ShockNet

Parallelizing Financial Contagion Modeling

By Nicholas Ching (nc2935), Erica Choi (eyc2130)

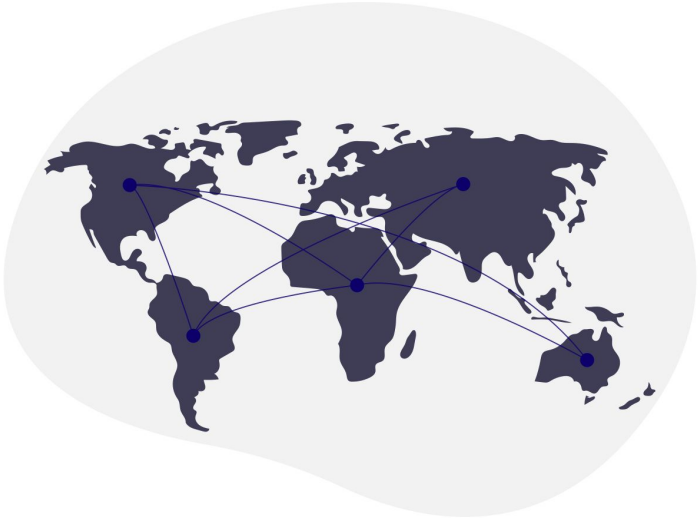
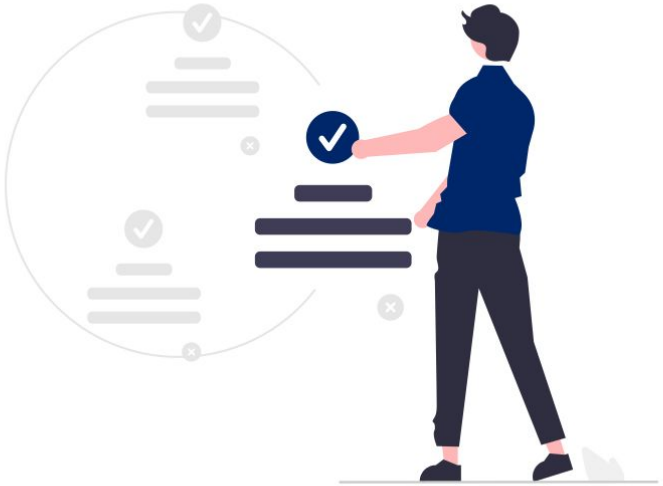


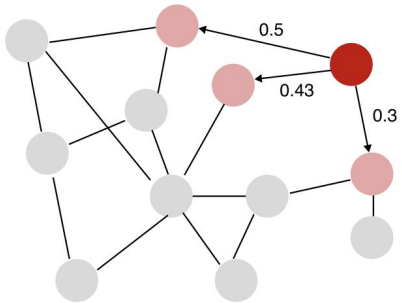
Table of Contents



Why Financial Contagion Modeling

The banking system is fundamentally linked. If one bank fails, what other banks will fail?

This problem is a graph problem



Given a seed set which neighbors will eventually fail?

The problem with stochasticity

- Need to use monte carlo simulations to model realistic spread
- When graphs become exponentially large, we are limited by computer / time

Parallelism is the answer

The Problem More Formally

Modeling financial shocks is an example of influence maximization. Our project focuses on the Independent Cascade Model (ICM)

Representing the network

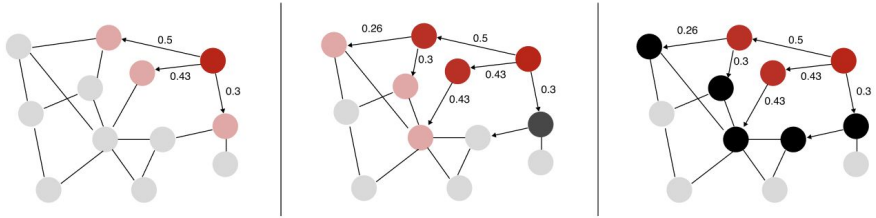
Directed Graph: $G = (V, E)$ where V denotes a set of entities and E denotes the set of relationships.

