Introduction to Parallel Computing in Julia

Thomas Wiemann
University of Chicago

TA Discussion # 6
Econ 31720

November 10, 2021

Motivation & Outline

Don't underestimate the importance of a program's runtime. Quote from the IO lunch two weeks ago (paraphrased): "*If your method runs faster than existing approaches, I might try it out. Otherwise, no chance!*"

Runtime is of substantive practical relevance for various reasons:

- convenience (seemingly very important to applied researchers)
- flexibility (e.g., allows for more extensive hyperparameter tuning)
- feasibility (e.g., does the program finish in finite time?)

Construction and implementation of efficient algorithms is difficult, and us being economists (and not computer scientists) doesn't help! Fortunately, one of the most common ways of speeding up computations is also reasonably straightforward: Parallel computing – i.e., the execution of several calculations simultaneously.

Today is meant as a brief introduction to:

- Key Concepts in Parallel Computing
- Parallel Computing in (and with) Julia

One size does not fit all

Not all programs benefit equally from parallelization. How (and whether) an algorithm can leverage parallel processing depends crucially on the nature of the problem it is trying to address.

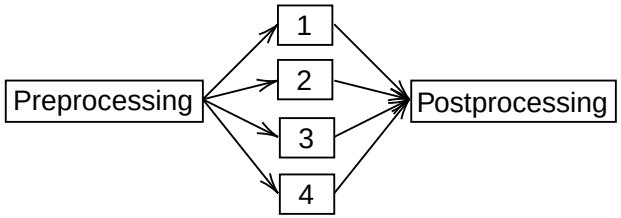
Some problems are *embarrassingly parallel* – i.e., they can be divided into independent sub-problems. Examples include the nonparametric bootstrap or numerical integration (Monte Carlo).

Other problems require their sub-problems to communicate periodically. *Fine-grained parallelism* describes problems that require frequent synchronization between sub-problems. Examples include gradient descent methods for separable loss functions (as used for deep learning). *Coarse-grained parallelism* describes problems that require infrequent synchronization between sub-problems. Examples include parallel tempering (an advanced MCMC technique).

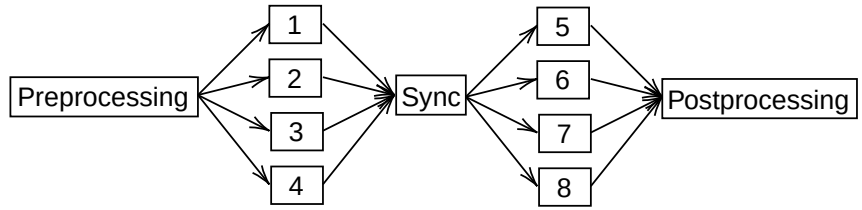
Some problems are inherently sequential and cannot be parallelized at all.

One size does not fit all (Contd.)

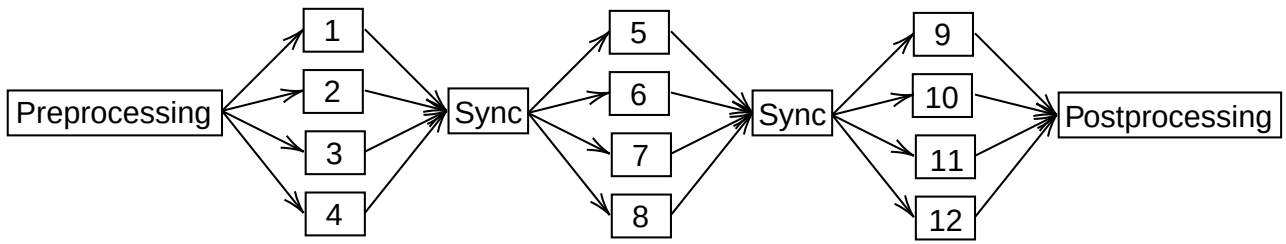
Figure 1: Degrees of Parallelism



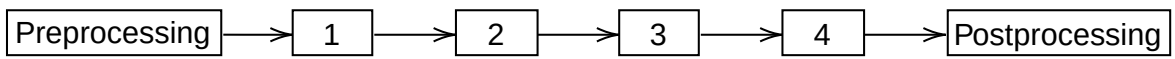
(a) Embarrassingly Parallel



(b) Coarse-Grained Parallelism



(c) Fine-Grained Parallelism



(d) Inherently Sequential

Amdhal's Law

Suppose the problem of interest is at least partially parallelizable. How much can we hope to gain from parallel computing?

