Helia & Performance

How to wring the most out of your deployment

Helia

IP Stewards Team

Maintainer

- Helia
- js-libp2p
- Many supporting libraries

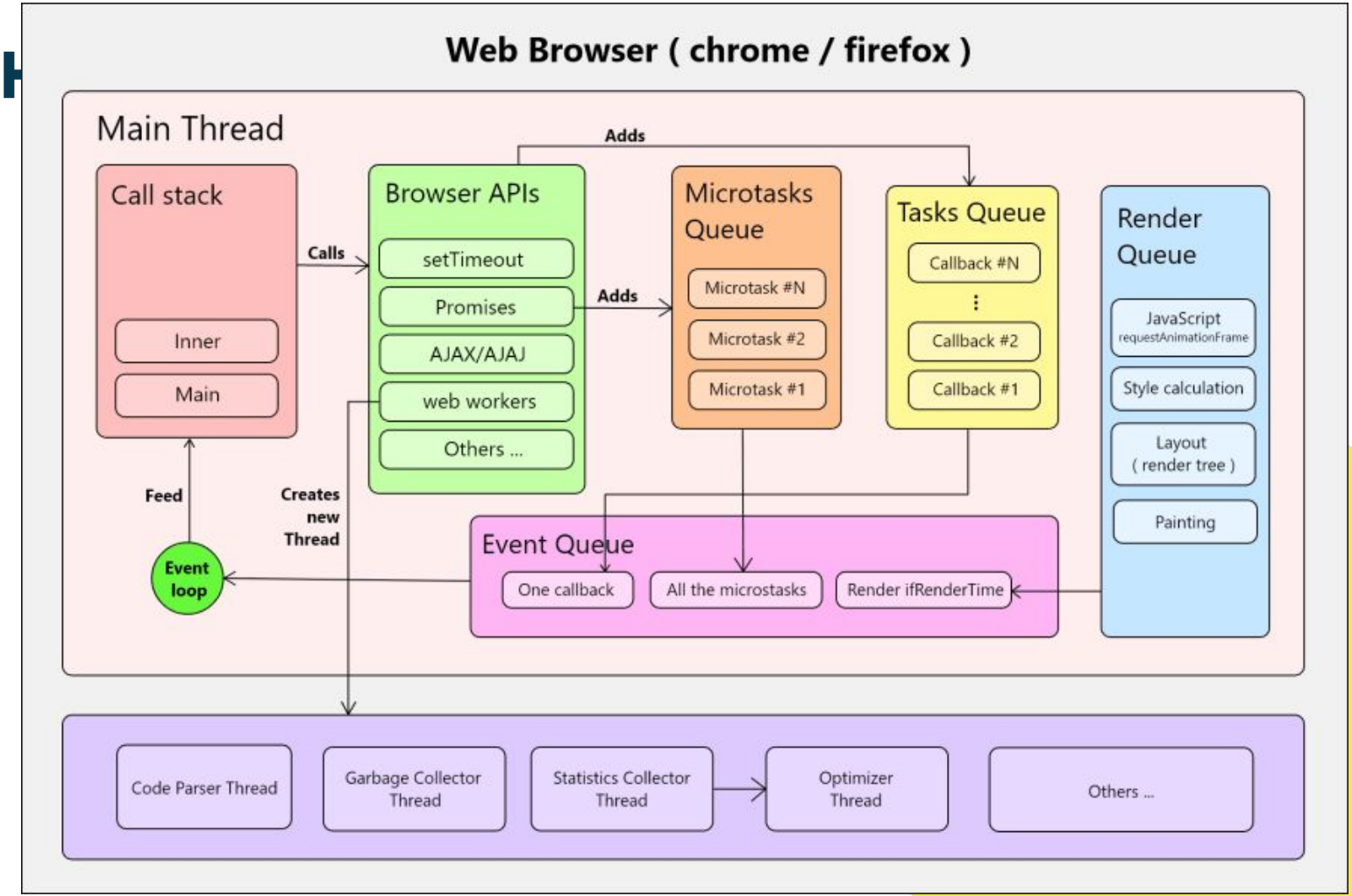
Alex Potsides
Protocol Labs

JavaScript is slow?

Yes, if used badly

- CPU
- async

Async is not free


Web Browser (chrome / firefox)
Main Thread
Call stack
Inner
Main
Calls
Browser APIs
setTimeout
Promises
AJAX/AJAJ
web workers
Others ...
Adds
Adds
Microtasks Queue
Microtask #N
Microtask #2
Microtask #1
Tasks Queue
Callback #N
Callback #2
Callback #1
Render Queue
JavaScript
requestAnimationFrame
Style calculation
Layout
(render tree)
Painting
Feed
Creates new Thread
Event loop
Event Queue
One callback
All the microstasks
Render ifRenderTime
Code Parser Thread
Garbage Collector Thread
Statistics Collector Thread
Optimizer Thread
Others ...

Demo

Helia

Storage matters

Helia

- I/O is a massive bottleneck
- Blockstores can be slow

S3 Block/Datastores

Best practices design patterns: optimizing Amazon S3 performance

PDF | RSS

Your applications can easily achieve thousands of transactions per second in request performance when uploading and retrieving storage from Amazon S3. Amazon S3 automatically scales to high request rates. For example, your application can achieve at least 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second per partitioned prefix. There are no limits to the number of prefixes in a bucket. You can increase your read or write performance by using parallelization. For example, if you create 10 prefixes in an Amazon S3 bucket to parallelize reads, you could scale your read performance to 55,000 read requests per second. Similarly, you can scale write operations by writing to multiple prefixes. For more information about creating and using prefixes, see Organizing objects using prefixes.

https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html

S3 Block/Datastores

Best practices design patterns: optimizing Amazon S3 performance

PDF | RSS

Your applications can easily achieve thousands of transactions per second in request performance when uploading and retrieving storage from Amazon S3. Amazon S3 automatically scales to high request rates. For example, your application can achieve at least 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second per partitioned prefix. There are no limits to the number of prefixes in a bucket. You can increase your read or write performance by using parallelization. For example, if you create 10 prefixes in an Amazon S3 bucket to parallelize reads, you could scale your read performance to 55,000 read requests per second. Similarly, you can scale write operations by writing to multiple prefixes. For more information about creating and using prefixes, see Organizing objects using prefixes.

https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html

S3 Blockstores

Default sharding strategy:

CID(bafyfoo…abcdef)
-> /fe/bafyfoo…abcdef

https://github.com/ipfs/js-stores/blob/main/packages/blockstore-s3/src/sharding.ts#L20

Other stores are available:

- **MountDatastore**
  - Combine multiple datastores separated by key prefixes
  - You might use a memory datastore for the /peers prefix - all peer data will be stored in memory, everything else on disk

https://npmjs.com/package/datastore-core
https://npmjs.com/package/blockstore-core

Other stores are available:

- **TieredDatastore**
  - Combine multiple datastores
  - Write to all, read from whichever is fastest?

https://npmjs.com/package/datastore-core
https://npmjs.com/package/blockstore-core

Maybe invent your own?

