Notes 7: Parallelization

Chris Paciorek

2025-02-11

Table of contents

Introduction

This document is the seventh of a set of notes, this document focusing on the parallelization. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

In some cases, I have set the code chunks to not execute when rendering this document, since the parallelization complicates execution and display of results.

Parallelization overview

Let's first discuss some general concepts related to parallelization, based on the overview in this SCF tutorial.

Glossary of terms

- *cores*: We'll use this term to mean the different processing units available on a single machine or node.
- *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 executing on a machine; multiple processes may be executing at once. A given executable (e.g., Julia or Python) may start up multiple processes at once. Ideally we have no more user 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*: This term gets used in various ways (including in place of 'processes' in the context of Slurm and MPI), but we'll use it to refer to the individual computational items you want to complete - e.g., one task per cross-validation fold or one task per simulation replicate/iteration.
- *threads*: multiple paths of execution within a single process; the OS 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 number of total threads across all processes not exceed the number of cores on a node. Note: this is a separate term/concept from hardware hyperthreading, in which a single core can have two hyperthreads and operate almost as if it had two cores.
- *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 the most popular.
- *load-balanced*: when all the cores that are part of a computation are busy for the entire period of time the computation is running.
- *latency*: the time delay or lag in starting an operation. Note this is a fixed cost and does not scale with the size of the operation.

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 multiple nodes 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.

Types of parallelization

- Threading
  - Multi-threaded computation involves a single process that executes multiple threads (paths of execution on (ideally) as many cores as there are threads.
  - All the threads have access to the same memory (which is efficient but can potentially cause conflicts (aka "race conditions").
  - One will see >100% CPU usage because all usage is associated with a single process.
- Multi-process
  - Multiple processes collaborate to execute a computation.
  - Processes may be controlled by a single main process and/or may execute distinct code based on their ID.
  - One will see multiple processes running.
- GPUs
  - Many processing units (thousands) that are slow individually compared to the CPU but

provide massive parallelism.
  - They have somewhat more limited memory (though modern GPUs like the A100 can have 80 GB of GPU memory).
  - They can only use data in their own memory, not in the CPU's memory, so one must transfer data back and forth between the CPU (the *host*) and the GPU (the *device*). This copying can, in some computations, constitute a very large fraction of the overall computation. So it is best to create the data and/or leave the data (for subsequent calculations) on the GPU when possible and to limit transfers.

More on GPUs in the next set of notes.

One of the nice things about Julia is that much of the parallelization functionality is available directly in the Base package/module or in a limited number of additional packages. In Python and R, there is a confusing explosion of different parallelization packages (though for R, I highly recommend the `future` package and related packages for all your non-GPU parallelization work). E.g., in Python there is `ipyparallel`, `ray`, `dask`, `multiprocessing` in addition to other numerical computing packages that can run code in parallel (and on the GPU), such as JAX and PyTorch.

Threading in Julia

This material is drawn from this SCF tutorial.

Threaded linear algebra

As with Python and R, Julia uses BLAS, a standard library of basic linear algebra operations (written in Fortran or C), for linear algebra operations. A fast BLAS can greatly speed up linear algebra relative to the default BLAS on a machine. Julia uses a fast, open source, free BLAS library called *OpenBLAS*. In addition to being fast when used on a single core, the openBLAS library is threaded - if your computer has multiple cores and there are free resources, your linear algebra will use multiple cores.

Here's an example.

```julia
using LinearAlgebra
using Distributions
using BenchmarkTools

n = 6000;
x = rand(Uniform(0,1), n,n);

println(BLAS.get_num_threads())
```

```
4
```

```julia
function chol_xtx(x)
    z = x'x;   ## `z` is positive definite.
    C = cholesky(z);
    return C;
end
```

```
BLAS.set_num_threads(4);
@btime chol = chol_xtx(x);
```

```
  5.753 s (5 allocations: 549.32 MiB)
```

```
BLAS.set_num_threads(1);
@btime chol = chol_xtx(x);
```

```
  6.859 s (5 allocations: 549.32 MiB)
```

We see that using four threads is faster than one, but in this case the speedup was (surprisingly) minimal. (In Python or R, I would expect a much bigger speedup and since all are using BLAS/LAPACK, I don't understand why there is so little difference in Julia.)