Let T denote the runtime of the program per unit of workload W. Suppose that a proportion of p of the program can benefit from parallelization. Amdhal's law gives the upper bound on the possible speedup from parallelization.

$$
\begin{aligned}
& T(s) = (1-p)\,T + p\frac{T}{s} \\
\Rightarrow \quad & \frac{T(1)}{T(s)} = \left[(1-p) + \frac{p}{s}\right]^{-1} \qquad (1) \\
\Rightarrow \quad & \lim_{s\to\infty} \frac{T(1)}{T(s)} = \left[(1-p)\right]^{-1},
\end{aligned}
$$

which gives the maximal theoretical speedup for a program whose share that benefits from parallelization is p.

For example, if 95% of a program benefits from parallelization, the maximal speedup is 20.

Overhead

Amdhal's law is a theoretical upper bound on the maximal speedup. Even if the problem is embarrassingly parallel (i.e., $p = 1$), the realizable speedup in practice is always bounded due to overhead.

Suppose it takes time T to complete the program sequentially. We may hope that a parallel implementation using M processes then it takes $\frac{T}{M}$ time. Unfortunately, in practice we observe $T_{real}(M) = \frac{T}{M} + T_{overhead}(M)$, where the overhead time $T_{overhead}$ increases in the number of processes M.

There are several reasons for that:

- the steps of the algorithms that are executed in parallel take different times to complete, so we must wait for the one that takes the longest;
- data transfer from process synchronizations;
- bottlenecks (e.g., if M exceeds the number of cores, not all processes can run simultaneously).

Overhead (Contd.)

Eventually as we add more processes, $T_{real}(M+1) \geq T_{real}(M)$ since

$$\begin{aligned} & \frac{T}{M+1} + T_{overhead}(M+1) \geq \frac{T}{M} + T_{overhead}(M) \\ \Rightarrow \quad & \frac{T}{M(M+1)} \leq T_{overhead}(M+1) - T_{overhead}(M), \end{aligned} \tag{2}$$

where the last inequality holds for large enough M as the LHS converges to zero but the RHS generally does not.

Adding more processes results in smaller and smaller gains and may even make things slower eventually.

Lesson: benchmark and profile you parallel implementations! In Julia, `BenchmarkTools.jl` and `ProfileView.jl` are excellent tools for this.

A Note on Computational Complexity

It's worth highlighting that parallel computing is not a remedy for high computational complexity: If a program does not scale well, parallel computing is not the solution.

As an example, consider a naive algorithm of matrix multiplication. Take $V \in \mathbb{R}^{n,m}$ and $W \in \mathbb{R}^{m,n}$. From linear algebra, we have

$$VW = \begin{bmatrix} v_1.w_{.1} & \cdots & v_1.w_{.n} \\ \vdots & \ddots & \vdots \\ v_n.w_{.1} & \cdots & v_n.w_{.n} \end{bmatrix}, \text{ where } v_i.w_{.j} = \sum_{k=1}^{m} v_{ik} w_{kj}, \qquad (3)$$

with subscripts denoting row and column-position, respectively. A simple counting exercise then implies that multiplication of V and W using this algorithm takes $N^2(M + M - 1)$ elementary operations (denoted typically by $\mathcal{O}(N^2 M)$).

So even as one new process is added whenever n is increased by 1, runtimes would necessarily increase.

Hardware for Parallel Computing

In addition to a good theoretical grasp of the problem at hand, it is crucial to understand which hardware is suitable for computation.

There are three hardware settings particularly relevant for Econ PhDs:

1. Your laptop.
   - Relatively few cores (say 4-8), of which you can use all but one.
   - Limited memory (say 8-32 GB), all of which is (typically) shared.
2. A computing server (e.g., Acropolis' head node or Google Cloud).
   - Moderate CPU power (say 20 cores) and more memory (+80 GB).
   - As with your laptop, memory is typically shared.
   - Possibly access to a Graphical Processing Unit (GPU).
3. A computing cluster (e.g., Acropolis' computing nodes).
   - Multiple nodes, each with multiple cores (say, 1-25) and good memory (+80GB).
   - Memory is not shared across nodes. Moving data to and from nodes is slow, but memory usage and CPU load can be spread better.

Parallel Computing in Julia

To facilitate a practice-oriented discussion about which hardware is useful in particular settings, we'll focus on parallel computing in Julia.

