Parallel processing

Chris Paciorek

2025-08-12

Table of contents

Overview

References:

- Tutorial on parallel processing using Python's Dask and R's future packages
- Tutorial on parallelization in various languages, including use of PyTorch and JAX in Python

> Parallelization and operating systems
>
> 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.

As context, let's consider some ways we might be able to achieve faster computation:

- better algorithms: This has been quite important in many areas of science but is not the focus here.
- more computationally-efficient implementations of an algorithm: This was a topic in Unit 5.
- faster computers: When CPUs were getting faster at a rapid pace (Moore's Law) this was quite important, but that's no longer the case for CPUs. However, GPU technology is improving rapidly.
- "more" computers: We can try to exploit more processors (more CPUs, more GPU threads, more compute nodes) for a given computation. That is the topic of this Unit.

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 computation 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 Perlmutter supercomputer at Lawrence Berkeley National Lab, which has 3072 CPU-only nodes and 1792 nodes with GPUs, and a total of about 500,000 CPU cores. Each node has 512 GB of memory for a total of 2.3 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. A given CPU will have multiple cores. (E.g, the AMD EPYC 7763 has 64 cores per CPU.)
- *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 have the same number of cores available to our code 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.

- *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.
- *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.

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 generally not an issue in Python or 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 Python and R that use MPI behind the scenes, we'll only cover distributed memory parallelization via `Dask`, which doesn't use MPI.

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.

Some other approaches to parallel processing

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 databases and 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 wouldn'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 Dask

Before we illustrate implementation of various kinds of parallelization, I'll give an overview of the `Dask` package, which we'll use for many of the implementations.

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 7 but here we'll introduce the basic functionality.

There are lots (and lots) of other packages in Python that also provide functionality for parallel processing, including `ipyparallel`, `ray`, `multiprocessing`, and `pp`.

Overview: Key idea

A key idea in Dask (and in R's `future` package and the `ray` package for Python) involve abstracting (i.e, divorcing/separating) the specification of the parallelization in the code away from the computational resources that the code will be run on. We want to:

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

The computational resources on which the code is run is sometimes called the *backend*.

Overview of parallel backends

One sets the *scheduler* to control how parallelization is done, whether to run code on multiple machines, and how many cores on each machine to use.

For example to parallelize across multiple cores via separate Python processes, we'd do this.

```
import dask
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
threads*	threads within current Python session	no	no
processes	background Python sessions	no	yes
distributed**	Python sessions across multiple nodes	yes	yes

(*) 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 'threads' scheduler; however computations on numeric data in numpy arrays, Pandas dataframes and other C/C++/Cython-based code will parallelize.

(**) 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).

Accessing variables and workers in the worker processes

Dask usually does a good job of identifying the packages and (global) variables you use in your parallelized code and importing those packages on the workers and copying necessary variables to the workers.

Here's a toy example that shows that the `numpy` package and a global variable `n` are automatically available in the worker processes without any action on our part.

Note the use of the `@delayed` decorator to flag the function so that it operates in a lazy manner for use with Dask's parallelization capabilities.

```
import dask
dask.config.set(scheduler='processes', num_workers = 4, chunksize = 1)
```

```
<dask.config.set object at 0x7eccfadf7c50>
```

```
import numpy as np
n = 10

@dask.delayed
def myfun(idx):
   return np.random.normal(size = n)


tasks = []
p = 8
for i in range(p):
    tasks.append(myfun(i))  # add lazy task

tasks
```

```
[Delayed('myfun-95f5bb80-faad-4339-a137-8d3883a75a11'), Delayed('myfun-72df575e-8367-4e84-9937-db8a02
```

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

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

We don't have to flag the function in advanced with `@delayed`. We could also have directly called the decorator like this, which as the advantage of allowing us to run the function in the normal way if we simply invoke it.

