Intro to Parallel Programming

Nate Hattersley, Justin Latona

Fall 2024

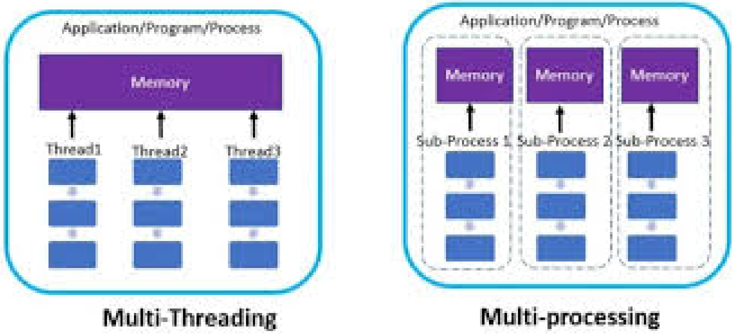
The University of Texas at Austin

The Modern Computer

- Modern computers have multiple *cores*
- Can process multiple instructions simultaneously
- Lets you watch football while doing homework
- We can use multiple cores to speed up estimation procedures
- Broadly known as parallel computing

Models of Parallel Computation I

Two important concepts are *threads* and *processes*



Multiple threads can run inside one parent process. Operating system schedules processes and threads across available cores

Models of Parallel Computation II

- Why prefer multi-processing over multi-threading? Data races. A Julia example:

```
using Base.Threads

winner = 0
@threads for i in 1:10
    winner = i
end
@show winner
```

- `winner` is indeterminate. What would happen in mutli-processing?

Models of Parallel Computation III

- A third model to understand is *cooperative multitasking*
- When making API calls, your computer is just waiting while the server responds
- Why not fire off more requests while waiting for the response?
- Single *thread*, multiple *tasks*
- For IO-bound applications (whereas multi-threading/processing is for CPU-bound)
- See `asyncio` and `aiohttp` in Python for the curious

Loops vs. Implicit Parallelization I

- Implicit parallelization: when programming language allows compiler or interpreter to automatically exploit parallelism inherent to a problem through the language's constructs.

```
import numpy as np

x = np.random.normal(0,1,10**7)  # 10 million random draws
y = np.square(x)                 # takes 0.05 seconds on my PC
y = x**2                         # about the same;
                                 # numpy overrides standard math operators,
                                 # invokes implicitly parallelized functions

y = np.zeros(x.shape[0])
for i in range(x.shape[0]):
    y[i] = x[i]**2               # takes 2.96 seconds on my PC
```

- Avoid loops wherever possible. Leverage implicit parallelization with vectorization and broadcasting.

Loops vs. Implicit Parallelization II

- "Vectorization" in high-level language (e.g. python, matlab, julia) means use of optimized, pre-compiled code in low-level language (e.g. C) to perform mathematical operations over sequence of data.
- E.g. numpy takes advantage of homogenous dtype in an array to delegate task of performing mathematical operations on array's contents to pre-compiled C code.
- Examples: square (unary); np.maximum (binary); np.sum (sequential)

```python
import numpy as np

x = np.random.normal(0,1,10**7)   # 10 million random draws
y = np.sum(x)                     # takes 0.02 seconds on my PC

y = 0
for i in range(x.shape[0]):
    y += x[i]                     # takes 1.91 seconds on my PC
```

- Most numpy math functions are "vectorized," including logical and LA functions.

Broadcasting I

- Mechanism allowing arrays of unequal shapes to be used in vectorized operations.
- Effectively "stretches" arrays to replicate contents along chosen dimensions, such that higher-dim array suits the operation.
- Enables element-wise calculation across otherwise incompatible shapes.
- E.g. let's raise a length-10 vector to powers (element-wise) contained in the columns of a (10, 10 million) matrix.

Broadcasting II

```python
import numpy as np

X = np.random.uniform(1,2,size=(10,10**7))
Y = np.random.uniform(2,10,size=10)

Z = Y**X                          # ValueError: operands could not be broadcast
                                  # together with shapes (10,) (10,10000000)

Z = np.zeros((X.shape))
for i in range(X.shape[1]):
    Z[:,i] = Y**X[:,i]            # could do it this way, takes 43.5 seconds

Z = Y[:,None]**X                  # broadcasting is MUCH faster: 8.9 seconds
```

- Gains are magnified over larger loops and more complicated calculations.
- Might have many such instances in a single structural model. Utilizing implicit parallelization in place of loops can make tractable an otherwise intractable model.