Today's discussion focuses on two relatively simple (but nevertheless high-payoff) parallel computing techniques:

- Multi-threading, which is idea for parallelized code that you develop on your laptop or are running on a computing server.
- Distributed computing without process-to-process communication, which is perfectly suited for large-scale embarassingly parallel problems that you are running on a computing cluster.

Parallel Computing in Julia

There are two additional parallel computing techniques that we won't cover today, but that you should be aware of:

- Distributed computing with process-to-process communication, which you may need for solving coarsely-grained parallel problems that require the high computing power of a computing cluster (rather than just a single server). If you are interested, `Distributed.jl` is a natural starting point.
- GPU-parallelization, which you may require for solving very large single-instruction multiple-data (SIMD) tasks (e.g., deep learning). For an example, see GPU-vs-CPU-benchmarks-with-Flux.jl. If you are interested, Google Collab provides (limited) GPU resources for free.

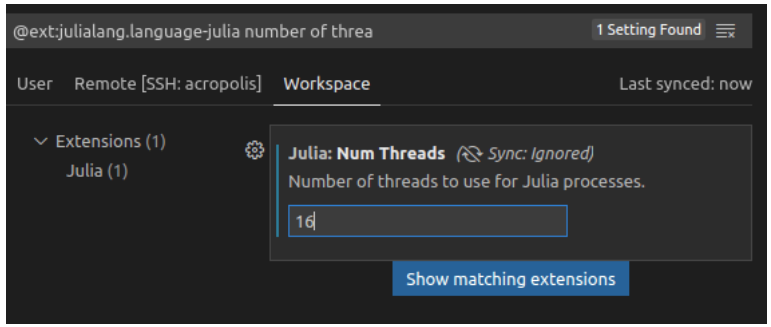
Multi-Threading in Julia

By default, Julia only utilizes a single thread on startup. Depending on your coding setup, there are different ways of enabling multi-threading.

If executing from the terminal, you can call `julia --threads 4` or set your environment variable (e.g., in Linux/Mac this is done via `export JULIA_NUM_THREADS=4`). See the Julia docs for more on his (link).

If you are using VS code (if not, you probably should: link), install the `julialang` extension in VS code, navigate to the extension settings, and set `Num Threads`.

Figure 3: Setting Julia Threads in VS Code



Multi-Threading in Julia

If you are using Jupter notebooks, run the following code (with appropriate thread number) to create a new kernel. Then restart Jupyter notebook and select the new kernel.

Setting up a Multi-Threaded Jupyter Kernel

```
using IJulia
IJulia.installkernel("Julia␣with␣4␣threads", env=Dict(
    "JULIA_NUM_THREADS" => "4",
))
```

To check whether your Julia session uses the desired number of threads, use `nthreads()`.

Checking the Number of Available Threads

```
using Base.Threads
nthreads()
julia> 4
```

Example 1: Multi-Threaded Monte Carlo

As a first example of how multi-threading can be leveraged to speed up computations, consider a simple Monte Carlo simulation for assessing finite sample properties of the least squares coefficient in a particular DGP.

1: Required input: P_θ a generative model of the data, N the number of observations, J the number of simulations;
2: **procedure** MC
3: **for** $j \in \{1, \ldots, J\}$ **do**
4: $\mathcal{D}_j \leftarrow \{(y_i, x_i)\}_{i=1}^N$, where $(y_i, x_i) \overset{iid}{\sim} P_\theta, \forall i$
5: $\hat{\beta}_j \leftarrow \textsc{getOLS}(\mathcal{D}_j)$
6: Return: $\{\hat{\beta}_j\}$

Note that this procedure is embarrassingly parallel so we may hope that it benefits noticeably from parallelization.

Example 1: Multi-Threaded Monte Carlo (Contd.)

The below gives a quick single-threaded implementation in Julia. Using the `@btime` macro from `BenchmarkTools.jl` indicates a runtime of approximately 0.47 seconds for a reasonably-sized problem.

Single-Threaded Monte Carlo

```
# Define single-threaded monte carlo
function simple_mc(nobs, nX, nsim)
    beta_mat = zeros((nsim, nX))
    for s in 1:nsim
        # Simulate data
        X = randn((nobs, nX))
        y = X * ones(nX) + randn(nobs)
        # Estimate the ols coefficient
        beta_mat[s, :] = inv(X' * X) * (X' * y)
    end
    return beta_mat
end

# Run benchmark
@btime simple_mc(10000, 3, 1000);
julia> 494.110 ms (16002 allocations: 460.47 MiB)
```

Example 1: Multi-Threaded Monte Carlo (Contd.)