```
tasks.append(dask.delayed(myfun)(i))
```

5. Illustrating the principles in specific case studies

Scenario 1: one model fit

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

General scenario: Parallelizing a single task.

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 `RandomForestClassifier` in scikit-learn's `ensemble` module indicates it can use multiple cores – note the `n_jobs` argument (not shown here because the help info is very long).

```
import sklearn.ensemble
help(sklearn.ensemble.RandomForestClassifier)
```

> Tip
>
> You'll usually need to look for an argument with one of the words *threads*, *processes*, *cores*, *cpus*, *jobs*, etc. in the argument name.

Scenario 1B: Parallelized linear algebra

If a method does linear algebra computations on large matrices/vectors, Python (and 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
- Apple's Accelerate framework BLAS (*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 shell environment variable `OMP_NUM_THREADS` is not set to one. (Macs make use of `VECLIB_MAXIMUM_THREADS` rather than `OMP_NUM_THREADS` and if MKL is being used, then one needs `MKL_NUM_THREADS`)

For parallel (threaded) linear algebra in Python, one can use an optimized BLAS with the `numpy` and (therefore) `scipy` packages, on Linux or using the Mac's *vecLib* BLAS. Details will depend on how you install Python, numpy, and scipy. More details on figuring out what BLAS is being used and how to install a fast threaded BLAS on your own computer are here.

Dask and some other packages also provide threading, but pure Python code is not threaded.

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

```
import numpy as np
import time
```

```
x = np.random.normal(size = (6000, 6000))

start_time = time.time()
x = np.dot(x.T, x)
U = np.linalg.cholesky(x)
elapsed_time = time.time() - start_time
print("Elapsed Time (8 threads):", elapsed_time)
```

We'd need to restart Python after setting `OMP_NUM_THREADS` to 1 in order to compare the time when run in parallel vs. on a single core. That's hard to demonstrate in this generated document, but when I ran it, it took 6.6 seconds, compared to 3 seconds using 8 cores.

> **Warning**
>
> Note that for smaller linear algebra problems, we may not see any speed-up or even that the threaded calculation might be slower because of overhead in setting up the parallelization and because the parallelized linear algebra calculation involves more actual operations than when done serially.

Scenario 1C: GPUs and linear algebra

Linear algebra with large matrices is often a very good use case for GPUs. So if you or someone implementing a method can run the linear algebra you need on a GPU, that can give a big speedup.

Here's an example of using the GPU to multiply large matrices using PyTorch. We could do this similarly with Tensorflow or JAX. I've just inserted the timing from running this on an SCF machine with a powerful GPU.

Packages such as PyTorch, Tensorflow, and JAX can run linear algebra calculations in parallel on either the CPU or GPU (depending on whether a GPU is available on the computer the code is being run on).

Under the hood, there are different implementations (sometimes called *kernels*) of a given computation For example, there would be both parallelized CPU and GPU implementations of matrix multiplication.

```
import torch

start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

gpu = torch.device("cuda:0")

n = 10000
x = torch.randn(n,n, device = gpu)
y = torch.randn(n,n, device = gpu)

## Time the matrix multiplication on GPU:
```

```
start.record()
z = torch.matmul(x, y)
end.record()
torch.cuda.synchronize()
print(start.elapsed_time(end))   # 120 ms.

## Compare to CPU:
cpu = torch.device("cpu")

x = torch.randn(n,n, device = cpu)
y = torch.randn(n,n, device = cpu)

## Time the matrix multiplication on CPU:
start.record()
z = torch.matmul(x, y)
end.record()
torch.cuda.synchronize()
print(start.elapsed_time(end))   # 18 sec.
```

The GPU calculation takes 100-200 milliseconds (ms), while the CPU calculation took 18 seconds using two CPU cores. That's a speed-up of more than 100x!

For a careful comparison between GPU and CPU, we'd want to consider the effect of using 4-byte floating point numbers for the GPU calculation.

We'd also want to think about how many CPU cores should be used for the comparison.

Scenario 1D: Parallelized vectorized calculations

Similarly, packages such as PyTorch, Tensorflow, and JAX can parallelize vectorized calculations on either the CPU or GPU.

Recall that in Unit 5, we used JAX to run a vectorized calculation on a very large 1-d array. JAX automatically ran that in parallel across multiple (CPU) cores.

Here we'll see that if we have GPU available, JAX will automatically run the calculation in parallel on the GPU. I'll do this separately on a machine with a GPU and paste the results in here.

