

Programming Camp - Lecture 4

Brian Higgins
Ciaran Rogers

This version: September 9, 2021

Presentation outline

1. High Performance Computing
2. Parallel Computing

1 | High Performance Computing

Parallel Computing

Why do we need high performance

1 Quantitative, data-driven economic models are computationally complex

- Finding equilibrium involves solving individual behavior and then iterating for many prices until markets clear;
- Estimation requires solving model with many parameters guesses;
- Games with many players; industries with strategic behavior;
- Auctions with many goods;
- Migration with many cities;

2 Big data can easily overwhelm our laptops

- Every housing transaction in the US (Corelogic, Green Library)
- Product level databases (Nielsen, UC Chicago)

Why do we need high performance

1 Quantitative, data-driven economic models are computationally complex

- Finding equilibrium involves solving individual behavior and then iterating for many prices until markets clear;
- Estimation requires solving model with many parameters guesses;
- Games with many players; industries with strategic behavior;
- Auctions with many goods;
- Migration with many cities;

2 Big data can easily overwhelm our laptops

- Every housing transaction in the US (Corelogic, Green Library)
- Product level databases (Nielsen, UC Chicago)

→ A mix of hardware and software can push the boundary of what is feasible

- Servers, with CPUs, GPUs, big RAM;
- Parallel programming;
- Distributed computing (many computers)

· **Today:** Show you the basics, and highlight what's out there

· Lots to learn from CS folk

- Machine learning tools – e.g. Tensorflow & GPU – can be leveraged for scientific computing
- Worth investing at this point in PhD

GPUs are changing machine learning and computer science

Market Summary > NVIDIA Corporation

NASDAQ: NVDA

226.62 USD +226.21 (55,173.17%) ↑ all time

Closed: Sep 7, 7:59 PM EDT · Disclaimer:

After hours 225.71 -0.91 (0.40%)

1 day

5 days

1 month

6 months

YTD

1 year

5 years

Max



1 | High Performance Computing

1.1 University Clusters vs. Paid Services

1.2 Using Stanford Servers

What type of computer cluster should you use?

- Paid services include Microsoft Azure, Amazon Web Services (AWS) and Google Cloud Platform.

	Paid Services	University Cluster
Monetary cost	[X] Expensive, often paid by the hour	[✓] Free to students and faculty
Waiting times	[✓] Instantly available	[X] Access often involves queues and waiting times
Ease of use	[✓] Low set up costs (e.g., turning on a computer + installing software)	[X] Complicated to understand and use (high fixed costs)

1 | High Performance Computing

1.1 University Clusters vs. Paid Services

1.2 Using Stanford Servers

Some Terminology

- **Computer cluster:**

Collection of separate computer servers, called nodes, which are connected via a fast interconnect.
Configuration allows many computers to work together.

- **Nodes:**

Individual computers designed to accomplish specific tasks (e.g., login nodes, computing nodes).

- **Job Scheduler:** (SLURM)

Software that organizes and assigns tasks to the computer cluster.
Provides a framework to start, execute and monitor jobs within the cluster.

List of Stanford Servers

1. Farmshare: rice (login node), wheat (big memory node), and oat (gpu node).
 - Coursework, and non-sponsored research.
 - Can only connect to rice, and access other nodes from there.

2. Sherlock
 - Sponsored research. Need a faculty sponsor to get access to.
 - Humanities and Sciences partition available `hns`.
 - Connect to login nodes (low computing power) and schedule jobs in high performance nodes.

3. Other servers available tailored to other needs.
 - GSB server: `yen`.
 - Center for Population Health Sciences (PHS): HIPAA compliant; useful for PII data (e.g., Medicare).

Using the terminal

- Connect through SecureCRT (Windows) or the terminal (Mac).

- Connecting to Stanford servers:

```
ssh StanfordID@rice.stanford.edu
```

```
ssh StanfordID@login.sherlock.stanford.edu
```

- Some useful commands:

<code>pwd</code>	list current working directory
------------------	--------------------------------

<code>cd</code>	change directory
-----------------	------------------

<code>touch file.txt</code>	create a new file called file.txt
-----------------------------	-----------------------------------

<code>vi file.txt</code>	open text editor in terminal
--------------------------	------------------------------

<code>module avail</code> or <code>ml avail</code>	show available modules
--	------------------------

<code>module load X</code> or <code>ml X</code>	load module X into workspace
---	------------------------------

<code>module list</code>	list loaded modules
--------------------------	---------------------

Transferring files to/from servers

- **Windows:**

- Use SecureCRT + SecureFX (paid software, free for Stanford students).
More information [click here](#).

- **Mac:**

- Use Cyberduck (free) or other paid software (e.g., Transmit).

- **Farmshare:**

- Transfer through AFS: `http://afs.stanford.edu`
- If working in batch jobs, store in `/farmshare/user_data/SUNetID`

- **Sherlock:**

- Sherlock is pretty flexible. Once you need more information [click here](#).

