Generating Super Magic Hashes

A Parallel Approach

Bhargav Vasist - 11th May 2023

Outline for this presentation

- Recap of Magic Hashes and Generation
- Progress (thus far)
- Results
- Discourse

Magic Hashes

(and Type Juggling)

Magic Hashes

- Specific hashes used to exploit Type Juggling attacks in PHP
- Can be used to detect vulnerabilities in authentication flows

```
php > $user_input = md5(240610708);
php > $test = 0;
php > if ($user_input == $test) { echo "EQ" ;} else { echo "Not EQ" ;}
EQ
```

In PHP two strings matching the regular expression `0+e[0-9]+` compared with `==` returns `true`:

```
'0e1' == '00e2'== '0e1337' == '0'
```

Indeed all these strings are equal to 0 in scientific notation.

Vulnerability Detection

1. Find 2 Magic Hashes to work as passwords
   for e.g. - ‘lowercasegzmgqmx’ and ‘lowercasifdvqkfr’
2. Register for a website with Password 1
3. Attempt to sign in with Password 2
4. If sign-in is successful, the system uses the specified algorithm
   and ‘==’ to compare them.
5. Vulnerability detected

Generating Magic Hashes - SHA1

- SHA1 Digest - 160bits or 40 HEX characters
- Total # of Hashes - 2 ^ 160
- Total # of Rounds per hash - 80
- Each round generates a subset of the digest which is input to the next round
- Ex: - 0e12149120354415335220758399492713921588
- Ex: - d4ee942416a6e4aad41941c1a6a0f92ac097661b

Generating Magic Hashes - SHA1

- Benchmark for 10million hashes generation @ 292ns/op
- Our requirement is to get ~2.4 trillion hashes to get >50%

```
$ go test -bench . -benchmem -benchtime=5s -v
=== RUN   TestParsing
--- PASS: TestParsing (0.00s)
goos: linux
goarch: arm64
BenchmarkParse_1-8      10000000           292 ns/op          280 B/op           7 allocs/op
```

~8 days to have a 50% chance of getting a Magic Hash

Input interpretation

292 ns (nanoseconds) $\times 2.4 \times 10^{12}$

Result

7.008×10^{14} ns (nanoseconds)

Unit conversions

700 800 seconds

194 hours 40 minutes

11 680 minutes

194.667 hours

8.11111 days

Parallelising the Generation - Progress

Goals

Idealised Algorithm

- Use 'N' processors, each generating a password.
- Pairs of processors generate code sections, divided into small and large chunks. They exchange the generated strings for hashing.
- The exchanged strings are hashed, ideally using shared memory for the 80 rounds. OpenMP with shared memory can be used for parallelisation since each round depends on the previous one.
- After generating the magic hash, broadcast it to all processors to conclude the algorithm.

Goals

Idealised Algorithm

- Use 'N' processors, each generating a password.
- ~~Pairs of processors generate code sections, divided into small and large chunks. They exchange the generated strings for hashing.~~
  EXTREMELY INEFFICIENT
- ~~The exchanged strings are hashed, ideally using shared memory for the 80 rounds. OpenMP with shared memory can be used for parallelization since each round depends on the previous one.~~
  IMPOSSIBLE FOR SHA1 ALGORITHM
- After generating the magic hash, broadcast it to all processors to conclude the algorithm.

Proposal for Parallelisation

How do we make it faster?

- Level 1 - Run code on a single processor with ’N’ cores.
- Level 2 - Split Generation and Hashing across ’N’ processors
- Level 3 - Split Generation between pairs of processors - Little/Big Endian style
- Level 4 - All the above across ’N’ processors with each subprocess multithreaded/multicored
- Level 5 - All of the above but now on N nodes/machines

Proposal for Parallelisation

How do we make it faster?