```
import time
import jax
import jax.numpy as jnp

def myfun_jnp(x):
    y = jnp.exp(x) + 3 * jnp.sin(x)
    return y

n = 500000000

key = jax.random.key(1)
```

```
x_jax = jax.random.normal(key, (n,))  # 32-bit by default
print(x_jax.platform())

t0 = time.time()
z_jax1 = myfun_jnp(x_jax).block_until_ready()
t_gpu = round(time.time() - t0, 3)

cpu_device = jax.devices('cpu')[0]
with jax.default_device(cpu_device):
    key = jax.random.key(1)
    x_jax = jax.random.normal(key, (n,))  # 32-bit by default
    print(x_jax.platform())
    t0 = time.time()
    z_jax2 = myfun_jnp(x_jax).block_until_ready()
    t_cpu = round(time.time() - t0, 3)

print(f"GPU time: {t_gpu}\nCPU time: {t_cpu}")
```

```
GPU time: 0.015
CPU time: 4.709
```

So the GPU is much faster, even though the JAX CPU implementation uses multiple threads (as can be seen with `top`).

Forcing JAX to use the CPU when a GPU is available is a bit of a hassle. When I tried to use `jax.config.update('jax_platform_name', 'cpu')`, it didn't seem to work for some reason.

Scenario 2: three different prediction methods on your data

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

General scenario: Parallelizing a small number of tasks.

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

Here we'll use the `processes` scheduler. In principal given this relies on numpy code, we could have also used the `threads` scheduler, but I'm not seeing effective parallelization when I try that.

```
import dask
import time
import numpy as np

def gen_and_mean(func, n, par1, par2):
    return np.mean(func(par1, par2, size = n))
```

```
dask.config.set(scheduler='processes', num_workers = 3, chunksize = 1)
```

```
<dask.config.set object at 0x7eccfbcc07d0>
```

```
n = 250000000

tasks = []
tasks.append(dask.delayed(gen_and_mean)(np.random.normal, n, 0, 1))
tasks.append(dask.delayed(gen_and_mean)(np.random.gamma, n, 1, 1))
tasks.append(dask.delayed(gen_and_mean)(np.random.uniform, n, 0, 1))

t0 = time.time()
results = dask.compute(tasks)
print(time.time() - t0)
```

```
10.23275899887085
```

```
t0 = time.time()
p = gen_and_mean(np.random.normal, n, 0, 1)
q = gen_and_mean(np.random.gamma, n, 1, 1)
s = gen_and_mean(np.random.uniform, n, 0, 1)
print(time.time() - t0)
```