A naive implementation of multi-threaded Monte Carlo would simply place the `@threads` macro before the `for`-loop. But this ignores that `randn` is state-dependent. To avoid issues with simultaneous memory changes, it's better to pass a random number generator for each thread.

Multi-Threaded Monte Carlo

```
# Check available threads
nthreads()
julia> 2

# Define multi-threaded monte carlo
function par_simple_mc(nobs, nX, nsim)
    beta_mat = zeros((nsim, nX))
    rnglist = [MersenneTwister() for i in 1:nthreads()]
    @threads for s in 1:nsim
        # Simulate data
        X = randn(rnglist[threadid()], (nobs, nX))
        y = X * ones(nX) + randn(rnglist[threadid()], nobs)
        # Estimate the ols coefficient
        beta_mat[s, :] = inv(X' * X) * (X' * y)
    end
    return beta_mat
end

# Run benchmark
@btime par_simple_mc(10000, 3, 1000);
julia> 259.005 ms (16038 allocations: 460.51 MiB)
```

Example 1: Multi-Threaded Monte Carlo (Contd.)

The speedup from using two threads in the parallelized Monte Carlo simulation is close to 2, suggesting little overhead. Ignoring variance in the benchmarks, the runtimes suggest approximately 0.012 seconds are spent on overhead (approximately 4.6% of the multi-threaded runtime).

Unfortunately, quick comparison of runtimes with different numbers of threads is not super quick in Julia (because you'd need to start it up each time), but it's an excellent exercise! (R's `doSNOW` or `foreach` packages make those comparisons a bit easier: you should check those, too.)

Example 2: Multi-Threaded EM

Gaussian mixture models are frequently used in IO and quantitative marketing to approximate individual heterogeneity.

A simple type of mixture model is the univariate Gaussian mixture model. Let $\phi(y)$ denote the pdf of a standard normal evaluated at $y \in \mathbb{R}$, then the pdf of a Gaussian mixture model with K components is given by

$$\sum_{k=1}^{K} \pi_k \sigma_k^{-1} \phi\left(\frac{y - \mu_k}{\sigma_k}\right), \tag{4}$$

where $\{(\mu_k, \sigma_k)\}$ are the (unknown) component-specific parameters and $\{\pi_k\}$ are the (unknown) component-membership probabilities. See here for a visualization.

Notice that the loglikelihood associated with (4) does not simplify conveniently (because we are taking the log of a sum).

Dempster et al. (1977) develop the expectation-maximization (EM) algorithm that allows for iterative optimization of the log-likelihood.

Example 2: Multi-Threaded EM (Contd.)

The below procedure – a simple EM algorithm for a univariate Gaussian Mixture – is an example of fine-grained parallelism.

1: Required input: $\{y_i\}_{i\in\mathcal{D}}$ the data, K the number of mixture components, $\{(\mu_k, \sigma_k, \pi_k)\}$ initial parameters and probabilities;
2: **procedure** EM
3: **repeat**
4: **for** $k \in \{1, \ldots, K\}$ **do** ▷ E-step
5: $\ell_{ik} \leftarrow \frac{1}{\sigma_k}\phi\left(\frac{y_i-\mu_k}{\sigma_k}\right), \quad \forall i$
6: $\tilde{\pi}_{ik} \leftarrow \frac{\ell_{ik}\pi_k}{\sum_{j=1}^{K}\ell_{ij}\pi_j}, \quad \forall i, k$
7: **for** $k \in \{1, \ldots, K\}$ **do** ▷ M-step
8: $\pi_k \leftarrow |\mathcal{D}|^{-1}\sum_i \tilde{\pi}_{ik}$
9: $\mu_k \leftarrow \frac{\sum_i y_i\tilde{\pi}_{ik}}{\sum_i \tilde{\pi}_{ik}}$
10: $\sigma_k^2 \leftarrow \frac{\sum_i (y_i-\mu_k)^2\tilde{\pi}_{ik}}{\sum_i \tilde{\pi}_{ik}}$
11: **until** convergence
12: Return: $\{(\mu_k, \sigma_k, \pi_k)\}$

Example 2: Multi-Threaded EM (Contd.)

Custom Gaussian Mixture Type

```
mutable struct myGaussMix
    mu # component means
    sgm # component standard devs
    probs # component probabilities
    probs_post # posterior component probabilities
    y # data
    K # number of components

    function myGaussMix(y, K)
        # Initialize component parameters randomly
        mu = rand(Normal(0, std(y)), K)
        sgm = std(y) * ones(K) ./ K
        probs = rand(Dirichlet(ones(K)))
        probs_post = zeros((nobs, K))
        # Construct object
        new(mu, sgm, probs, probs_post, y, K)
    end
end
```

(Specifying a custom type like this may be beneficial if we also want to simulate from the mixture after fitting it to the data. But I have my doubts about whether this is the most computationally efficient approach.)