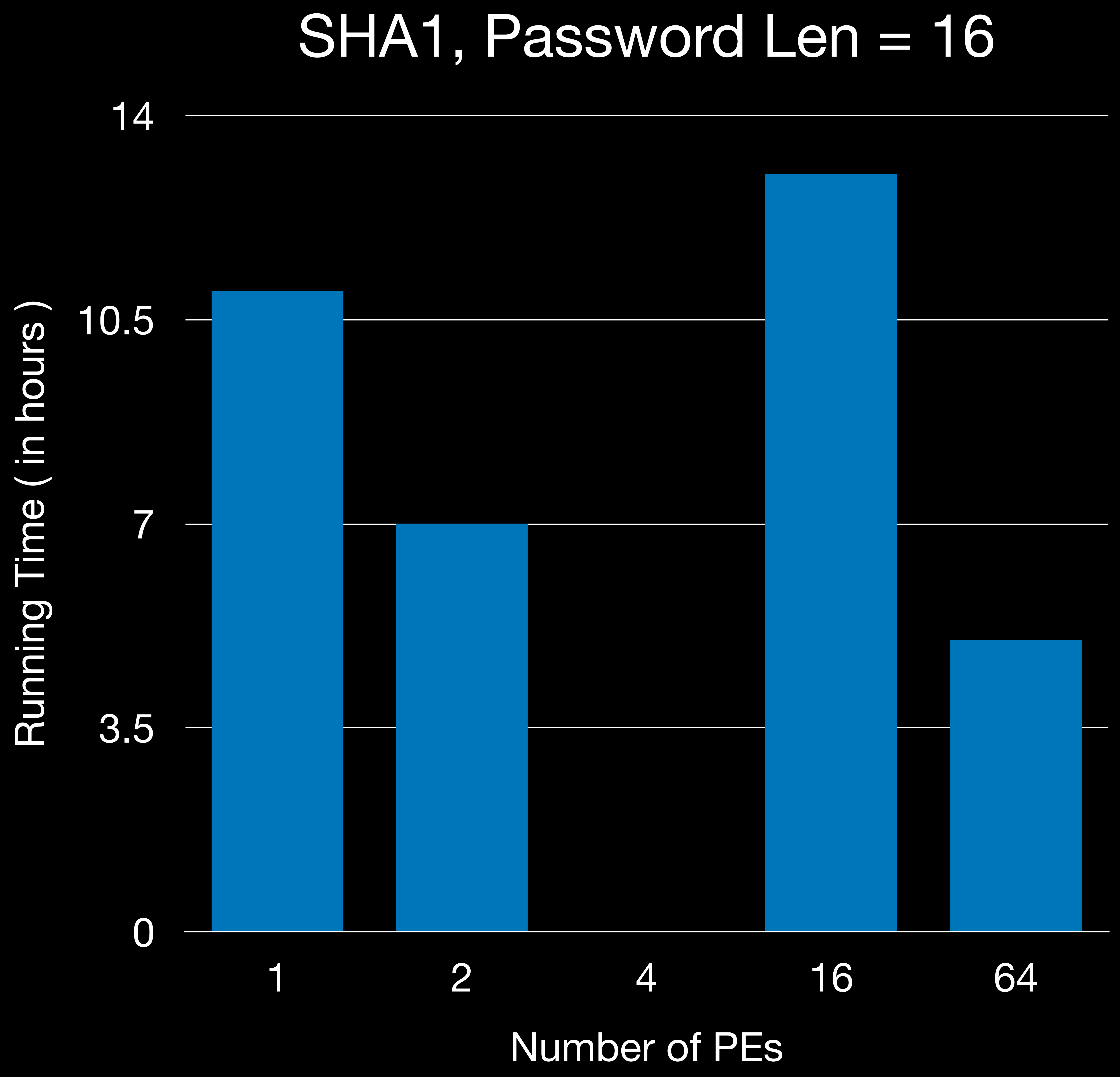
- Level 1 - Run code on a single processor with ’N’ cores. ✅
- ~~Level 2 - Split Generation and Hashing among ’N’ processors~~ ❌
- ~~Level 3 - Split Generation between pairs of processors - Little/Big Endian style~~ ❌
- ~~Level 4 - All the above amongst ’N’ processors with each subprocess multithreaded/multicored~~ ❌
- Level 5 - All of the above but now on N nodes/machines ✅
- Level 6 - Micro-optimise sections of code instead of parallelising ✅

Progress (?)