```
13.193596124649048
```

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

You could also have used tools like a parallel map here as well, as we'll discuss in the next scenario.

Lazy evaluation, synchronicity, and blocking

If we look at the delayed objects, we see that each one is a representation of the computation that needs to be done and that execution happens lazily. Also note that `dask.compute` executes *synchronously*, which means the main process waits until the `dask.compute` call is complete before allowing other commands to be run. This synchronous evaluation is also called a *blocking* call because execution of the task in the worker processes blocks the main process. In contrast, if control returns to the user before the worker processes are done, that would be *asynchronous* evaluation (aka, a *non-blocking* call).

Note: the use of `chunksize = 1` forces Dask to immediately start one task on each worker. Without that argument, by default it groups tasks so as to reduce the overhead of starting each task individually, but when we have few tasks, that prevents effective parallelization. We'll discuss this in much more detail in Scenario 4.

Lazy evaluation is a concept that works well with computational graphs where the start of one piece of a computation can't start until another piece ends. For example, when we use `dask.delayed`, Dask will first figure out the computational graph underlying a set of function calls and then execute them in the order needed. This makes use of lazy evaluation.

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

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

General scenario: Parallelizing tasks via a parallel map.

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

Here I'll illustrate using a parallel map, using this simulated dataset and basic use of `RandomForestRegressor()`.

First, let's set up our fit function and simulate some data.

In this case our fit function uses global variables. The reason for this is that we'll use Dask's `map` function, which allows us to pass only a single argument. We could bundle the input data with the `fold_idx` value and pass as a larger object, but here we'll stick with the simplicity of global variables.

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold

def cv_fit(fold_idx):
    train_idx = folds != fold_idx
    test_idx = folds == fold_idx
    X_train = X.iloc[train_idx]
    X_test = X.iloc[test_idx]
    Y_train = Y[train_idx]
    model = RandomForestRegressor()
    model.fit(X_train, Y_train)
    predictions = model.predict(X_test)
    return predictions


np.random.seed(1)

# Generate data
n = 1000
p = 50
X = pd.DataFrame(np.random.normal(size = (n, p)),\
                 columns=[f"X{i}" for i in range(1, p + 1)])
Y = X['X1'] + np.sqrt(np.abs(X['X2'] * X['X3'])) +\
    X['X2'] - X['X3'] + np.random.normal(size = n)

n_folds = 10
seq = np.arange(n_folds)
folds = np.random.permutation(np.repeat(seq, 100))
```

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).

```
n_cores = 2
from dask.distributed import Client, LocalCluster
cluster = LocalCluster(n_workers = n_cores)
c = Client(cluster)

tasks = c.map(cv_fit, range(n_folds))
results = c.gather(tasks)
# We'd need to sort the results appropriately to align them with the observations.
```

Now suppose you have 4 cores (and therefore won't have an equal number of tasks per core with the 10 tasks). 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 group the tasks into batches in advance, because some cores may finish a lot more quickly than others. Starting the tasks one by one (not in batches) is called *dynamic allocation*.

In contrast, if the computation time is about the same for the different tasks and you have a large number of tasks (in which case the effect of averaging may also help with load-balancing) then you want to group the tasks into batches. This is called *static allocation* or *prescheduling*. This avoids the extra overhead (~1 millisecond per task) of scheduling many tasks.

Dynamic allocation

With Dask's `distributed` scheduler, Dask starts up each delayed evaluation separately (i.e., dynamic allocation).

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 dynamic allocation under Dask's distributed scheduler. Then in the next section, we'll compare to the worst-case scenario with all four slow tasks in a single batch.

```
import scipy.special

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

## 4 slow tasks and 12 fast ones.
n = np.repeat([10**7, 10**5, 10**5, 10**5], 4)

def fun(i):
    print(f"Working on {i}.")
    return np.mean(scipy.special.gammaln(np.exp(np.random.normal(size = n[i]))))
```

```
t0 = time.time()
out = fun(1)
```

```
Working on 1.
```

```
print(time.time() - t0)
```

```
0.6115095615386963
```

```
t0 = time.time()
out = fun(5)
```

```
Working on 5.
```

```
print(time.time() - t0)
```

```
0.014568090438842773
```

```
t0 = time.time()
tasks = c.map(fun, range(len(n)))
results = c.gather(tasks)
print(time.time() - t0)  # 0.8 sec.
```

```
1.933548927307129
```

```
cluster.close()
```

Note that with relatively few tasks per core here, we could have gotten unlucky if the tasks were in a random order and multiple slow tasks happen to be done by a single worker.

Static allocation

Next, note that by default the 'processes' scheduler sets up tasks in batches, with a default chunksize of 6. In this case that means that the first 4 (slow) tasks are all allocated to a single worker.