Each edge has an assigned weight $w(e) \in [0.1, 0.5]$ representing the probability of influence propagation

Objective

Given some seed set S, compute the expected number of "infected" nodes within the network

The Independent Cascade Model Algorithm



1. Mark the nodes in *S* as infected.
2. For each neighbor *e* of infected nodes, generate a random number r $\in$ [0, 1]. If $r \leq w(e)$, *e* is added to the set of infected nodes.
3. Repeat step 2 until no new nodes are infected.
4. The output of the simulation is the size of the infected nodes.
5. Repeat steps 1~4 N times and take the average of the outputs

Approaching Monte Carlo

Parallelizing the ICM is about parallelizing Monte Carlo. Its instructive, but there are some tricks

Properties that make monte carlo parallelizable

- Simulations are encapsulated. There is minimal data exchange between each simulation
- Each trial is fundamentally independent.
- No global state

In theory, Monte Carlo is embarrassingly parallel

Techniques we used

- Static Chunking
- parMap & rdeepseq
- Split Random Generation (more on this later)

Should technically speed things up…

But…

An overview of our development

Theory can only take you so far…

Sequential Implementation

Compute Time: 7m 14s

- Base implementation of Graph Building
- Implementation of Independent Cascade Model
- First implementation of Monte Carlo Simulations

Unoptimized Parallelization

Compute Time: 13m 14s

- Utilize static chunking
- Utilize parMap and rdeepseq
- **Terrible execution time** because of **singular random number generator**