- **HotContentBlockstore?**
  - Cache the 1000 most requested blocks in memory?

It’s up to you!

https://npmjs.com/package/interface-datastore
https://npmjs.com/package/interface-blockstore

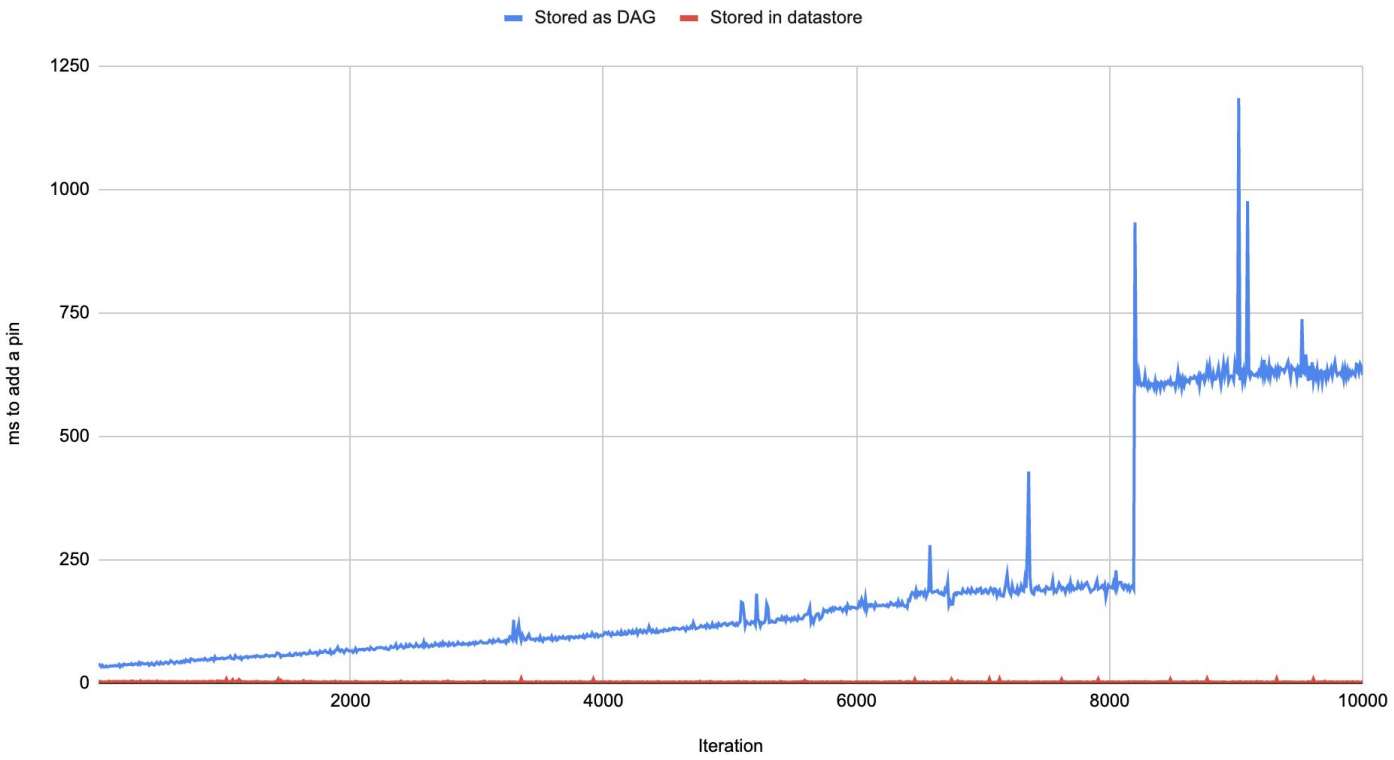
Content routing matters

- **Resolving content takes a long time**
- **Configure delegates**
  - **Streaming (fast)**
    - @libp2p/ipni-content-routing
  - **Non-streaming (slow)**
    - @libp2p/reframe-content-routing
    - @libp2p/delegated-content-routing