By default, Julia will set the number of threads for linear algebra equal to the number of processors on your machine.

As seen above, you can check the number of threads being used with:

```
BLAS.get_num_threads()
```

```
1
```

You can also control the number of threads used for linear algebra via the `OMP_NUM_THREADS` (or `OPENBLAS_NUM_THREADS`) environment variable in the shell:

```
## Option 1
export OMP_NUM_THREADS=4
julia

## Option 2
OMP_NUM_THREADS=4 julia
```

Threaded for loops and map-like operations

In Julia, you can directly set up software threads to use for parallel processing.

Here we'll see some examples of running a for loop in parallel, both acting on a single object and used as a parallel map operation.

Here we can operate on a vector in parallel:

```
using Base.Threads

n = 50000000;
x = rand(n);

@threads for i in 1:length(x)
    x[i] = tan(x[i]) + 3*sin(x[i]);
end
```

If anyone is familiar with OpenMP, the use of the macro above to tag the loop for parallelization will look familiar.

We could also threads to carry out a parallel map operation, implemented as a for loop.

```
n = 1000;

function test(n)
    x = rand(Uniform(0,1), n,n);
    z = x'x;
    C = cholesky(z);
    return(C.U[1,1])  ## Arbitrary output to assure that the computation was done.
end

a = zeros(12)
@threads for i in 1:12
    a[i] = test(n);
end
```

Spawning tasks on threads

You can also create (aka *spawn*) individual tasks on threads, with the tasks running in parallel.

Let's see an example (taken from here of sorting a vector in parallel, by sorting subsets of the vector in separate threads.

```
import Base.Threads.@spawn

# sort the elements of `v` in place, from indices `lo` to `hi` inclusive
function psort!(v, lo::Int=1, hi::Int=length(v))
    println(current_task(), ' ', lo, ' ', hi)
    if lo >= hi                       # 1 or 0 elements; nothing to do.
        return v
    end
    if hi - lo < 100000               # Below some cutoff, run in serial.
        sort!(view(v, lo:hi), alg = MergeSort)
        return v
    end

    mid = (lo+hi)>>>1                 # Find the midpoint.

    ### Set up parallelization here ###

    ## Sort two halves in parallel, one in current call and one in a new task
    ## in a separate thread:

    half = @spawn psort!(v, lo, mid)  # Task to sort the lower half.
    psort!(v, mid+1, hi)              # Sort the upper half in the current call.
    wait(half)                          # Wait for the lower half to finish.

    temp = v[lo:mid]                    # Create workspace for merging.

    i, k, j = 1, lo, mid+1              # Merge the two sorted sub-arrays.
    @inbounds while k < j <= hi         #   `@inbounds` skips array bound checking for efficiency.
        if v[j] < temp[i]
            v[k] = v[j]
            j += 1
        else
            v[k] = temp[i]
            i += 1
        end
        k += 1
    end
    @inbounds while k < j
        v[k] = temp[i]
        k += 1
        i += 1
    end

    return v
end
```

```
psort! (generic function with 3 methods)
```

How does this work? Let's consider an example where we sort a vector of length 250000.

The vector gets split into elements 1:125000 (run in task #1) and 125001:250000 (run in the main call). Then the elements 1:125000 are split into 1:62500 (run in task #2) and 62501:125000 (run in task #1), while the elements 125001:250000 are split into 125001:187500 (run in task #3) and 187501:250000 (run in the main call). No more splitting occurs because vectors of length less than 100000 are run in serial.

Assuming we have at least four threads (including the main process), each of the tasks will run in a separate thread, and all four sorts on the vector subsets will run in parallel.

```
x = rand(250000);
## Run once to invoke JIT.
psort!(x);
sort!(x);
```

```
Task (runnable) @0x00007f84d3b3cfb0 1 250000
Task (runnable) @0x00007f84d3b3cfb0 125001 250000
Task (runnable) @0x00007f84d3b3cfb0 187501 250000
Task (runnable) @0x00007f84d4f04010 1 125000
Task (runnable) @0x00007f84d4f04010 62501 125000
Task (runnable) @0x00007f84d4f04330 125001 187500
Task (runnable) @0x00007f84d4f044c0 1 62500
```

We see that the output from `current_task()` shows that the task labels correspond with what I stated above.

