nSims <- 50
res <- future_sapply(seq_len(nSims), fun, future.seed = 1)
```

Dealing with parallel random number generation when using `foreach` or `future()` is a bit more involved. See the tutorial.

6. Additional details and topics (optional)

Setting the number of threads (cores used) in threaded code (including parallel linear algebra in R)

In general, threaded code will detect the number of cores available on a machine and make use of them. However, you can also explicitly control the number of threads available to a process.

For most threaded code (that based on the openMP protocol), the number of threads can be set by setting the OMP_NUM_THREADS environment variable (VECLIB_MAXIMUM_THREADS on a Mac). E.g., to set it for four threads in the bash shell:

```
export OMP_NUM_THREADS=4
```

Do this before starting your R or Python session or before running your compiled executable.

Alternatively, you can set OMP_NUM_THREADS as you invoke your job, e.g., here with R:

```
OMP_NUM_THREADS=4 R CMD BATCH --no-save job.R job.out
```

Important warnings about use of threaded BLAS

Speed and threaded BLAS

In many cases, using multiple threads for linear algebra operations will outperform using a single thread, but there is no guarantee that this will be the case, in particular for operations with small matrices and vectors. You can compare speeds by setting OMP_NUM_THREADS to different values. In cases where threaded linear algebra is slower than unthreaded, you would want to set OMP_NUM_THREADS to 1.

More generally, if you are using the parallel tools in Section 4 to simultaneously carry out many independent calculations (tasks), it is likely to be more effective to use the fixed number of cores

available on your machine so as to split up the tasks, one per core, without taking advantage of the threaded BLAS (i.e., restricting each process to a single thread).

Conflicts between openBLAS and various R functionality

In the past, I've seen various issues arising when using threaded linear algebra. In some cases when the parallelization uses forking, I have seen cases where R hangs and doesn't finish the linear algebra calculation.

I've also seen a conflict between threaded linear algebra and R profiling (recall the discussion of profiling in the efficient R tutorial).

Some solutions are to set OMP_NUM_THREADS to 1 to prevent the BLAS from doing threaded calculations or to use parallelization approaches that avoid forking.

7. Using Dask in Python

Dask has similar functionality to R's future package for parallelizing across one or more machines/nodes. In addition, it has the important feature of handling distributed datasets - datasets that are split into chunks/shareds and operated on in parallel. We'll see more about distributed datasets in Unit 8 but here we'll introduce the basic functionality.

Scheduler

The scheduler is the analogue of plan in the R future package. For example to parallelize across multiple cores via separate Python processes, we'd do this.

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)
```

This table shows the different types of schedulers.

Type	Description	Multi-node	Copies of objects made?
synchronous	not in parallel (serial)	no	no
threaded	threads within current Python session	no	no
processes	background Python sessions	no	yes
distributed	Python sessions across multiple nodes	yes	yes

Comments:

1. Note that because of Python's Global Interpreter Lock (GIL) (which prevents threading of Python code), many computations done in pure Python code won't be parallelized using the 'threaded' scheduler; however computations on numeric data in numpy arrays, Pandas dataframes and other C/C++/Cython-based code will parallelize.

2. It's fine to use the distributed scheduler on one machine, such as your laptop. According to the Dask documentation, it has advantages over multiprocessing, including the diagnostic dashboard (see the tutorial) and better handling of when copies need to be made. In addition, one needs to use it for parallel map operations (see next section).

Parallel map

This is the analog of apply/lapply/sapply type functions in R. As we've discussed those are examples of map operations.

To do a parallel map, we need to use the distributed scheduler, but it's fine to do that with multiple cores on a single machine (such as a laptop).

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(n_workers = 4)
c = Client(cluster)

# code in calc_mean.py will calculate the mean of many random numbers
from calc_mean import *

p = 20
n = 100000000  # must be of type integer
# set up and execute the parallel map
inputs = [(i, n) for i in range(p)]
# execute the function across the array of input values
future = c.map(calc_mean_vargs, inputs)
results = c.gather(future)
results
```

The map operation appears to cache results. If you rerun the above with the same inputs, you get the same result back essentially instantaneously. HOWEVER, that means that if there is randomness in the results of your function for a given input, Dask will just continue to return the original output.

Futures / delayed evaluation

The analog of using `future()` in R to delay/parallelize tasks is shown here. We use `delayed` to indicate the tasks and then to actually evaluate the code we need to run `compute`. This is another example of lazy evaluation.

```
# code in calc_mean.py will calculate the mean of many random numbers
from calc_mean import *

import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)

futures = []
```

```
p = 10
n = 100000000
for i in range(p):
    futures.append(dask.delayed(calc_mean)(i, n))  # add lazy task

futures
results = dask.compute(futures)  # compute all in parallel
```

Final notes

I've only hit a few highlights here, in particular analogous functionality to the future package, but there's lots more details in the tutorial.

Some additional comments regarding the principles of parallelization already discussed:

1. You can set up nested parallelizations easily using `delayed`.
2. Dask generally uses dynamic allocation (no prescheduling), which can be a drawback on some cases. You may want to manually break up computations into chunks in some cases.
3. You generally don't want to call `compute` separately for multiple steps of a computation, as Dask will generally avoid keeping things in memory. Instead, write out the code for all the steps and then call `compute` once.
4. Except with the `threads` scheduler, copies are made of all objects passed to the workers. However if you use the `distributed` scheduler, you can arrange things so one copy is sent for each worker (rather than for each task).
5. With a bit more work than in the future package in R, you can set up safe parallel random number generation.