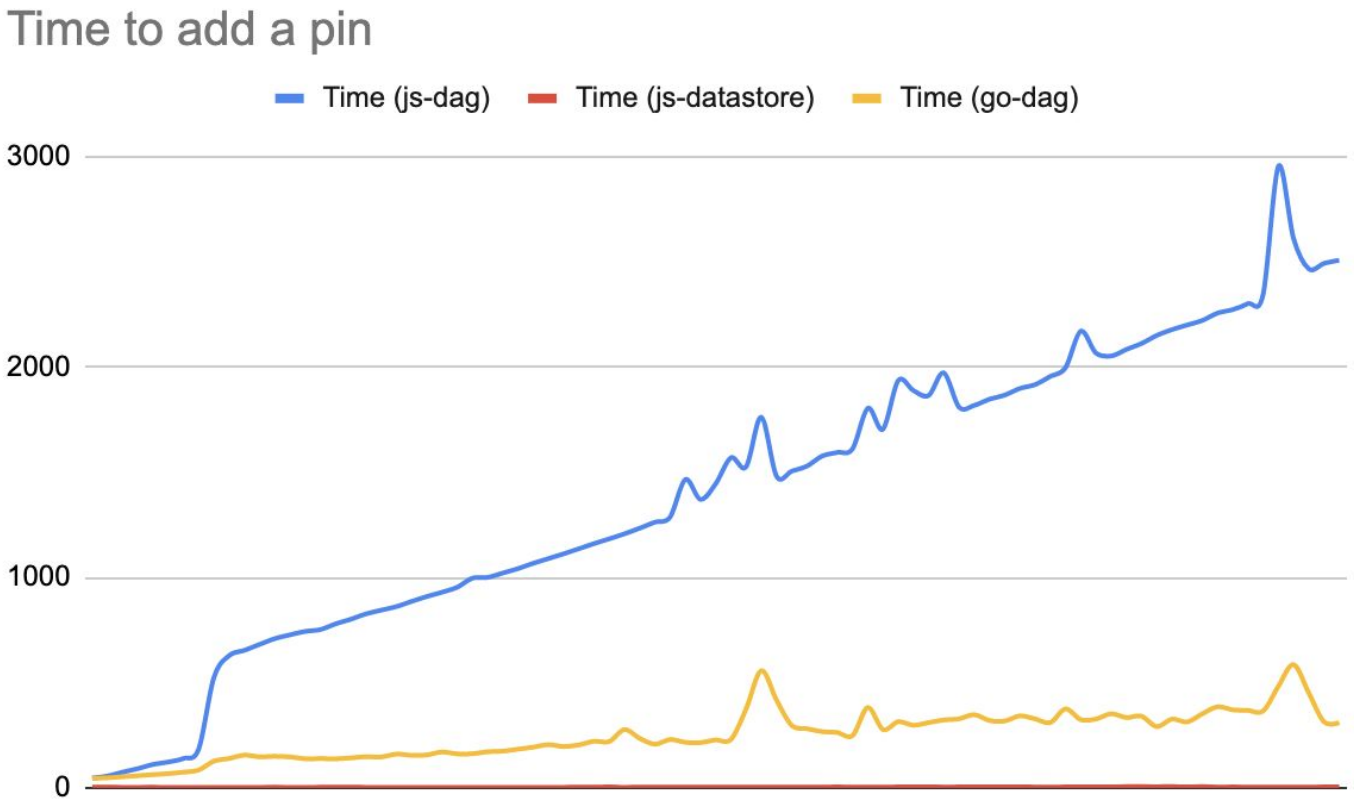
Pinning & garbage collection

Helia

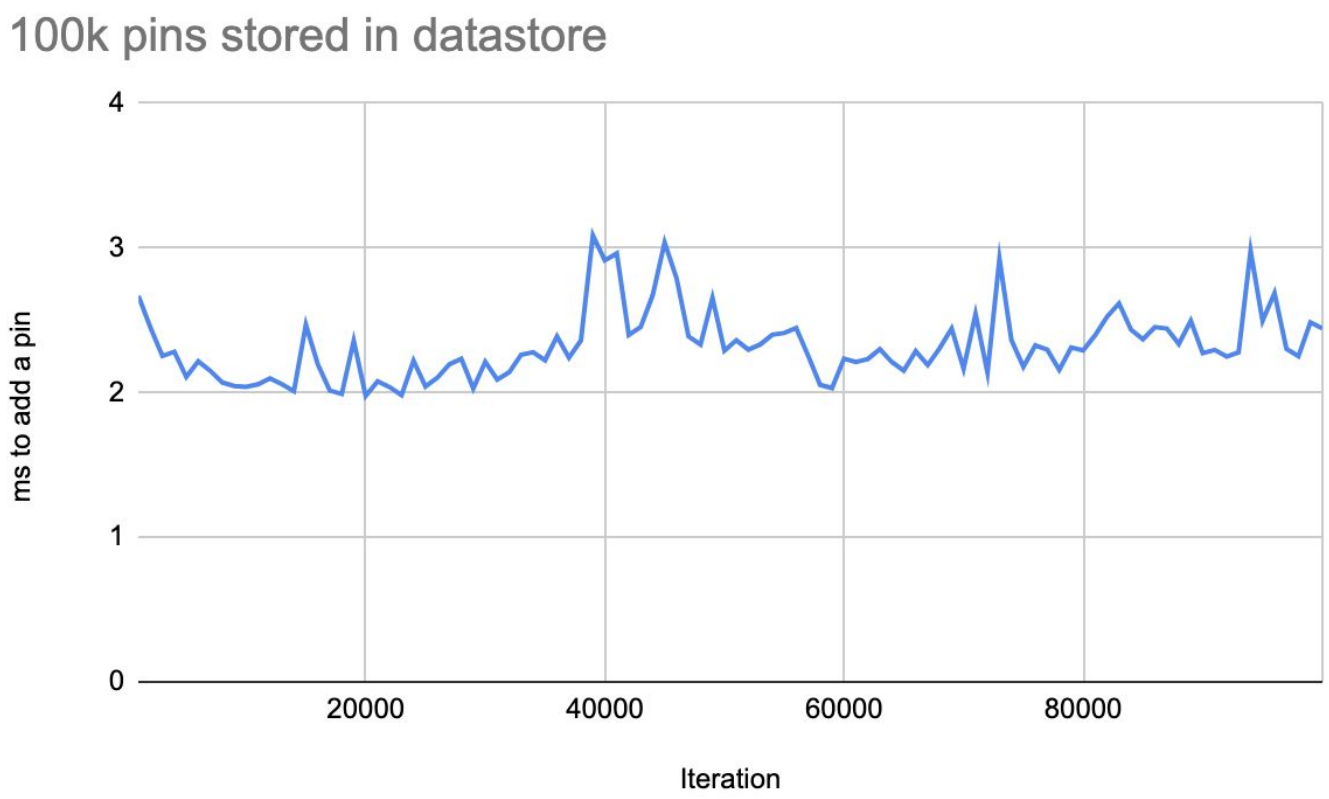
Stored as DAG vs stored in datastore



Helia


Time to add a pin
Time (js-dag)
Time (js-datastore)
Time (go-dag)
3000
2000
1000
0