Example 2: Multi-Threaded EM (Contd.)

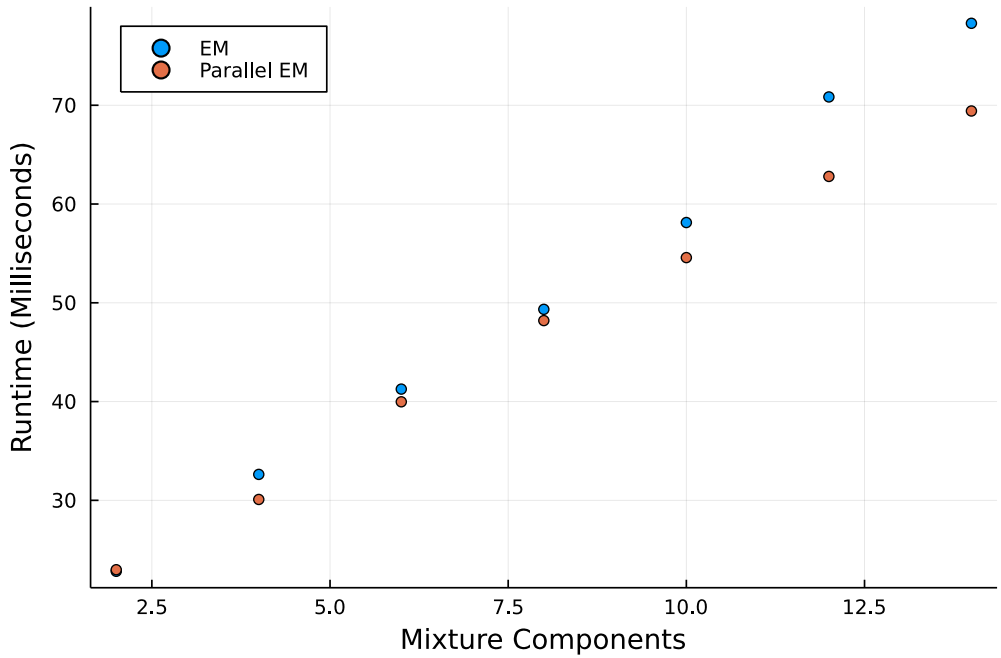
Multi-Threaded EM

```
function par_fit!(m::myGaussMix; max_iter = 1000, tol = 1e-6)
    # Get values from m
    mu = m.mu; sgm = m.sgm
    probs = m.probs; probs_post = m.probs_post
    K = m.K; nobs = length(m.y)
    # Run EM algorithm
    likeli = zeros((nobs, K))
    log_l = zeros(max_iter)
    log_l_old = -9e6
    for j in 1:max_iter
        # E-step
        @threads for k in 1:K
            likeli[:, k] = pdf(Normal(mu[k], sgm[k]), m.y) .* probs[k]
        end
        probs_post = likeli ./ repeat(mapslices(sum, likeli, dims = 2)', K)'
        # M-step
        @threads for k in 1:K
            probs[k] = mean(probs_post[:, k])
            mu[k] = sum(m.y .* probs_post[:, k]) ./ sum(probs_post[:, k])
            sgm[k] = sum((m.y .- mu[k]).^2 .* probs_post[:, k]) ./ sum(probs_post[:, k])
            sgm[k] = sqrt(sgm[k])
        end
        # Compute current value of the likelihood
        mixture_j = MixtureModel(Normal, [(mu[k], sgm[k]) for k in 1:K], probs)
        log_l[j] = sum(logpdf(mixture_j, m.y))
        # Check for convergence
        (log_l[j] - log_l_old) < tol ? break : nothing
    end
    # Export values to m
    m.mu = mu; m.sgm = sgm
    m.probs = probs; m.probs_post = probs_post
    return nothing
end
```

Example 2: Multi-Threaded EM

The runtime of the multi-threaded EM implementation does not improve greatly compared to the single-threaded EM. This suggests considerable overhead – too bad! Maybe there is a benefit for "massive mixtures" (i.e., very many components), but those appear rare in economic applications.