Julia

```julia
using Distributed; addprocs(4)

@everywhere begin
    using Statistics
    a = 4
end

pmap(1:100) do i
    mean(randn(a)) * i
end
```

R

```r
library(parallel)

cl <- makePSOCKCluster(4)
a <- 4

clusterExport(cl, "a")

parSapply(cl, 1:100, function(i) mean(rnorm(a)) * i)
```

Python

```
import numpy as np, functools
from multiprocessing import Pool

def par_func(rng,a,i):
    return np.mean(rng.normal(size=a))*i

def outer(n_procs):
    rng = np.random.default_rng(1)
    a = 4
    with Pool(n_procs) as p:
        p.map(functools.partial(par_func,rng,a),np.arange(1,100).tolist())

if __name__ == '__main__':
    outer(4)
```

- Avoid referring to globally defined objects, pass all function inputs explicitly to child processes (as in Julia and R) by freezing into function signature.
- This is “best practice” for pseudo-random number generation in `numpy`, but *not* when the PRNGs are being drawn in parallel. More on this later.

Matlab

```
clear; clc;

a = 4;
P = 4;
pool = parpool('local', P);

parfor i = 1:100
    mean(randn(a,1)) * i
end

delete(pool);
```

Example: Multinomial Choice Problem - Setup

- 100,000 consumers i, each with an observed scalar individual characteristic W_i.
- 10 product choices j, each with scalar product characteristic X_j.
- Unobserved consumer-specific component of indirect utility S_i is i.i.d. std. normal
- Use 1,000 simulation draws S_{is} for each consumer to form a simulated likelihood.
- Assuming parameters $\beta = (\beta_0, \beta_1, \beta_2, \beta_3)$, indirect utility for i-th consumer, j-th product and s-th simulant is

$$U_{ijs} = \beta_0 + (\beta_1 + \beta_2 W_i + \beta_3 S_{is})X_j + \epsilon_{ijs}$$

- ϵ_{ijs} is idiosyncratic unobserved utility, assumed distributed type-1 extreme value.
- Draw (X_j, W_i, S_{is}), generate fake consumer choices and parameters.
- Parallelize formation of likelihood over consumer-choice pairs, integrating-out simulants to compute CCPs.

Multinomial Choice (python + multiprocessing): DGP

```
import numpy as np
from multiprocessing import Pool
import functools

def data():
    N_cons = 100000                              # 100,000 consumers (W)
    N_choices = 10                               # 10 choices (X)
    N_sims = 1000                                # 1000 sim draws (S)
    W = np.random.uniform(0,1,N_cons)
    X = np.random.uniform(0,1,N_choices)[:,None]
    S = np.random.normal(0,1,N_sims)[None,:]     # prep for broadcasting
    Y = np.random.randint(0,10,N_cons) + 1       # 100k consumer choices (Y)
    prod_IDs = np.arange(10).astype(int) + 1     # product IDs 1-10
                                                 # match choices to prod. IDs
                                                 # to create binary choice matrix
    aux = Y[:,None] / prod_IDs[None,:]           # (N_cons x N_prods)
    choices = np.zeros((aux.shape[0],aux.shape[1]))
    choices[aux==1] = 1
    beta = np.random.uniform(0,1,4)              # fake parameters
    W_list = W.tolist()                          # send W to list for p.map
    return W_list, X, S, choices, beta
```

Multinomial Choice (python + multiprocessing): Compute Likelihood

```
def comp_CCP(X,S,beta,W_list):
    e_util_ijs = np.exp(beta[0] + (beta[1] + beta[2]*W_list + beta[3]*S)*X)
    CCP_ijs = e_util_ijs/np.sum(e_util_ijs,axis=0) # compute CCPs after broadcast
    CCP_ij = np.mean(CCP_ijs,axis=1)              # integrate-out simulants
    return CCP_ij                                 # length-10 vector for i

def total_LL(n_processes):
    W_list, X, S, choices, beta = data()
                        # freeze inputs into CCP function signature
    partial_comp_CCP = functools.partial(comp_CCP,X,S,beta)
    with Pool(n_processes) as p:          # call process pool
                        # p.map takes iterable of consumer characteristics
                        # and generates list of CCP function outputs
        CCP_ij = np.vstack(p.map(partial_comp_CCP,W_list))
        total_LL = np.sum(choices*np.log(CCP_ij) + (1-choices)*np.log(1-CCP_ij))
    return total_LL
                            # need to declare main module, any calls above
if __name__ == '__main__':  # are sent to child processes and duplicated
    n_processes = 20        # allocate 20 cores (of 40 available on VM)
    tot_LL = total_LL(n_processes)
```