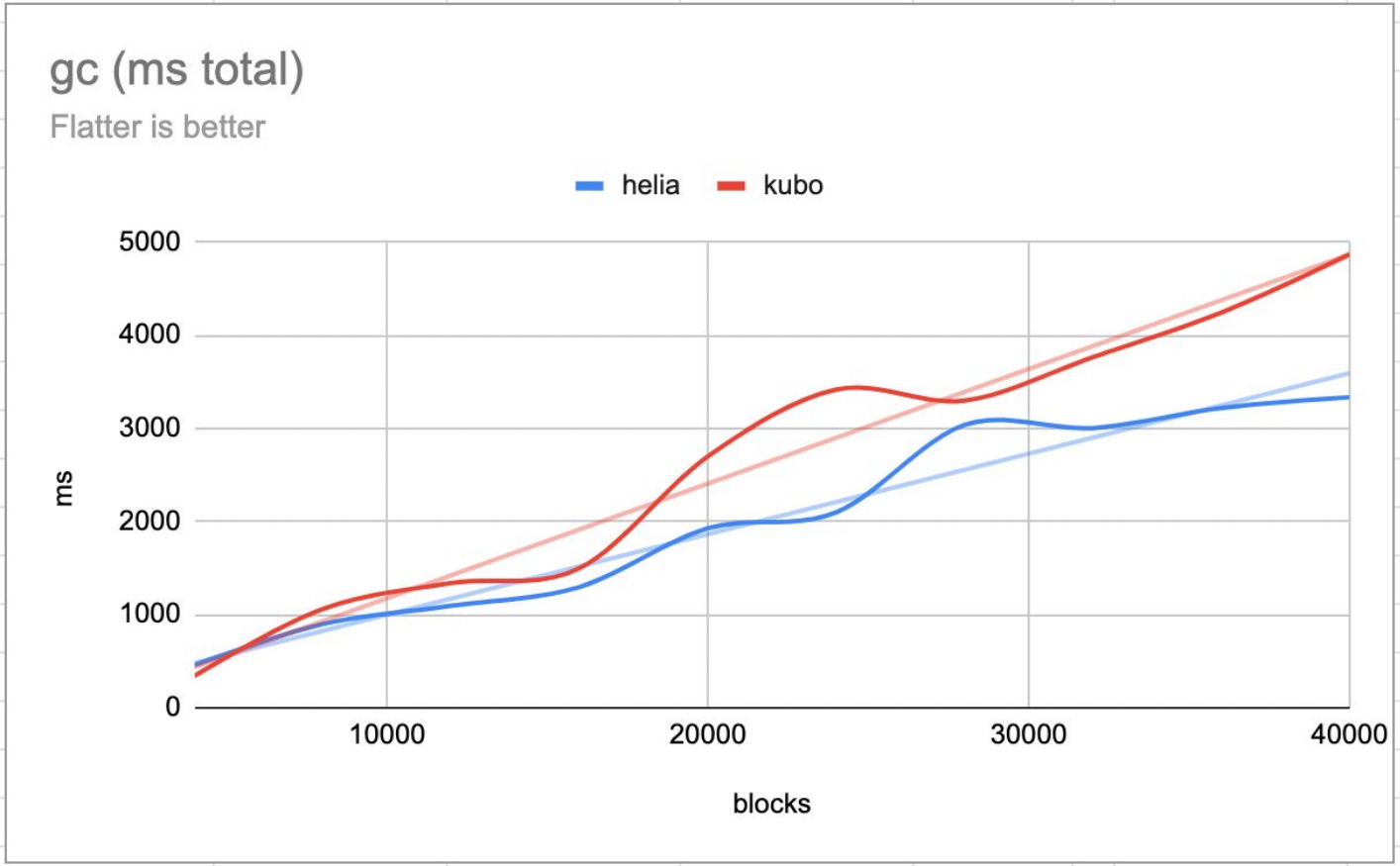
Helia


100k pins stored in datastore
ms to add a pin
4
3
2
1
0
20000
40000
60000
80000
Iteration

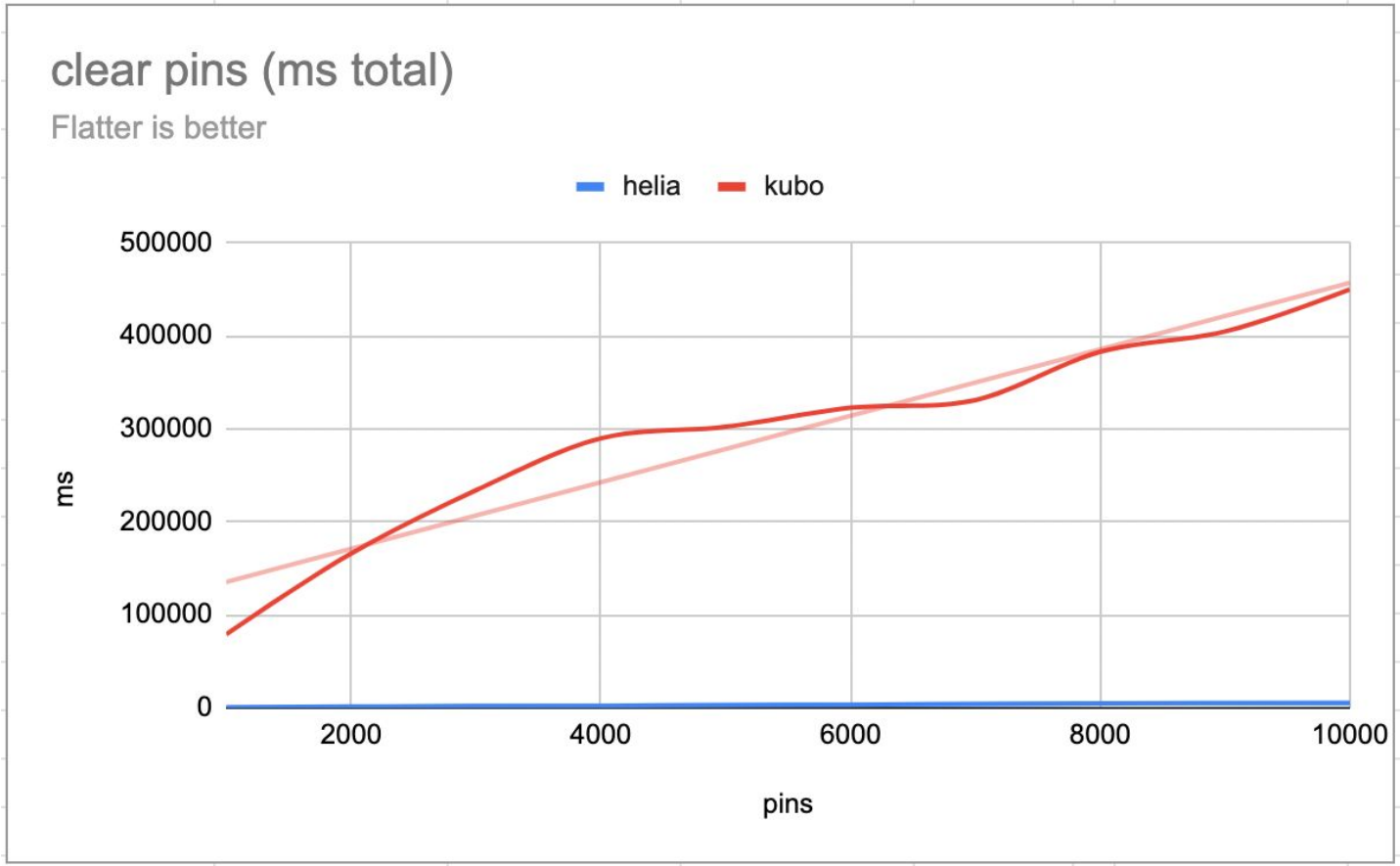
Until now

- **Uses reference counting for pins**
- **https://github.com/ipfs/helia/pull/36**
- **Benchmark suite is in the helia repo:**
  - https://github.com/ipfs/helia/tree/main/benchmarks/gc