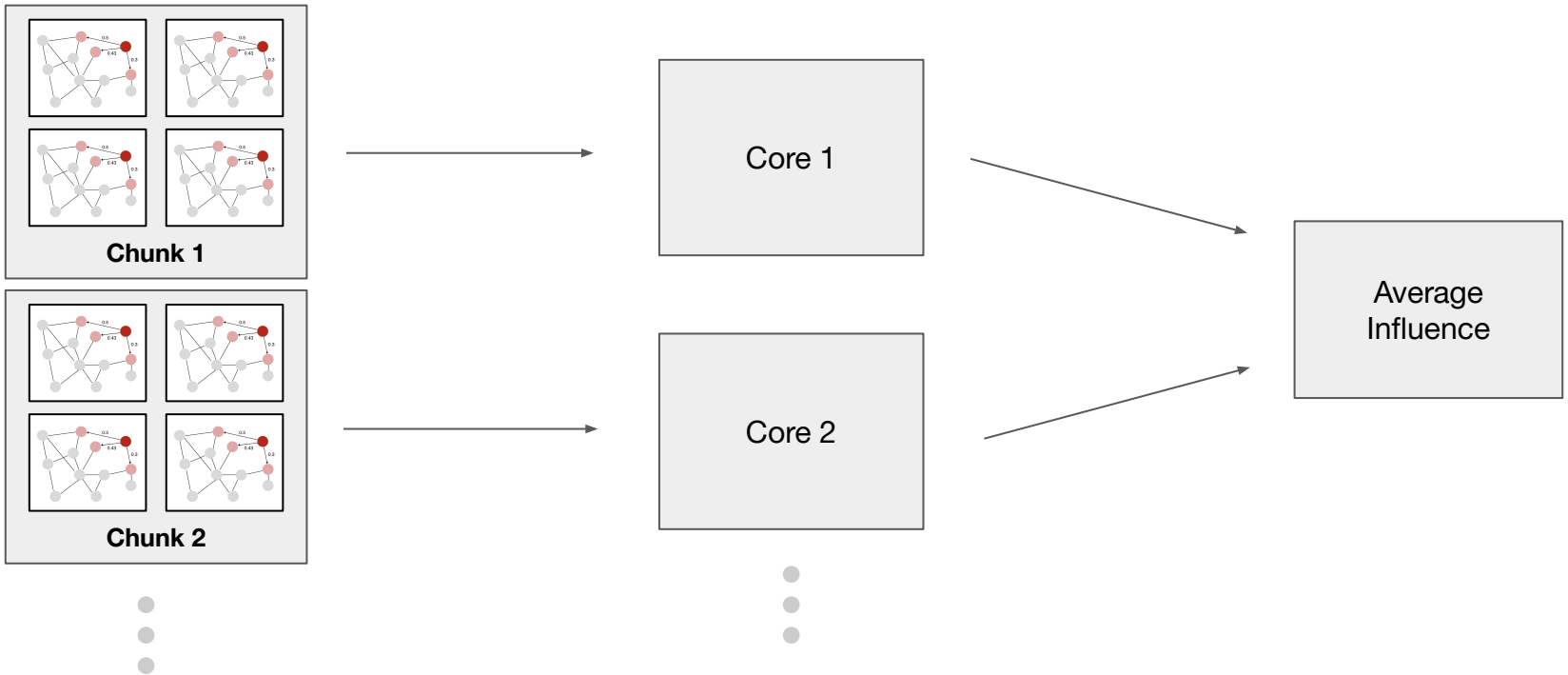
Optimized Parallelization

Compute Time: 1m 10s

- Utilize static chunking
- Utilize parMap and rdeepseq
- Utilize stdgen to split random number generator

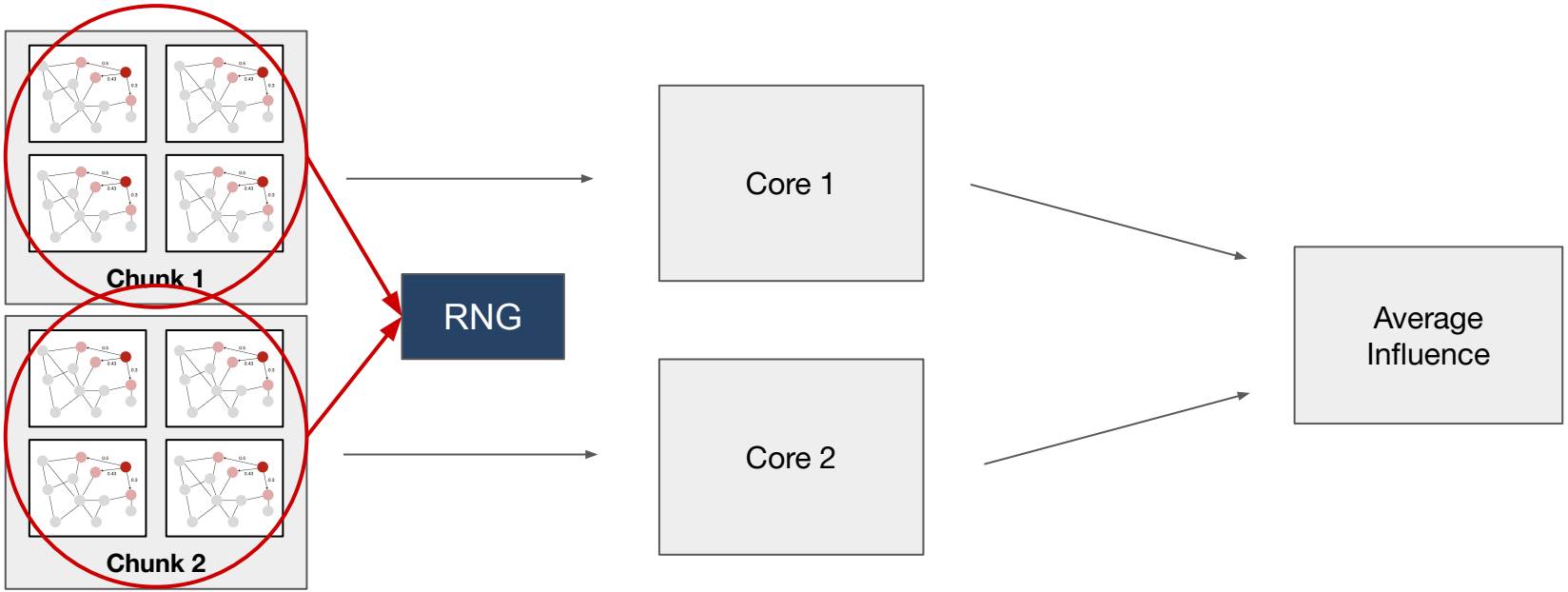
Our First Implementation

On paper this works. We distribute workload across cores, but are facing a problem