Multinomial Choice (python): Discussion I

- Runs in 1.1 seconds on 20 cores.
- “Serial” implementation (also iterating over consumers, but “mapped”) takes about 11 seconds.
- Or, vectorize/broadcast to avoid the loop over consumers? Work in 3D:

```
W = np.random.uniform(0,1,N_cons)[:,None,None]
X = np.random.uniform(0,1,N_choices)[None,:,None]
S = np.random.normal(0,1,N_sims)[None,None,:]
    # other corresponding modifications to functions...
    # (see Github repository for code)
```

- Takes about 30 seconds. Why? We said broadcasting was faster than a loop.
- The issue is `np.exp` of 3D tensor. Only 10s of operation is broadcast to produce 3D tensor, other 20s is exponential of it.

Multinomial Choice (python): Discussion II

- Some `numpy` functions (powers especially) don't vectorize well into high-dimensions. Lots of overhead.
- Numerical example:

```
np.exp time over 100 runs, 10^6 draws: 10.19 ms
mapped math.exp time over 100 runs, 10^6 draws: 0.00026 ms
for-loop math.exp time over 100 runs, 10^6 draws: 339.47 ms
```

- `map` is 39,000 times faster than `np.exp`; 1.3 million times faster than true serial implementation.
  - `map` "works as an iterator to return a result after applying a function to every item of an iterable."
  - Still iterating, but effectively manually vectorizes function onto iterable.
- `np.exp` is implicitly parallel but still slower.
- A true serial (for-loop) implementation over consumer features in MNC problem would take too long, haven't computed it.

Multinomial Choice (python): Discussion III

- Broadcasting works well in high-dimensions with matrix algebra:
  - `matmul`,
  - broadcast operations over arrays, etc.
  - Depending on operation, should try different approaches.
- Taking exponents of a 3D matrix is inherent challenge to computing CCPs (per guess of params) in estimating RC logit demand models with product features and both observed and unobserved individual heterogeneity.

Multinomial Choice (python): Discussion IV

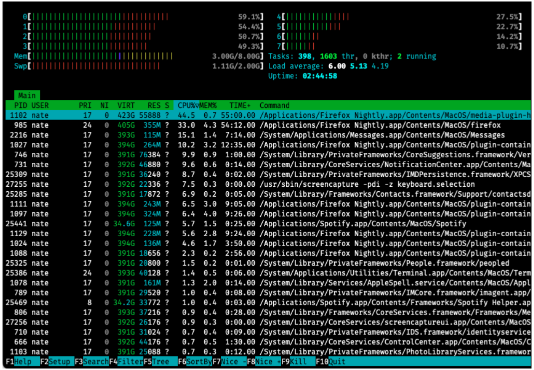
- Workarounds include:
- Iterating over a dimension and taking exponentials of 2D matrices. This should be further optimized:
  - Use `map` to "manually vectorize" (as above)
  - `flatten` a dimension + `bincount` → 2D, no loop
- And/or "just-in-time" compilation: native machine level code optimized to your functions (compile to GPU or CPU); what Julia + Matlab do automatically. Powerful.
- In Python: `jax` and `numba` packages.
- `jax.jit` compiling the CCP and likelihood calc. over 3D takes 3.8 seconds (vs. 30).
- Good idea to use `jit`-compilation in large problems. Need to try `jit` with `map` to further optimize MNC example.

Multinomial Choice (python): Discussion V

- In many situations, you would not be able to replace your entire parallelized operation with broadcasting anyway; need a loop.
- Gains from parallelization can be even larger in real-world problems, depending on what you're parallelizing.
- E.g. bootstrap; iterating over estimation starting points; simulations; running regressions on many separate datasets.
- Key is the operation needs to be "parallelizable:" Passing *many* objects separately through one process (process can change given the iterable).
- Process pool can handle multiple iterables: `Pool.starmap` in python, or package into a list of tuples and use `Pool.map`. Read more, here (link).