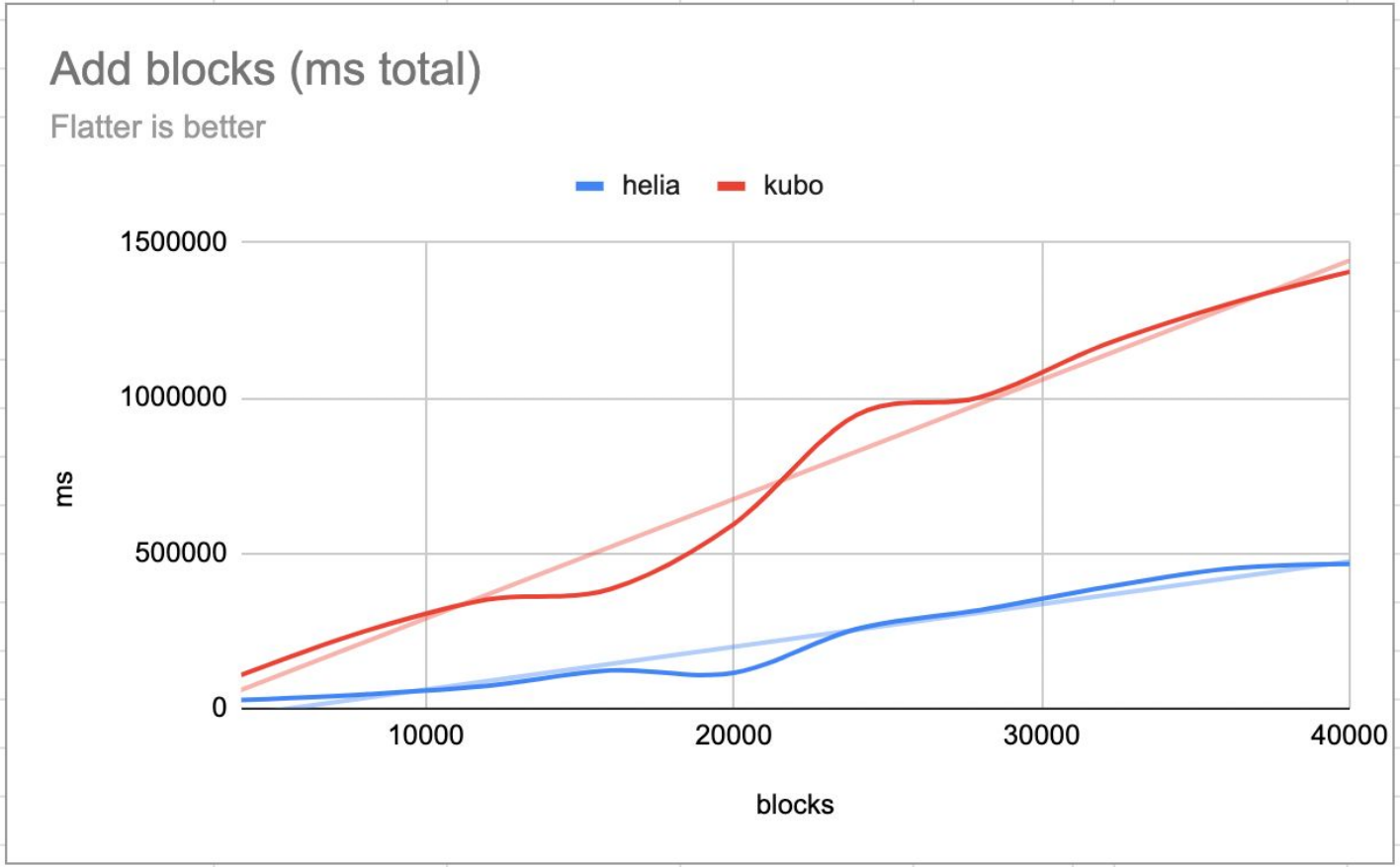
Helia


gc (ms total)
Flatter is better
helia
kubo
5000
4000
3000
2000
1000
0
ms
10000
20000
30000
40000
blocks