Why our first iteration failed

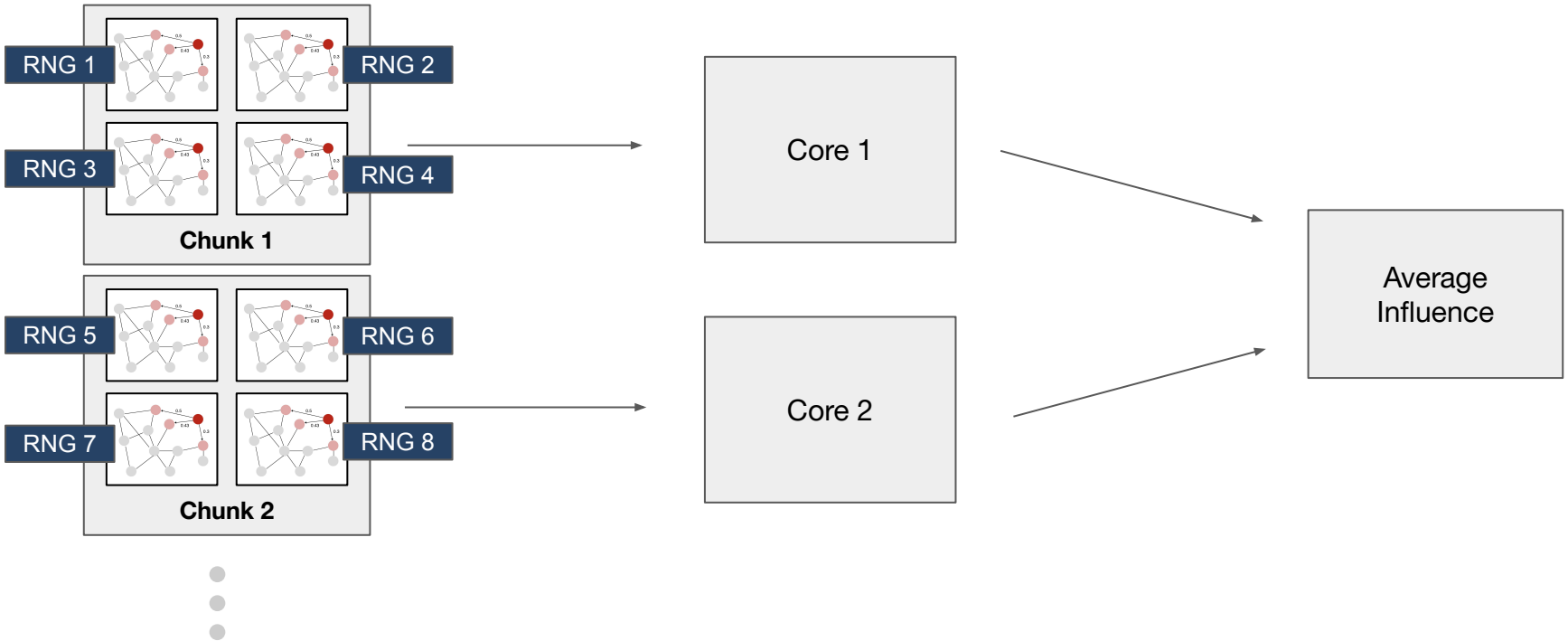
All of these simulations are accessing a single random number generator (RandomRIO)



Recall that each simulation makes calls to the RNG for each recursive propagation.
Multiplied by thousands of simulations and you have **a HUGE bottleneck**

Optimized Version

Creating independent RNG's **solves the bottleneck** and gives us **statistical independence**.



What Haskell features did we use?