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

```
<dask.config.set object at 0x7ecce1b1f360>
```

```
tasks = []
p = len(n)
for i in range(p):
    tasks.append(dask.delayed(fun)(i))  # add lazy task

t0 = time.time()
results = dask.compute(tasks)  # compute all in parallel
print(time.time() - t0)
```

```
2.8317203521728516
```

To force dynamic allocation, we can set `chunksize = 1` (as was shown in our original example of using the `processes` scheduler).

```
dask.config.set(scheduler='processes', num_workers = 4, chunksize = 1)
```

```
<dask.config.set object at 0x7ecce194b0b0>
```

```
tasks = []
p = len(n)
for i in range(p):
    tasks.append(dask.delayed(fun)(i))  # add lazy task

t0 = time.time()
results = dask.compute(tasks)  # compute all in parallel
print(time.time() - t0)
```

```
2.462970018386841
```

In principle that should be faster than with static allocation, but for some reason we are not seeing that to be the case here…

We haven't illustrated it here, but if each task is quick and we have a lot of tasks, we likely want static allocation to avoid the extra overhead of starting each task individually.

Choosing static vs. dynamic allocation in Dask

With the `distributed` scheduler, Dask starts up each delayed evaluation separately (i.e., dynamic allocation). And even with a distributed `map()` it doesn't appear possible to ask that the tasks be broken up into batches. Therefore if you want static allocation, you could use the `processes` scheduler if using a single machine, or if you need to use the `distributed` schduler you could break up the tasks into batches manually.

With the `processes` scheduler, static allocation is the default, with a default chunksize of 6 tasks per batch. You can force dynamic allocation by setting `chunksize = 1`.

(Note that in R, static allocation is the default when using the `future` package.)

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

Specific 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.

General scenario: parallelizing nested tasks or a large number of tasks, ideally across multiple machines.

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 range(n):
  for method in range(M):
     ### code here

## revised code: flatten the loops
for idx in range(n*M):
    fold = idx // M
    method = idx % M
    print(idx, fold, method)### code here
```

Rather than flattening the loops at the loop level (which you'd need to do to use `map`), one could just generate a list of delayed tasks within the nested loops.

```
for fold in range(n):
  for method in range(M):
     tasks.append(dask.delayed(myfun)(fold,method))
```

The future package in R has some nice functionality for easily parallelizing with nested loops.

Scenario 5B: Parallelizing across multiple nodes

If you have access to multiple machines networked together, including a Linux cluster, you can use Dask to start workers 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 Python as you usually would. Then the following code will parallelize on workers across the machines specified.

```
from dask.distributed import Client, SSHCluster
# First host is the scheduler.
cluster = SSHCluster(
    ["gandalf.berkeley.edu", "radagast.berkeley.edu", "radagast.berkeley.edu",
    "arwen.berkeley.edu", "arwen.berkeley.edu"]
)
c = Client(cluster)

## On the SCF, Savio and other clusters using the SLURM scheduler,
## you can figure out the machine names like this, repeating the name of the
## first machine to account for the main/scheduler/controller process:
##
## machines = subprocess.check_output("srun hostname", shell = True,
```

```
##             universal_newlines = True).strip().split('\n')
## machines = [machines[0]] + machines

def fun(i, n=10**6):
    return np.mean(np.random.normal(size = n))

n_tasks = 120

tasks = c.map(fun, range(n_tasks))
results = c.gather(tasks)

## And just to check we are actually using the various machines:
import subprocess

c.gather(c.map(lambda x: subprocess.check_output("hostname", shell = True), \
               range(4)))

cluster.close()
```

Scenario 6: Stratified analysis on a very large dataset

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

General scenario: Avoiding copies when working with large data in parallel.

In many 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. That is because the separate processes do not have access to objects used in the main (or other) processes, even though they are sharing the same physical memory. So the data needs to be sent to each process (or task) individually and then stored as distinct objects in memory.

Here when we use the `processes` scheduler, we make copies. Whether there is a copy per task or a copy per process seems to depend on exactly what the parallelized code is doing (perhaps based on whether there is random number generation in the code).