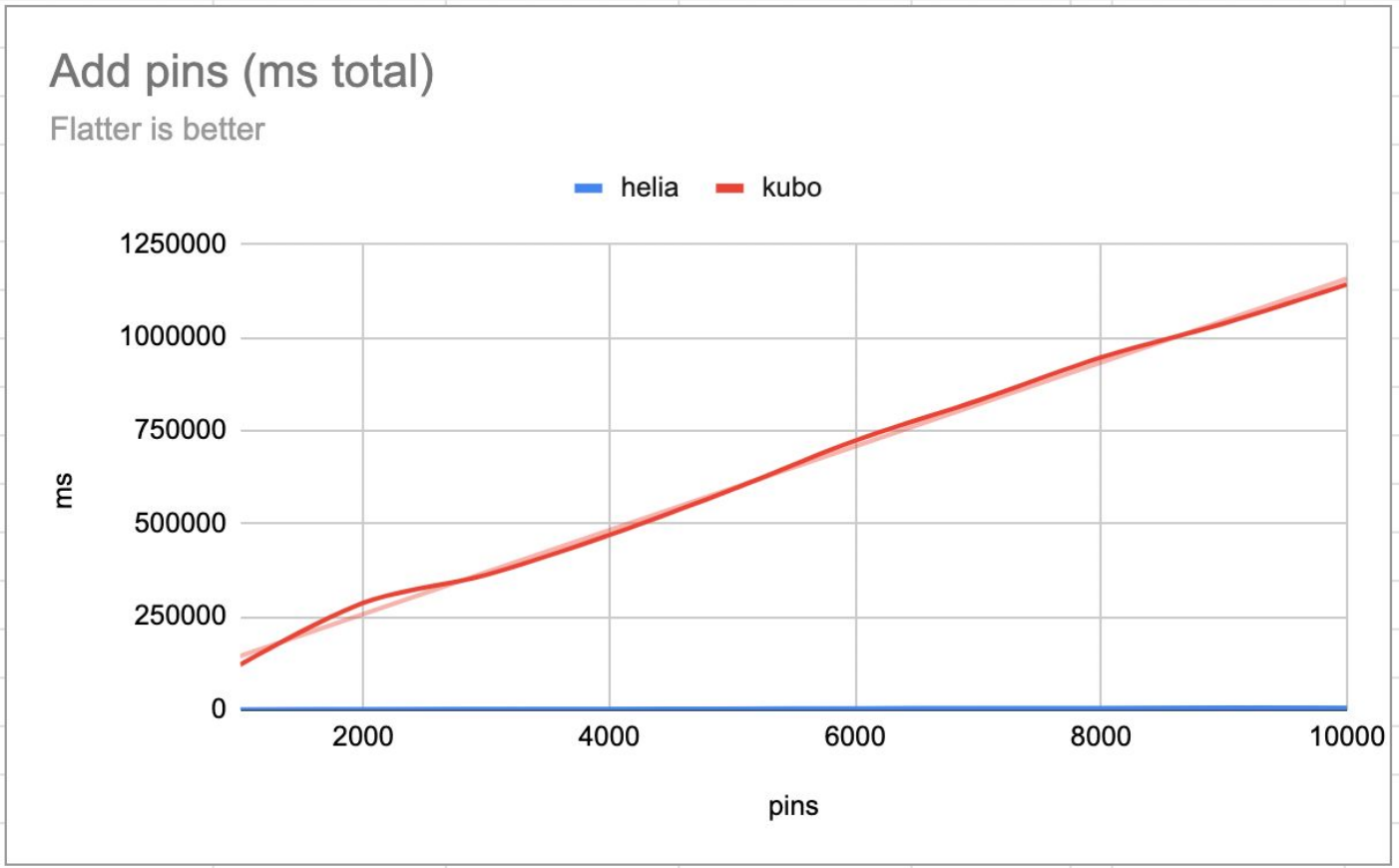
Helia


clear pins (ms total)
Flatter is better
helia
kubo
500000
400000
300000
200000
100000
0
ms
2000
4000
6000
8000
10000
pins

Helia


Add blocks (ms total)
Flatter is better
helia
kubo
1500000
1000000
500000
0
ms
10000
20000
30000
40000
blocks

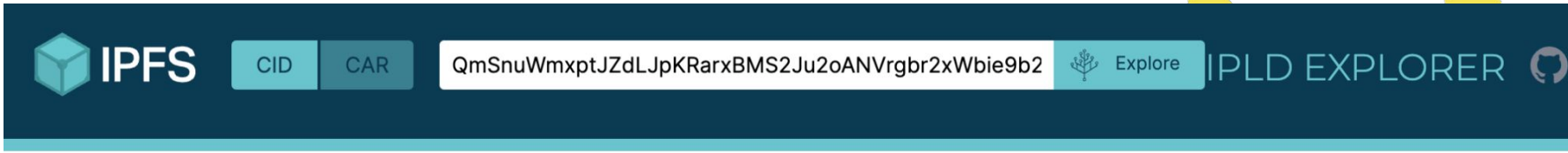
Helia


Add pins (ms total)
Flatter is better
helia
kubo
1250000
1000000
750000
500000
250000
0
ms
2000
4000
6000
8000
10000
pins

DAGs and Bitswap

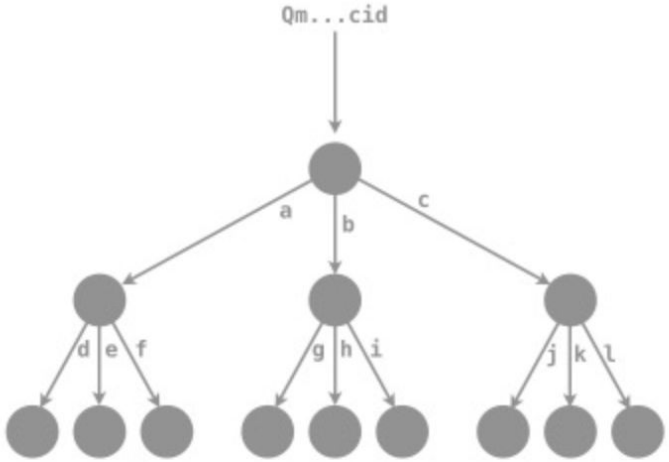
Helia

What is a DAG?



Helia

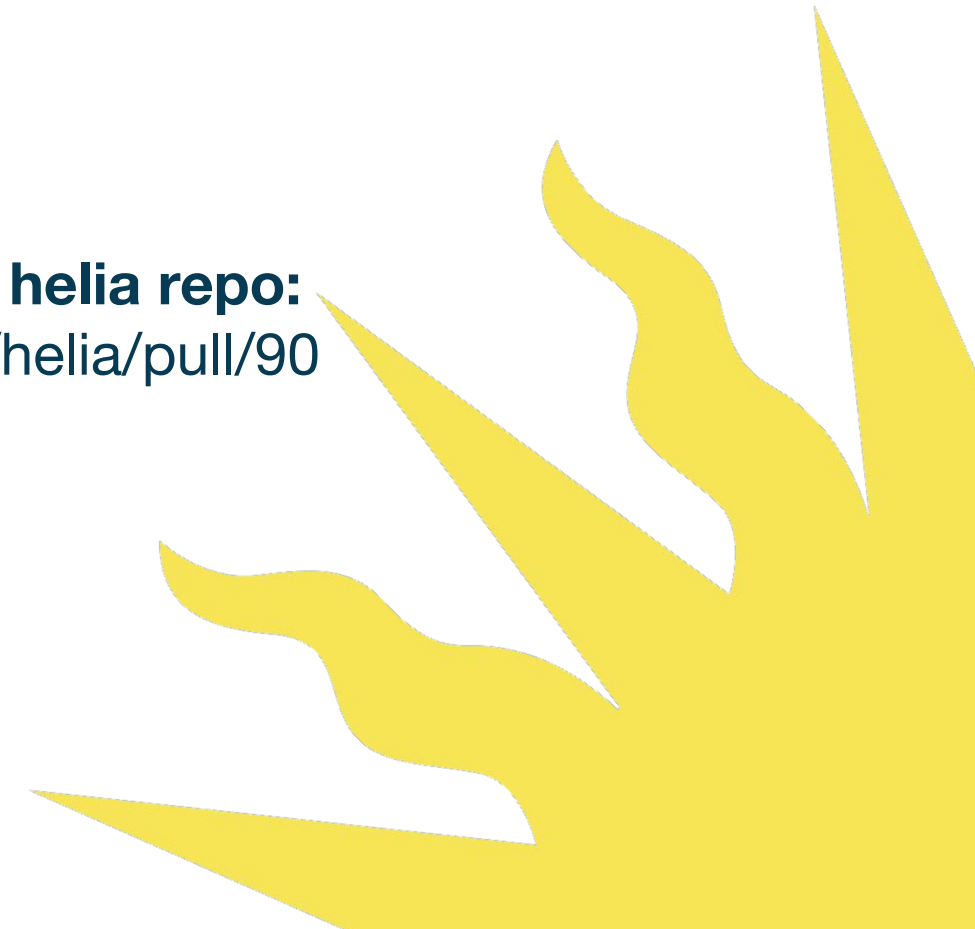
What is a DAG?