Resource Monitoring

- Windows: Ctrl+Alt+Del (Task Manager); Mac: Activity Monitor
- Mac, Linux and Cygwin Terminals: `top/htop`



- Control output of top with your keyboard (link)

File Transfer to VM

- Option 1: Use `scp` from command line. Refer to account on VM as <EID>@ovrw-econ-p02.la.utexas.edu.
- Note: This is the address for the IO VM. Other research fields have separate VM allocations; the address will differ slightly.
- Option 2: Use a GUI `scp` client like WinSCP.
- Option 3: `RDP` local mirror. Access local files in a remote session through your PC's remote desktop GUI. Instructions differ slightly for Windows (link) and Mac (link).
- DO NOT use Box Drive or the Box website. You may be prompted to log into Box Drive upon accessing the VM. Ignore the prompt.

Memory Management

- Point 1: What is RAM? What is disk memory?
- Point 2: Size on disk $\neq$ size in RAM
- Point 3: Multiple processes $\implies$ duplicate memory
- Conclusion: Be careful! Know your RAM limits!
- Example: ML packages that bootstrap

Memory Tips I

- Julia and R: Only pass chunks of your dataframe (if ∼two copies fit in memory)

```
using CSV, DataFrames, Distributed

df = CSV.read("really-big-file.csv") |> DataFrame

pmap(eachrow(df)) do row
    # do things with row...
end
```

- Python can do similar: use `pandas.iterrows`. Still not copying df to child processes.

Memory Tips II

- An alternative: *don't* load the file, pass indices to subprocesses and have subprocesses load parts of the file using limit/offset arguments

```
from multiprocessing import Pool
from math import ceil
import pandas as pd
from functools import partial

def process_batch(N, B, batch_num):
    offset = batch_num * B
    limit = B if offset + B < N else N - offset
    df = pd.read_csv("really-big-file.csv", skiprows = offset, nrows = limit)
    # do something with df...

if __name__ == '__main__':
    BATCH_SIZE = 1000
    N_ROWS = # however many rows your CSV has
    f = partial(process_batch, N_ROWS, BATCH_SIZE)
    with Pool(4) as p:
        p.map(f, range(ceil(N_ROWS / BATCH_SIZE)))
```

Memory Tips III

- Chunked (aka batched) reading is another way to process very large files
  - `chunksize` in the `pandas.read_csv` function (link)
  - read_csv_chunked in R (link)
  - CSV.Chunks in Julia (link)
  - `datastore` in Matlab (link)
- This is about memory management, *not* parallel processing—these functions read one portion of the file at a time. Need to figure out how to add parallelization on top if this is important for performance
- The Python example code on previous slide is effectively parallel batch processing. . .

Cluster Access

- We also have access to one of the world's largest supercomputers at TACC
- Need a prof to sponsor you as PI and pay for an allocation
- This is a *cluster* of computers
- Controlled by Simple Linux Utility for Resource Management (SLURM)
- For when your computational problems get really big

Pseudo-Random Number Generation I

- "Random" numbers generated by a computer only approximate a stochastic process. Sequence generated deterministically from an initial seed.
- How do we seed an RNG so it is reproducible? And how do we do it in parallel?
- In the initial Python multiprocessing example, we seeded a single RNG that can be passed to functions: best-practice (vs. global RNG) outside of parallel environments.
- But in parallel, the RNG is passed to each child process so the draws are the same on each child process.
- Unlikely this is what you want; would be more efficient to draw the random numbers before the parallel process and pass those draws to the child processes.
- Answer is to spawn a `SeedSequence` and child seeds (equal to number of child processes) from it and RNGs from those, then pass those RNGs to the child processes.
- More on the hazards and best practices of PRNG in parallel (link 1, link 2).

Pseudo-Random Number Generation II

- In Julia, the ability to control how you send variables to subprocesses lets you control whether or not the draws are the same in subprocesses.

```
using Distributed; addprocs(2)

@everywhere seed = randn()
@info "Same" seed_one=@fetchfrom(2, seed) seed_two=@fetchfrom(3, seed)

@info "Different" seeds = pmap(1:5) do i
    i => (procid = myid(), x = randn())
end
```

Github Repository

- Github repository (link) with all of our code.
- Includes Python, Matlab, Julia and C implementations of the MNC problem.
- Also speed comparisons across languages and methods on a binomial choice, Rust (1987)-inspired likelihood calculation.
- Corresponding Wiki (link), includes some details which expand on these slides.