I'll use `@time` since using the repeated runs in `@btime` would end up sorting an already sorted vector.

```julia
x = rand(250000);
y = x[:];
@time psort!(x);
@time sort!(y; alg = MergeSort);
```

```
Task (runnable) @0x00007f84d3b3cfb0 1 250000
Task (runnable) @0x00007f84d3b3cfb0 125001 250000
Task (runnable) @0x00007f84d3b3cfb0 187501 250000
Task (runnable) @0x00007f84d4f06ef0 1 125000
Task (runnable) @0x00007f84d4f06ef0 62501 125000
Task (runnable) @0x00007f84d4f07080 125001 187500
Task (runnable) @0x00007f84d4f073a0 1 62500
  0.018580 seconds (640 allocations: 2.886 MiB)
  0.176549 seconds (174.99 k allocations: 12.453 MiB, 89.46% compilation time)
```

Once we account for the compilation time, it looks like the parallel sort is not faster than the non-parallel sort.

The number of tasks running in parallel will be at most the number of threads set in the Julia session.

Controlling the number of threads

You can see the number of threads available:

```julia
Threads.nthreads()
```

```
1
```

You can control the number of threads used for threading in Julia (apart from linear algebra) either by:

- setting the `JULIA_NUM_THREADS` environment variable in the shell before starting Julia, or
- starting Julia with the `-t` (or `--threads`) flag, e.g.: `julia -t 4`.

Note that we can't use `OMP_NUM_THREADS` as the Julia threading is not based on OpenMP.

Multi-process parallelization in Julia

This material is drawn from this SCF tutorial.

We'd want to think about how JIT compilation fits into the multi-process parallelization. I have not considered that in these materials.

Parallel map operations

We can use `pmap` to run a parallel map operation across multiple Julia processes (on one or more machines). `pmap` is good for cases where each task takes a non-negligible amount of time, as there is overhead (latency) in starting the tasks.

Here we'll carry out multiple computationally expensive calculations in the map.

We need to import packages and create the function on each of the worker processes using `@everywhere`.

```julia
using Distributed

if nprocs() == 1
    addprocs(4)
end

nprocs()
```

```
WARNING: using Distributed.@spawn in module Main conflicts with an existing identifier.
```

```
5
```

```julia
@everywhere begin
    using Distributions
    using LinearAlgebra
    function test(n)
        x = rand(Uniform(0,1), n,n)
        z = x'x
        C = cholesky(z)
        return C.U[2,3]  ## Arbitrary output to assure that the computation was done.
    end
end

result = pmap(test, repeat([5000],12))
```

```
12-element Vector{Float64}:
 11.989461426380654
 11.658824964255993
 11.118426196924434
 11.76973230570381
 10.881534214124033
 11.869038452560202
 11.376016391448012
 11.882006932816214
 11.488738551062665
 11.591460054395633
 11.496472381835751
 11.615753274221893
```

One can use static allocation (prescheduling) with the `batch_size` argument, thereby assigning that many tasks to each worker to reduce latency.

Parallel `for` loops

One can execute `for` loops in parallel across multiple worker processes as follows. This is particularly handy for cases where one uses a reduction operator (e.g., the `+` here) so that little data needs to be copied back to the main process. (And in this case we don't copy any data to the workers either.)

Here we'll sum over a large number of random numbers with chunks done on each of the workers, comparing the time to a basic `for` loop.

```julia
using BenchmarkTools

function forfun(n)
    sum = 0.0
    for i in 1:n
        sum += rand(1)[1]
    end
    return(sum)
end

function pforfun(n)
   out = @sync @distributed (+) for i = 1:n
       rand(1)[1]
   end
   return(out)
end

n=50000000
@time forfun(n);
@btime forfun(n);
```

```
  2.414207 seconds (50.01 M allocations: 2.981 GiB, 13.18% gc time, 0.43% compilation time)
  1.848 s (50000001 allocations: 2.98 GiB)
```

```julia
@time pforfun(n);
@btime pforfun(n);
```

```
  1.755817 seconds (498.47 k allocations: 33.221 MiB, 23.43% compilation time)
  485.876 ms (319 allocations: 13.39 KiB)
```

The use of `@sync` causes the operation to block until the result is available so we can get the correct timing.