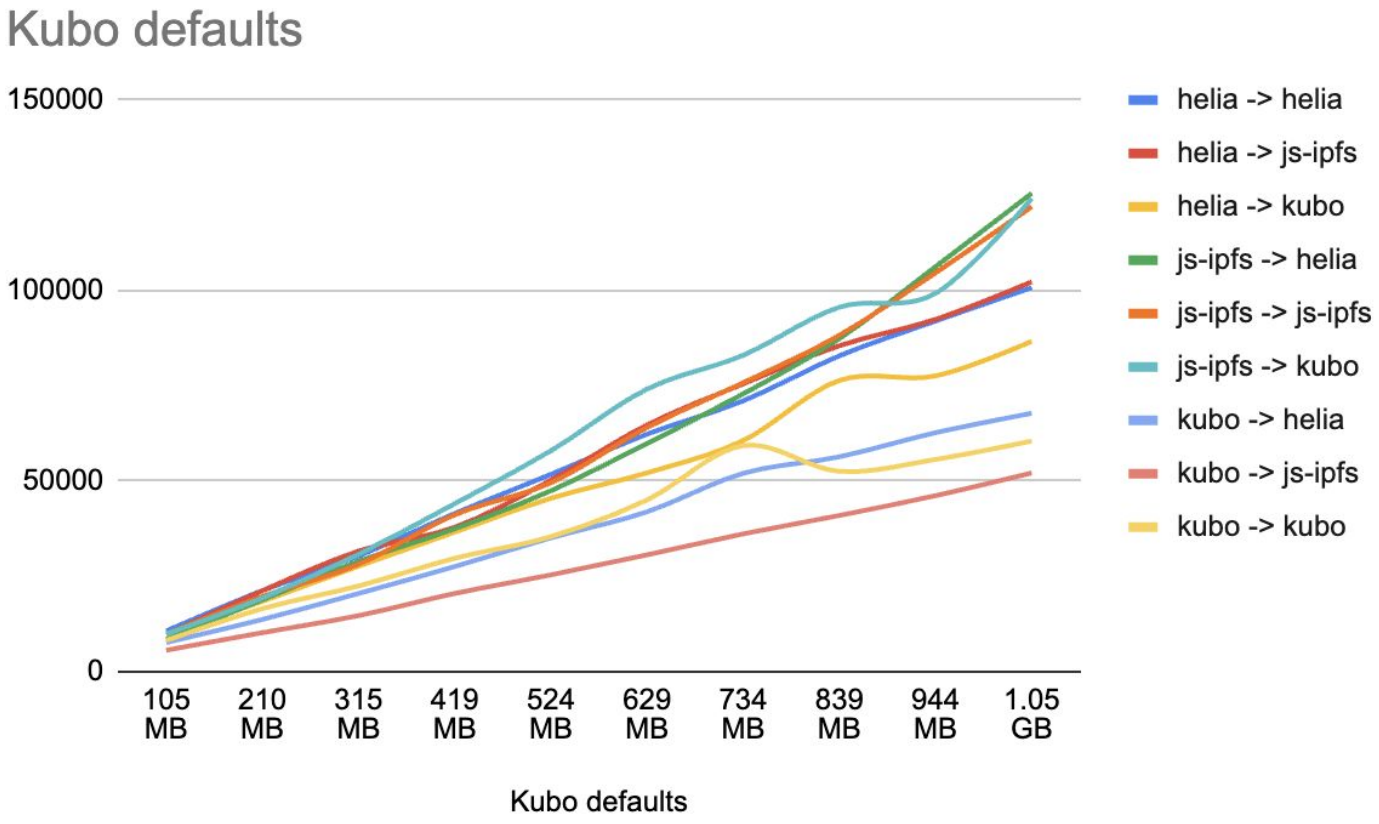
IPLD Dag

…maybe don’t?