Haskell makes parallelization easier…

parMap

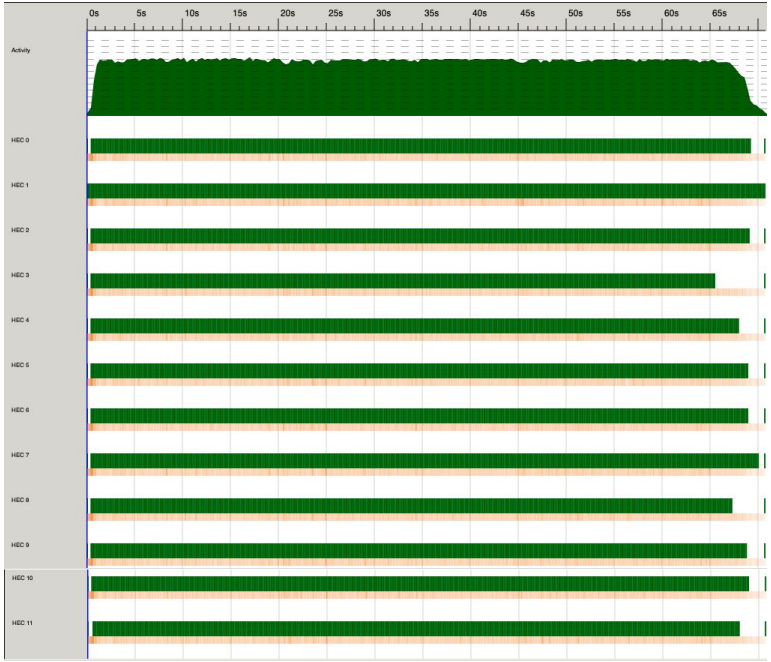
- Abstract chunk distribution to available cores
- Creates spark to be processed in parallel
- No need for low-level operations

rdeepseq

- Ensure each parallel chunk of work is fully evaluated
- “Forces” the compiler to fully execute parallel work instead of returning thunks

From Haskell, we avoid the need to manually handle mutable data states, data races and synchronization primitives

Findings



Optimized Parallel ICM Threadscope

```
~/De/Col/1. Parallel Functional Programming/shocknet main !2 ?13 ❯ time ./parallel_icm
Building graph...
Graph built. Running Monte Carlo simulations...
Average Influence (Monte Carlo): 26574.094
 638,731,222,504 bytes allocated in the heap
  47,954,528,320 bytes copied during GC
      77,935,424 bytes maximum residency (274 sample(s))
       1,292,936 bytes maximum slop
             285 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     19870 colls, 19870 par   56.524s  10.998s     0.0006s    0.0055s
  Gen  1       274 colls,   273 par   20.966s   2.593s     0.0095s    0.0281s

  Parallel GC work balance: 75.98% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 12 (11 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.010s  (  0.010s elapsed)
  MUT     time  551.244s  ( 57.121s elapsed)
  GC      time   77.489s  ( 13.591s elapsed)
  EXIT    time    0.005s  (  0.004s elapsed)
  Total   time  628.749s  ( 70.726s elapsed)

  Alloc rate    1,158,708,427 bytes per MUT second

  Productivity  87.7% of total user, 80.8% of total elapsed

./parallel_icm +RTS -N12 -ls -s  609.01s user 19.75s system 888% cpu 1:10.75 total
```

Spark usage Parallel ICM scope

Questions?

Appendix: How are we building graphs?

What we are given

- Number of nodes our graph will have
- Number of edges we want
- An empty set

- Randomly pick a pair from the set of available nodes. Only keep unique pairs
- Randomly assign each pair a random weight between [0.1, 0.5]
- Take the list of weighted edges and create a graph data structure.

- Gr() a b: A Gr is a parameterized graph type where:
  - a is the type of data stored at each node.
  - b is the type of data stored on each edge.
- The tree structure we are using is haskell's patriciaTree implementation
- Just searched online for tree structures to use.
- Its efficient, and has some abstractions that we take advantage (like lsuc which gets successor nodes)

Appendix: How does split work?

```
stdGen <- getStdGen
let gens = take numSims $ iterate (snd . split) stdGen
```

```
split :: g -> (g, g)
```

We are recursively splitting stdGen

[stdGen,
  snd (split stdGen),
  snd (split (snd (split stdGen))),
  snd (split (snd (split (snd (split stdGen))))),
  …
]

- Split will take StdGen as input, and return two StdGen type classes
- We use this function to split StdGen into as many simulations as we want to run,

Appendix: Aggregating Results

```
    let results = parMap rdeepseq (\genChunk ->
                      sum [ evalRand (simulateOnce graph seeds) gen | gen <- genChunk ]
                  ) chunks
```

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

```
rdeepseq :: NFData a => Strategy a
```

`rdeepseq` fully evaluates its argument.

- In this case, each chunk gives us an intermediate result.
- These intermediate results are a part of the list [b], and are summed to get a final result
- We are doing most of the summing computations in parallel. (only chunk computations sequentially)
- This approach reduces the number of calculations for free