```
import os

def do_analysis(i,x):
    '''
    A fake "analysis", identical for each task.
    '''
    # Check number of processes and copies.
    print(f"Object id: {id(x)}, process id: {os.getpid()}")
    return np.mean(x)

n_cores = 4
```

```
x = np.random.normal(size = 5*10**7)   # our big "dataset"

dask.config.set(scheduler='processes', num_workers = n_cores, chunksize = 1)
```

```
<dask.config.set object at 0x7eccfa4d4050>
```

```
tasks = []
p = 8
for i in range(p):
    tasks.append(dask.delayed(do_analysis)(i,x))

t0 = time.time()
results = dask.compute(tasks)
t_elapsed = time.time() - t0
```

```
Object id: 140528840440112, process id: 1148556
Object id: 140619687115056, process id: 1148559
Object id: 140509694916912, process id: 1148561
Object id: 140539164195120, process id: 1148560
Object id: 140528840440112, process id: 1148556
Object id: 140619687115056, process id: 1148559
Object id: 140509694916912, process id: 1148561
Object id: 140539164195120, process id: 1148560
```

```
print(t_elapsed)
```

```
31.994030237197876
```

A much better approach here would be to use the `threads` scheduler, in which case all workers can access the same data objects with no copying (but of course we cannot modify the data in that case without potentially causing problems for the other tasks). Without the copying, this is really fast.

```
dask.config.set(scheduler='threads', num_workers = n_cores)
```

```
<dask.config.set object at 0x7eccfa4d4050>
```

```
tasks = []
p = 8
for i in range(p):
    tasks.append(dask.delayed(do_analysis)(i,x))

t0 = time.time()
results = dask.compute(tasks)
```

```
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
```

```
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
Object id: 139418396946672, process id: 1819498
```

```
print(time.time() - t0)
```

```
0.08447074890136719
```

We can also consider using the the `distributed` scheduler (which is fine to use on a single machine or multiple machines). In order to have one copy per worker instead of one copy per task, we can apply `delayed()` to the global data object.

However, with the Dask distributed scheduler, it is complicated to assess what is going on, because the scheduler seems to try to optimize assignment of tasks to workers in a way that may cause an imbalance in the number of tasks assigned to each worker. In this example, all the tasks are assigned to a single worker.

(If instead the `do_analysis` function ran this: `np.mean(x + np.random.normal(size=5*10**7))`, we'd see the tasks be done on more than one worker.)

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

x = dask.delayed(x)  # To have one copy per worker.

tasks = []
p = 8
for i in range(p):
    tasks.append(dask.delayed(do_analysis)(i,x))

t0 = time.time()
results = dask.compute(tasks)
```

```
/system/linux/miniforge-3.13/lib/python3.13/site-packages/distributed/client.py:3370: UserWarning: Se
This may cause some slowdown.
Consider loading the data with Dask directly
 or using futures or delayed objects to embed the data into the graph without repetition.
See also https://docs.dask.org/en/stable/best-practices.html#load-data-with-dask for more information
  warnings.warn(
```

```
print(time.time() - t0)
```

```
1.443199872970581
```

```
cluster.close()
```

Also, Dask gives a warning about sending the data to the workers in advance. I'm not sure of the distinction between what it is recommending and use of `dask.delayed(x)`. When I tried to use `scatter()` in various ways, I wasn't able to silence the warning.

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

We'll probably skip this for now and come back to it when we discuss random number generation in the Simulation Unit.

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.

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

General scenario: Safely handling random number generation in parallel.

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. One option is to set the random number seed to different values for each replicate. One danger in setting the seed like that is that the random numbers in the different replicate 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.

We can use functionality with numpy's PCG64 or MT19937 generators to be completely safe in our parallel random number generation. Each provide a `jumped()` function that moves the RNG ahead as if one had generated a very large number of random variables (2^{128}) for the Mersenne Twister and nearly that for the PCG64).