Write the following code using a text editor

test_matlab.m

```
clc; clear;
rng(29134)
a = random('normal',0,1,10,1)
mean(a)
disp("HELLO FROM MATLAB")
```

test_stata.do

```
set obs 10
set seed 1094
gen random = runiform()
summ random, d
disp "HELLO FROM STATA"
```

test_batch.sbatch

```
#!/bin/bash
#SBATCH -p normal
#SBATCH --job-name=test
#SBATCH --error=test_%j.err
#SBATCH --output=test_%j.out
#SBATCH --nodes=1
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jimenezh@stanford.edu

echo "HELLO FROM BATCH"
cd /farmshare/user_data/jimenezh/test_server

module load matlab
matlab -nodesktop -nosplash -nodisplay < test_matlab.m > output_matlab.txt

module load stata-mp
stata-mp < test_stata.do > output_stata.txt
```

Steps:

Say that I have a `code.sbatch` ready to run.

1. Open Fetch / SecureFX to start your connection:
Hostname: `rice.stanford.edu`
Username: `<SUNet ID>`
Password: `<SUNet Password>`
2. Go to `/farmshare/user_data/<SUNet ID>`.
3. Create a `test_data` folder and copy your code files there.
4. Start your ssh connection. Open the terminal and write:
`ssh <SUNet ID>@rice.stanford.edu`
5. From the terminal, change to the Farmshare directory.
`cd /farmshare/user_data/<SUNet ID>/test_data`
6. Run the code on the server.
`sbatch test_batch.sbatch` or `srun test_batch.sbatch`.
7. Check your status with `squeue -u $USER`.

Other SLURM useful commands

- Submit jobs to specific partitions:

```
#SBATCH -p hns,normal
```

(other: bigmem, gpu, long).

- Request more / less time than default:

```
#SBATCH --time=01:00:00
```

(less time may make your code start faster).

- Request a specific amount of RAM:

```
#SBATCH --mem=128000
```

(note: not all memory - core combinations are available).

- Switch from rice node to a wheat node:

```
srun --qos=interactive --pty /bin/bash -l
```


High Performance Computing

2 | Parallel Computing

Why is some code faster than others?

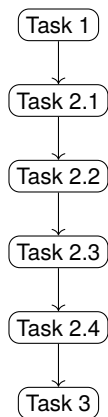
- C is typically fastest language as it compiles code direct to machine readable code
 - Costs more in developer time though!
- Most languages convert your code to C which in turn converts to machine code
 - Matlab, Stata, Python (e.g. numpy, scipy)
- Built-in packages are highly optimized
 - $\beta = \mathbf{X}'\mathbf{X}^{-1}\mathbf{X}'\mathbf{Y}$ runs close to C
 - Parallelization and high performance algorithms under the hood

Why is some code faster than others?

- C is typically fastest language as it compiles code direct to machine readable code
 - Costs more in developer time though!
- Most languages convert your code to C which in turn converts to machine code
 - Matlab, Stata, Python (e.g. numpy, scipy)
- Built-in packages are highly optimized
 - $\beta = \mathbf{X}'\mathbf{X}^{-1}\mathbf{X}'\mathbf{Y}$ runs close to C
 - Parallelization and high performance algorithms under the hood
- Custom algorithms – like value functions w/loops — can be slow though
- Options to go faster
 - 1 Eliminate loops, use vectorized code
 - 2 Write inner loop in C
 - Matlab mex files
 - 3 Just-in-time compilers
 - Julia; Python-Numba, @njit
 - Run slower first time, then much faster
 - 4 Parallelization **Today!**
 - 5 Really parallel: GPU, multi-GPU nodes
 - Python: cupy, numba, dask, tensorflow; Julia...
 - Lots of digestible material from CS folks

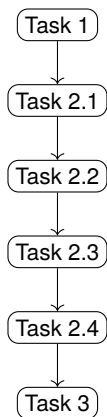
Intuition for parallel computing

We are used to running code like this:

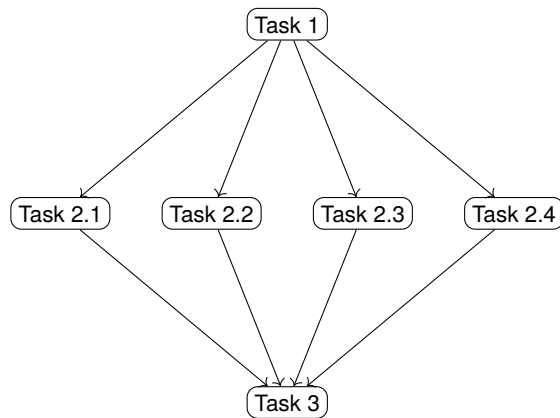


Intuition for parallel computing

We are used to running code like this:

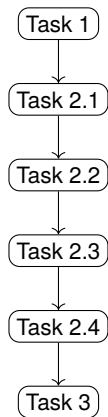


But we could do something else instead:

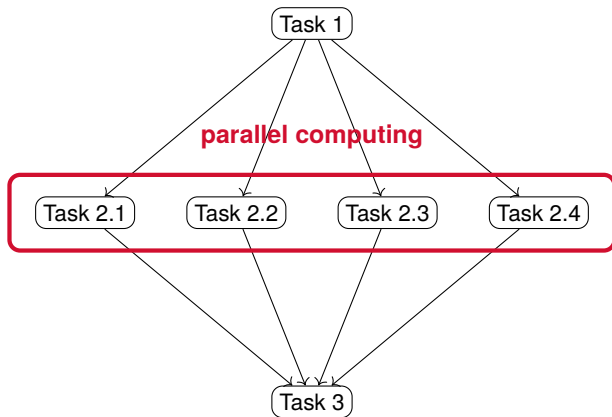


Intuition for parallel computing

We are used to running code like this:



But we could do something else instead:



Explicit Parallelization

- Some software (e.g., Stata) has its own parallel implementation in the backend.
We will discuss here explicit parallelization: asking software to run in parallel and implementing it ourselves.
- Parallel implementations allow code to run in as many CPUs as available in a computer.
The number of CPUs is usually the same as the number of cores, but not always
(CPUs = sockets \times cores \times threads per core).
- There are explicit parallel implementations in MATLAB (e.g., parfor), in R (e.g., BLAS), and Python (e.g., numba, joblib).
- When should you use parallel implementations in your code?
 - When little or no manipulation is needed to separate the problem into a number of parallel tasks.
 - No dependency or need for communication between these parallel tasks.
 - But be wary of overhead costs (tracking individual tasks, communicating results) — need “complicated enough” tasks!
- Common examples: bootstrap, Markov chain Monte Carlo, gradient computation in non-linear problems.

Parallelization on the GPU - kernal code

1 Vectorized code and *gpuArray()*

- 1 Python: Cupy does many numpy operations using the GPU
- 2 Python: Tensorflow Machine Learning libraries heavily vectorized

2 Cuda kernal

- Traditiaonlly written in C CUDA (language for NVIDIA GPUS)
- Now easy to write with Numba in Python (and Julia, matlab options)
- CUDA Kernal: elementwise function to run on the “device” (aka the GPU)
- The host (aka the CPU) sends blocks to be run in parallel on GPU
- GPU memory is separate, need to initialize/end to GPU
- Can scale to multiple GPUs

Parallelization on the GPU - kernel code

```
from __future__ import division
from numba import cuda
import numpy
import math
# CUDA kernel
@cuda.jit
def matmul(A, B, C):
    """ Perform matrix multiplication of  $C = A * B$  """
    row, col = cuda.grid(2)
    if row < C.shape[0] and col < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[row, k] * B[k, col]
        C[row, col] = tmp
```

Parallelization on the GPU - host code

```
A = numpy.full((24, 12), 3, numpy.float) # matrix containing all 3's
B = numpy.full((12, 22), 4, numpy.float) # matrix containing all 4's
# Copy the arrays to the device
A_global_mem = cuda.to_device(A)
B_global_mem = cuda.to_device(B)
# Allocate memory on the device for the result
C_global_mem = cuda.device_array((24, 22))
# Configure the blocks
threadsperblock = (16, 16)
blockspgrid_x = int(math.ceil(A.shape[0] / threadsperblock[0]))
blockspgrid_y = int(math.ceil(B.shape[1] / threadsperblock[1]))
blockspgrid = (blockspgrid_x, blockspgrid_y)

# Start the kernel
matmul[blockspgrid, threadsperblock](A_global_mem, B_global_mem, C_global_mem)
# Copy the result back to the host
C = C_global_mem.copy_to_host()
```

Example: Bootstrapping Clustered Standard Errors

- We imagine a setting where errors are correlated across individuals within groups. Denote by $g \in \{1, \dots, G\}$ the given groups (e.g., states or classrooms) and by $n \in \{1, \dots, N\}$ the individuals within each of these groups. We have $G \times N$ observations in our data.
- Imagine that the true model is $Y_{ng} = \beta \cdot X_{ng} + \eta_{ng}$ with $\eta_{ng} = \varepsilon_g + \varepsilon_{ng}$, $\varepsilon_g \sim \text{i.i.d. } \mathcal{N}(0, 1)$ and $\varepsilon_{ng} \sim \text{i.i.d. } \mathcal{N}(0, 1)$. Parametrization implies that errors are correlated across individuals within the same group.
- OLS still unbiased estimator for β . However, the formula for standard errors needs to account for correlation of unobservables within the same group. One idea to compute standard errors via bootstrap:
 1. Sample groups (with replacement) from the observed data.
 2. For each selected group, select all individuals in the group to construct a dataset of size $G \times I$, possibly w/ repeated obs.
 3. Estimate β via OLS within the bootstrapped data.
 4. Repeat 1-3 B times to compute the bootstrap estimator.
- We parametrize the model with $G = 50$, $N = 100$, $X_{ng} \sim \text{i.i.d. } \mathcal{N}(0, 1)$ and $\beta = 2$.