Figure 4: Runtime Comparison for a Simple EM Algorithm



Notes. Runtime is for 50 iterations of a randomly initialized EM algorithm on 20,000 datapoints generated from a 20-component univariate Gaussian mixture model. The parallel EM is run on the Acropolis head node using 16 threads (which is not a very nice thing to do!).

Getting Started with Acropolis

Acropolis is the computing cluster of the social science division. It consists of a single head node (40 cores, 2TB of memory) and 32 compute nodes (each with 28 cores and 250GB of memory).

Knowing how to leverage these resources can greatly help with your computational needs. When working on a moderately sized problem that requires more memory than your laptop has, developing on the head node is a good solution (but be mindful of others). Leveraging the cores from the compute nodes is ideal for embarrassingly parallel problems, where each instance takes a considerable amount of time.

If you have an SSD affiliation, you can easily request an account (link).

If you have a booth affiliation, you get can access to the Mercury computing cluster (link). Mercury uses different scheduler so submission files are a bit different, but their documentation is *excellent*!

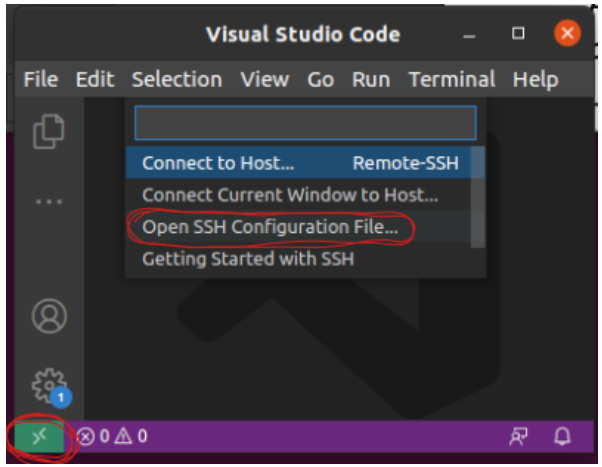
Getting Started with Acropolis (Contd.)

Once you have an account, there are various ways of connecting to Acropolis (or Mercury):

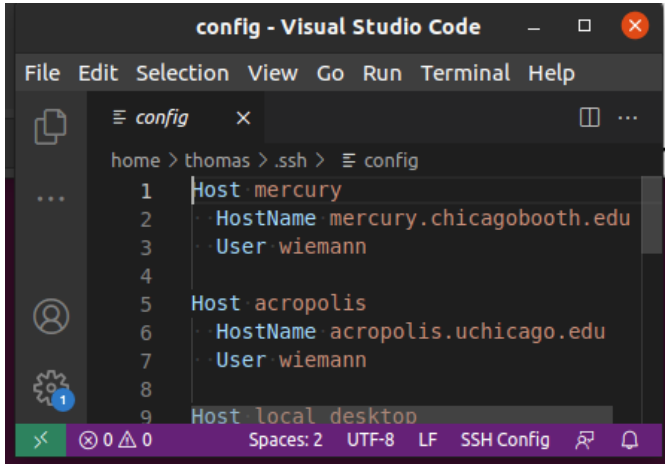
- EasyVNC provides a graphical interface (link), which can be convenient for simple organizational tasks (e.g., moving data) but you should probably not use it for active development.
- Your terminal (in Linux/Mac) or Putty (in Windows) allows you to connect via SSH (link). This is particularly useful for quickly checking the status of your programs on the cluster or `git` operations, but – again – not great for active development.
- Jupyter notebook and VS code allow for remote development on the cluster. My (or better: Ed's) recommendation: use the VS code extension `ms-vscode-remote` (link).

Connecting to Acropolis' Head Node via VS Code

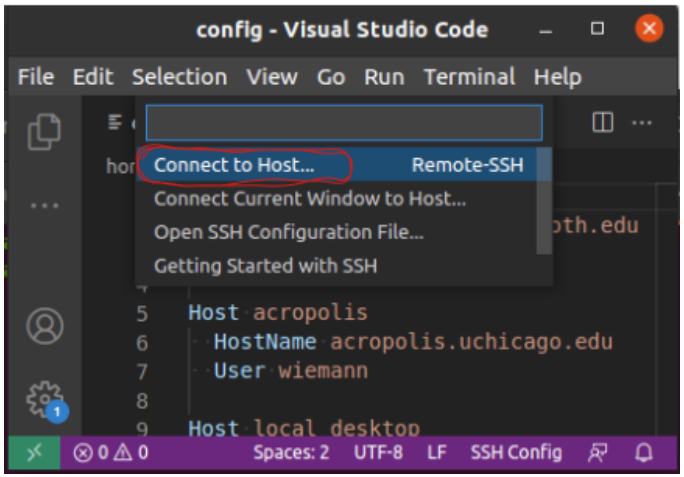
Figure 5: Key steps in setting up VS Code with Acropolis