Here's how we can set up the use of the PCG64 generator:

```
bitGen = np.random.PCG64(1)
rng = np.random.Generator(bitGen)
rng.random(size = 3)
```

```
array([0.51182162, 0.9504637 , 0.14415961])
```

Now let's see how to jump forward. And then verify that jumping forward two increments is the same as making two separate jumps.

```
bitGen = np.random.PCG64(1)
bitGen = bitGen.jumped(1)
rng = np.random.Generator(bitGen)
rng.normal(size = 3)
```

```
array([ 1.23362391,  0.42793616, -1.90447637])
```

```
bitGen = np.random.PCG64(1)
bitGen = bitGen.jumped(2)
```

```
rng = np.random.Generator(bitGen)
rng.normal(size = 3)
```

```
array([-0.31752967,  1.22269493,  0.28254622])
```

```
bitGen = np.random.PCG64(1)
bitGen = bitGen.jumped(1)
bitGen = bitGen.jumped(1)
rng = np.random.Generator(bitGen)
rng.normal(size = 3)
```

```
array([-0.31752967,  1.22269493,  0.28254622])
```

We can also use `jumped()` with the Mersenne Twister.

```
bitGen = np.random.MT19937(1)
bitGen = bitGen.jumped(1)
rng = np.random.Generator(bitGen)
rng.normal(size = 3)
```

```
array([ 0.12667829, -2.1031878 , -1.53950735])
```

So the strategy to parallelize across tasks (or potentially workers if random number generation is done sequentially for tasks done by a single worker) is to give each task the same seed and use `jumped(i)` where `i` indexes the tasks (or workers).

```
def myrandomfun(i):
    bitGen = np.random.PCG(1)
    bitGen = bitGen.jumped(i)
    # insert code with random number generation
```

One caution is that it appears that the period for PCG64 is 2^{128} and that `jumped(1)` jumps forward by nearly that many random numbers. That seems quite strange, and I don't understand it.

Alternatively as recommended in the docs:

```
n_tasks = 10
sg = np.random.SeedSequence(1)
rngs = [Generator(PCG64(s)) for s in sg.spawn(n_tasks)]
## Now pass elements of rng into your function that is being computed in parallel

def myrandomfun(rng):
    # insert code with random number generation, such as:
    z = rng.normal(size = 5)
```

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}.

6. Additional details and topics (optional)

Avoiding repeated calculations by calling `.compute` once

As far as I can tell, Dask avoids keeping all the pieces of a distributed object or computation in memory. However, in many cases this can mean repeating computations or re-reading data if you need to do multiple operations on a dataset.

For example, if you are creating a Dask distributed dataset from data on disk, I think this means that every distinct set of computations (each computational graph) will involve reading the data from disk again.

One implication is that if you can include all computations on a large dataset within a single computational graph (i.e., a call to compute) that may be much more efficient than making separate calls.

Here's an example with Dask dataframe on some air traffic delay data, where we make sure to do all our computations as part of one graph:

```
import dask
dask.config.set(scheduler='processes', num_workers = 6)
import dask.dataframe as ddf
air = ddf.read_csv('/scratch/users/paciorek/243/AirlineData/csvs/*.csv.bz2',
      compression = 'bz2',
      encoding = 'latin1',   # (unexpected) latin1 value(s) 2001 file TailNum field
      dtype = {'Distance': 'float64', 'CRSElapsedTime': 'float64',
      'TailNum': 'object', 'CancellationCode': 'object'})
# specify dtypes so Pandas doesn't complain about column type heterogeneity

import time
t0 = time.time()
air.DepDelay.min().compute()   # about 200 seconds.
print(time.time()-t0)
t0 = time.time()
air.DepDelay.max().compute()   # about 200 seconds.
print(time.time()-t0)
t0 = time.time()
(mn, mx) = dask.compute(air.DepDelay.max(), air.DepDelay.min())  # about 200 seconds
print(time.time()-t0)
```

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

Cautions about 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).

7. Introduction to R's future package (optional)

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
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.