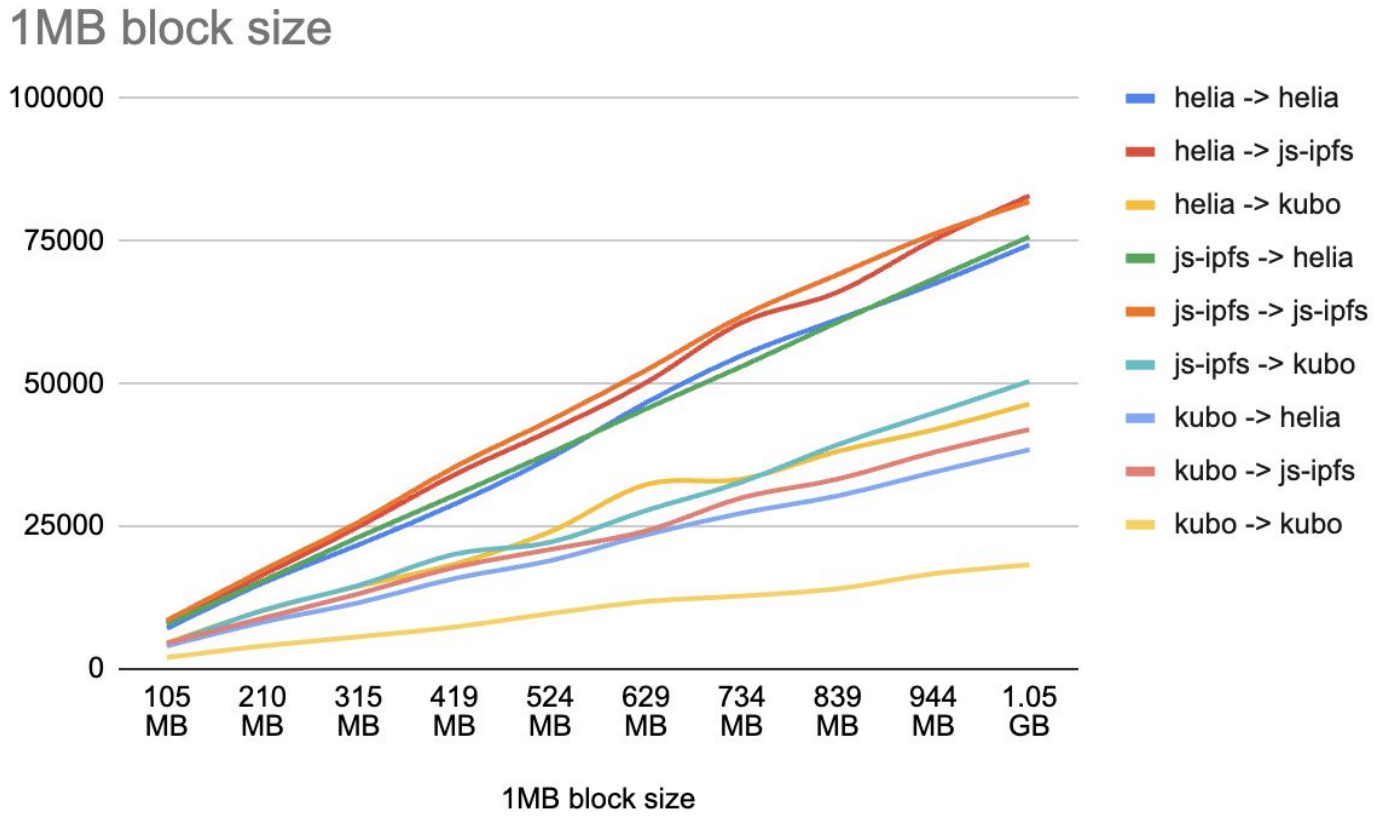
- **Benchmark suite is in the helia repo:**
  - https://github.com/ipfs/helia/pull/90



Helia


Kubo defaults
150000
100000
50000
0
105 MB
210 MB
315 MB
419 MB
524 MB
629 MB
734 MB
839 MB
944 MB
1.05 GB
Kubo defaults
helia -> helia
helia -> js-ipfs
helia -> kubo
js-ipfs -> helia
js-ipfs -> js-ipfs
js-ipfs -> kubo
kubo -> helia
kubo -> js-ipfs
kubo -> kubo

Helia


1MB block size
100000
75000
50000
25000
0
105 MB
210 MB
315 MB
419 MB
524 MB
629 MB
734 MB
839 MB
944 MB
1.05 GB
1MB block size
helia -> helia
helia -> js-ipfs
helia -> kubo
js-ipfs -> helia
js-ipfs -> js-ipfs
js-ipfs -> kubo
kubo -> helia
kubo -> js-ipfs
kubo -> kubo

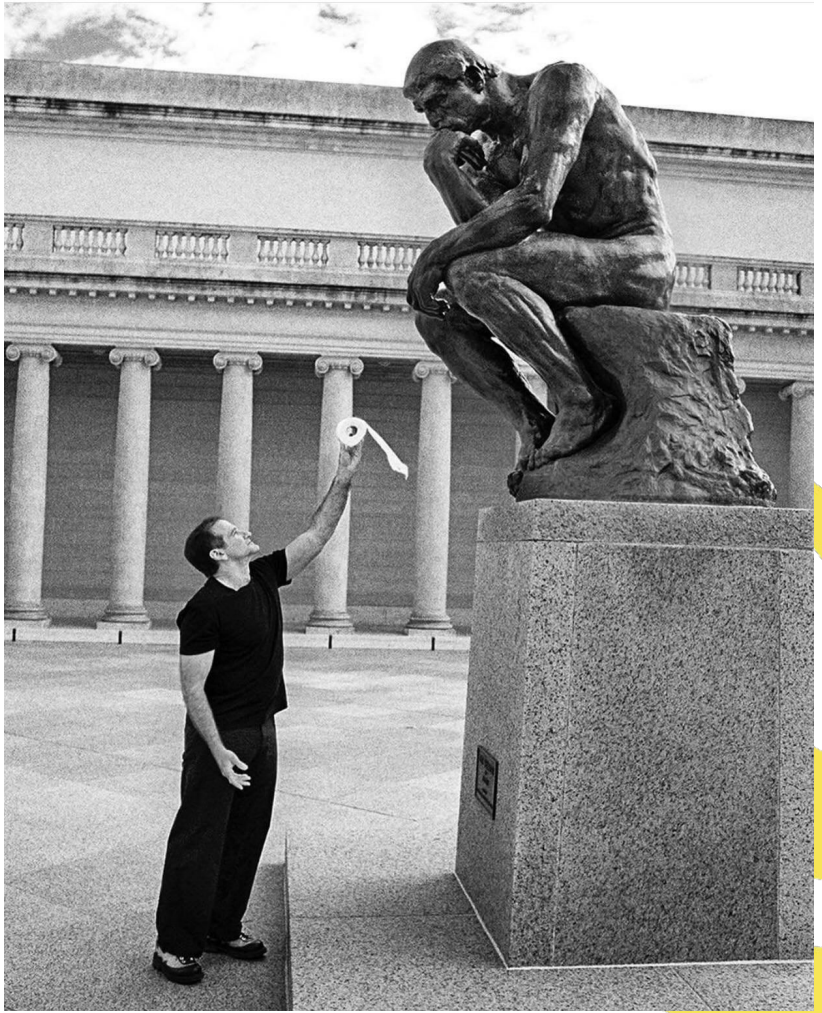
Helia

Helia -> Helia

	256 KiB	**1 MiB**
105 MB	10683ms	7111ms
1.05 GB	100790ms	75353ms

Kubo -> Kubo

	256 KiB	**1 MiB**
105 MB	8184ms	2012ms
1.05 GB	60525ms	18262ms



The end!

- https://github.com/ipfs/helia
- https://github.com/ipfs-examples/helia-examples

Alex Potsides
@achingbrain