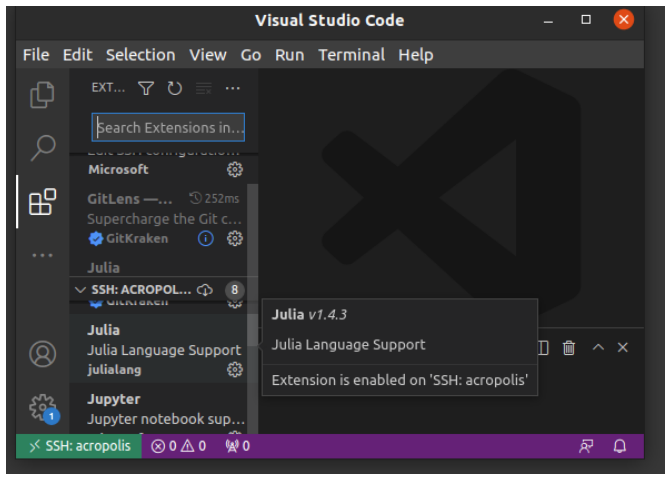
(a) Configure SSH,



(b) enter your CNetID,



(c) then connect to acropolis,



(d) and install vs-code extensions.

Example 3: Distributed Monte Carlo

To provide a simple but concrete illustration on how to leverage Acropolis for embarrassingly parallel programs, we will run through a simple data-dependent simulation based on Berry et al. (1995) and inspired roughly by Chernozhukov et al. (2015).

Data dependent-simulations are generally much more interesting than just making up a DGP entirely on your own: it speaks about the performance of a method in a (practically relevant) economic setting. It also ties the researchers hands (a little bit), so others might also find it a more valuable signal.

For an empirical example of data-dependent simulations, see Angrist and Frandsen (2020) (and TA Session 3!). Interesting related readings include Athey et al. (2021) and Ferman (2021).

Example 3: Distributed Monte Carlo (Contd.)

Let $\{(y_i, d_i, x_i, z_i)\}_{i \in \mathcal{D}}$ denote the observed sample. The considered data simulation procedure is as follows:

1. Calculate $\hat{\tau}_{TSLS}$ via TSLS on the original data. Let $\hat{\gamma}_{FS}$ and $\hat{\beta}_{SS}$ denote the first and second-stage coefficients. Approximate $Var(\epsilon|X)$ via linear regression of the squared TSLS residuals on the controls. Denote the corresponding function of the controls by $\tilde{\omega}$.

2. Sample from the empirical distribution of (X, Z) by bootstrapping B individuals from the original data. Denote the sample $\mathcal{D}_b$.

3. Simulate $\nu_i \overset{iid}{\sim} \mathcal{N}(0, \kappa_1)$, where κ_1 is a simulation hyperparameter. Set $\tilde{d}_i^{(b)} := [x_i^\top, z_i^\top]\hat{\gamma}_{FS} + \nu_i, \forall i \in \mathcal{D}_b$.

4. Simulate $\varepsilon_i \overset{iid}{\sim} \mathcal{N}(0, \kappa_2)$, where κ_2 is a simulation hyperparameter. Set $y_i^{(b)} := [d_i, x_i^\top]\hat{\beta}_{SS} + \tilde{\omega}(x_i)(\varepsilon_i + \rho\nu_i)$, where τ_0 is the chosen value of the parameter of interest and ρ is the simulation hyperparameter that governs the endogeneity bias.

Example 3: Distributed Monte Carlo (Contd.)

The simulation hyperparameters $(\kappa_1, \kappa_2, \rho)$ are determined to (approximately) match moments in the data:

- κ_1 is chosen such that $Var_n(\tilde{D}) \approx Var_n(D)$,
- κ_2 and ρ are jointly determined such that $Var_n(\tilde{Y}) \approx Var_n(Y)$ and the OLS coefficients on D in the true data and the simulated data are approximately equal.

The Monte Carlo simulation then amounts to: 1) simulating from the BLP-based DGP, and 2) calculating the TSLS coefficient and corresponding standard errors. Key parameters of interest are the median absolute bias of the coefficient on D and the coverage rate of a 95% confidence interval.