Without a reduction operation, depending on what one is doing, one coud end up passing a lot of data back to the main process, and this could take a lot of time. For such calculations, one would generally be better off using threaded for loops in order to take advantage of shared memory.

⚠ Random number generation in parallel

We'd have to look into how the random number seed is set on each worker to better understand any issues that might arise from parallel random number generation, but I believe that each worker has a different seed (but note that this does not explicitly ensure that the random number streams on the workers are distinct, as is the case if one uses the L'Ecuyer algorithm).

Passing data to the workers

With multiple workers, particularly on more than one machine, one generally wants to be careful about having to copy large data objects to each worker, as that could make up a substantial portion of the time involved in the computation.

One can explicitly copy a variable to the workers in an `@everywhere` block by using Julia's interpolation syntax:

```julia
x = randn(5);
println(x[1])

@everywhere begin
    x = $x  # copy to workers using interpolation syntax
    println(pointer_from_objref(x), ' ', x[1])
end

sleep(2)  # There is probably a better way to wait until all worker printing is done.
```

```
-1.9228287148471004
Ptr{Nothing} @0x00007f8454daee30 -1.9228287148471004
      From worker 2:    Ptr{Nothing} @0x00007f3485d08010 -1.9228287148471004
      From worker 4:    Ptr{Nothing} @0x00007f5cc9b08010 -1.9228287148471004
      From worker 5:    Ptr{Nothing} @0x00007fe85bba4010 -1.9228287148471004
      From worker 3:    Ptr{Nothing} @0x00007f3a6c1a4010 -1.9228287148471004
```

We see based on `pointer_from_objref` that each copy of `x` is stored at a distinct location in memory, even when processes are on the same machine.

Also note that if one creates a variable within an `@everywhere` block, that variable is available to all tasks run on the worker, so it is global' with respect to those tasks. Note the repeated values in the result here.

```julia
@everywhere begin
    x = rand(5)
    function test(i)
        return sum(x)
    end
end

result = pmap(test, 1:12, batch_size = 3)
```

```
12-element Vector{Float64}:
 3.3431159557599757
 2.629200084231368
 2.032317558622009
 2.7901307863117344
 3.3431159557599757
 2.629200084231368
 2.032317558622009
 2.7901307863117344
 3.3431159557599757
 2.629200084231368
 2.032317558622009
 2.7901307863117344
```

If one wants to have multiple processes all work on the same object, without copying it, one can consider using Julia's SharedArray (one machine) or DArray from the DistributedArrays package (multiple machines) types, which break up arrays into pieces, with different pieces stored locally on different processes.

Spawning tasks

One can use the `Distributed.@spawnat` macro to run tasks on processes, in a fashion similar to using `Threads.@spawn`. More details can be found here.

Using multiple machines

In addition to using processes on one machine, one can use processes across multiple machines. One can either start the processes when you start the main Julia session or you can start them from within the Julia session. In both cases you'll need to have the ability to ssh to the other machines without entering your password. In some cases you as the user might need to set up SSH keys.

To start the processes when starting Julia, create a "machinefile" that lists the names of the machines and the number of worker processes to start on each machine.

Here's an example machinefile:

```
radagast.berkeley.edu
radagast.berkeley.edu
gandalf.berkeley.edu
gandalf.berkeley.edu
```

Note that if you're using Slurm on a Linux cluster, you could generate that file in the shell from within your Slurm allocation like this:

```
srun hostname > machines
```

Then start Julia like this:

```
julia --machine-file machines
```

If anyone has used MPI for programming parallel code, the idea of a "machine file" or "host file" will look familiar.

From within Julia, you can add processes like this (first we'll remove the existing worker processes started using `addprocs()` previously):

```
rmprocs(workers())

addprocs([("radagast", 2), ("gandalf", 2)])
```

```
4-element Vector{Int64}:
 6
 7
 8
 9
```

To check on the number of processes:

```
nprocs()
```

```
5
```

When we demo this in class, we'll try our parallel map example (the linear algebra calculation) and check that we have workers running on the various machines using `top` or `ps` after sshing to the machines.

```
result = pmap(test, repeat([5000],12))
```