Apart from the initial estimation of the TSLS model on the Berry et al. (1995) data and the finel aggregation of the coefficient estimates, the procedure is embarrassingly parallel.

Example 3: Distributed Monte Carlo (Contd.)

Acropolis allows us to use 350 cores simultaneously, but each compute node only has 28 cores. Instead of coding up a multi-threaded procedure as before, we therefore proceed as follows:

1. Create a script `blm95_sim.jl` that 1) loads the Berry et al. (1995) data, 2) fits the TSLS to the original data, 3) simulates from the generative model defined previously, 4) estimates the TSLS coefficient on the simulated data, and 5) saves the coefficient and standard error to a `.csv` file.
2. Create a `qsub` submission script that requests 350 instances, each with 1 CPU core and 4GB of memory, and schedules 1000 executions of the `blm95_sim.jl` script.
3. Submit the `qsub` script via the terminal in Acropolis and wait for the simulations to finish. This will generate 1000 `.csv` files.
4. Create and execute a script `combine_sim.jl` that loads and aggregates the files from our simulation, and stores the output in a new `.csv` file.

You can download the project folder here if you want to try it out. Copy the folder to Acropolis and read `README.md` for instructions.

Example 3: Distributed Monte Carlo (Contd.)

qsub Sumbission Script

```
#!/bin/bash
#PBS -N sim_BLP95 # Name of the job
#PBS -j oe # joins output and error stream
#PBS -l nodes=1:ppn=1,mem=4gb # Per job, use 1 node w/ 1 CPU and 4GB RAM
#PBS -o temp/log # directory of the log files
#PBS -m abef # email options
#PBS -M wiemann@uchicago.edu # email destination
#PBS -V # Export environment variables to batch job
#PBS -t 1-100 # Number of jobs to run

# Set the working directory to the directory from which the file was called
cd $PBS_O_WORKDIR

#Run the julia script
julia "julia_scripts/blp95_sim.jl"
```

Question: Suppose the script `blp95_sim.jl` is multi-thread enabled. How would you adapt the submission script?

Answer: Change the requested resources per node (e.g., change `ppn=1` to `ppn=4`, and add `--threads 4` after `julia` in the last line.

Example 3: Distributed Monte Carlo (Contd.)

Table 1: Simulation Results

Mean Bias	Median Absolute Bias	Cov. Rate (95%)
0.001	0.008	1.00

Notes. Results are based on 100 simulations adapted from Berry et al. (1995) and Chernozhukov et al. (2015).

Table 1 gives the results from the simple exercise. We may view the result as somewhat reassuring, as TSLS performs reasonably well despite the relatively small sample of $N = 2217$ observations in Berry et al. (1995).

Other interesting avenues include deviating from the linear TSLS model. E.g., instead of generating from a linear fist and second stage, generate from a partially linear IV model as in Chernozhukov et al. (2018).

(My original plan was to show you results where the first and second stage were nonlinear (but additively separable) in X and (X, Z). Using acropolis in such a setting makes more sense because a single simulation instance takes much more time.)

Last Comments

Before you avidly start parallelizing all your code, it's worth emphasizing that other programming paradigms should not be disregarded.

- Avoid premature optimization. It's generally a good idea to start with a single-threaded implementation. If the runtime is concerning you, start with profiling and check whether the runtime-intensive parts of the code would actually benefit from parallelization.
- Computers are cheap and thinking hurts. Always value your time more than the computer's time. If you spend hours on a negligible speedup, that's typically not time well spent.
- Keep it simple, stupid. You should strive to keep your solutions simple and transparent. Parallelization is not an excuse for complicating your code.

References

Angrist, J. and Frandsen, B. (2020). Machine labor. NBER Working Paper No. 26584.

Athey, S., Imbens, G. W., Metzger, J., and Munro, E. (2021). Using wasserstein generative adversarial networks for the design of monte carlo simulations. *Journal of Econometrics*.

Berry, S., Levinsohn, J., and Pakes, A. (1995). Automobile prices in market equilibrium. *Econometrica*, pages 841–890.

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W., and Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters: Double/debiased machine learning. *The Econometrics Journal*, 21(1).

Chernozhukov, V., Hansen, C., and Spindler, M. (2015). Post-selection and post-regularization inference in linear models with many controls and instruments. *American Economic Review: Papers & Proceedings*, 105(5):486–90.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22.

Ferman, B. (2021). A simple way to assess inference methods. *arXiv preprint arXiv:1912.08772*.

Visualizing Gaussian Mixtures

Figure 7: Simulated Gaussian Mixture with Four Components