Randomness Hurdle

The problem with random generation

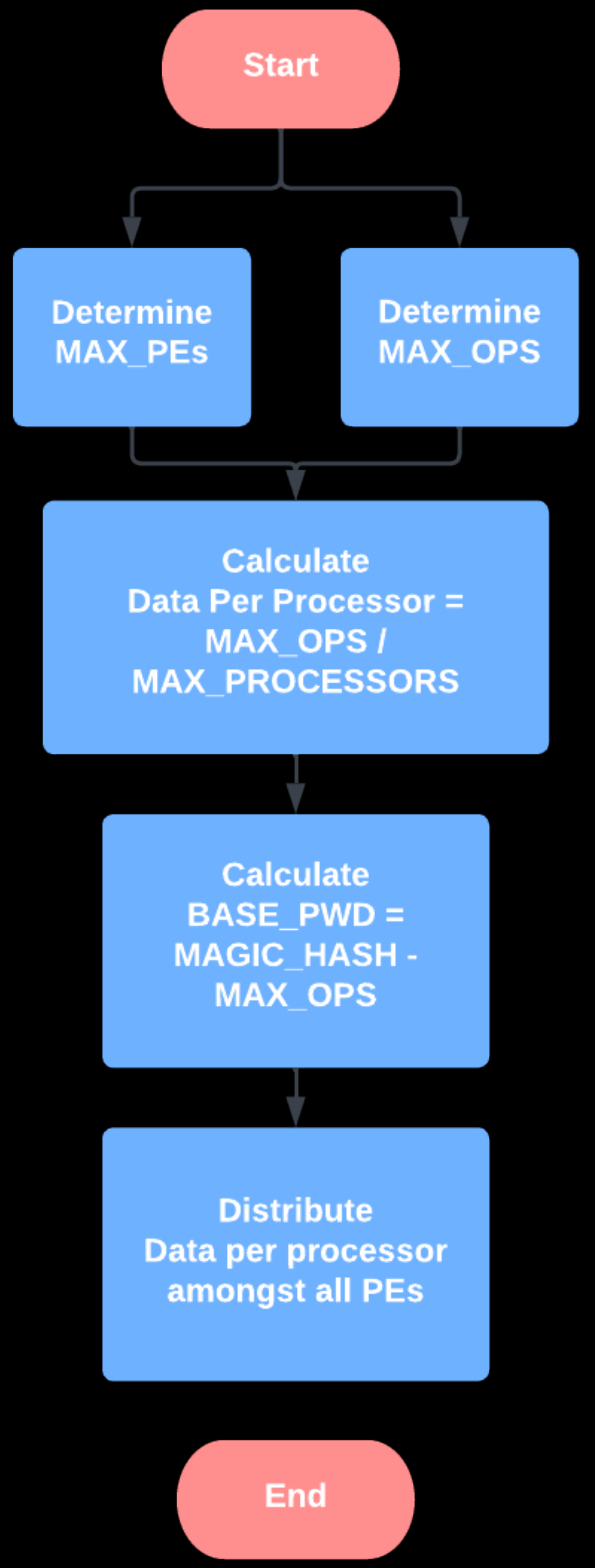
- Randomly generated strings generate random results
- Random results = no distinction between Dependent v/s Control variables
- No distinction = no scientific conclusion
- Also, running time in hours constrains number of experiments



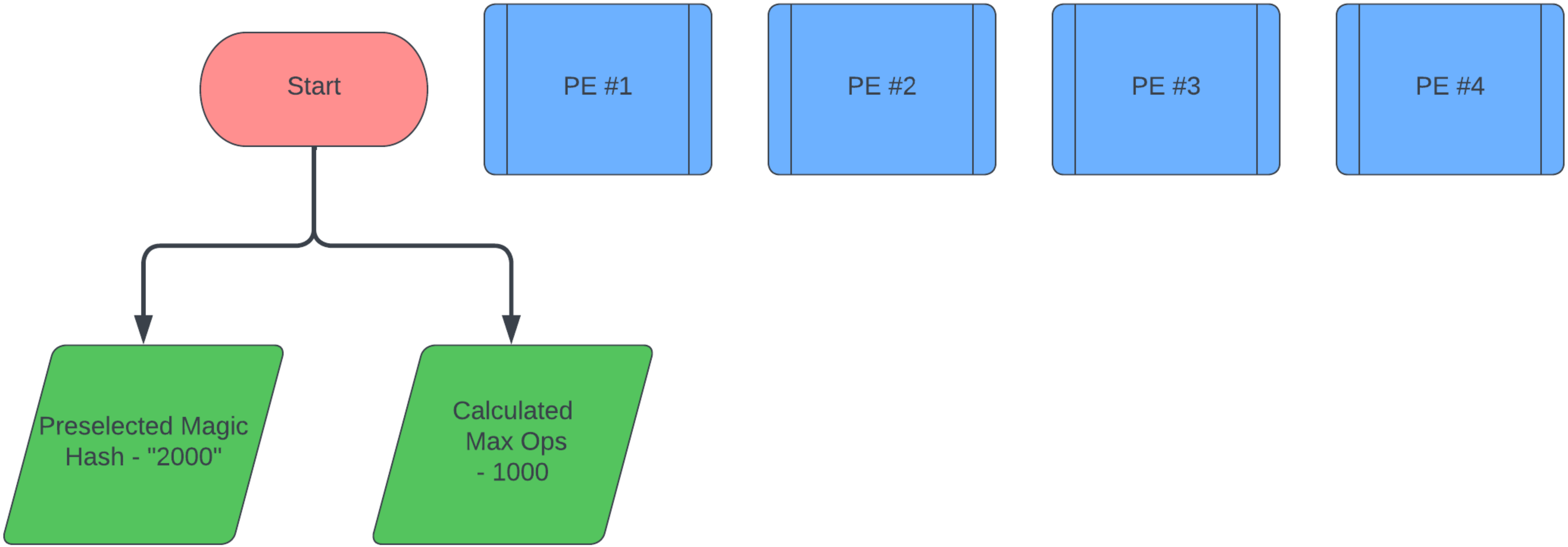
Randomness Hurdle

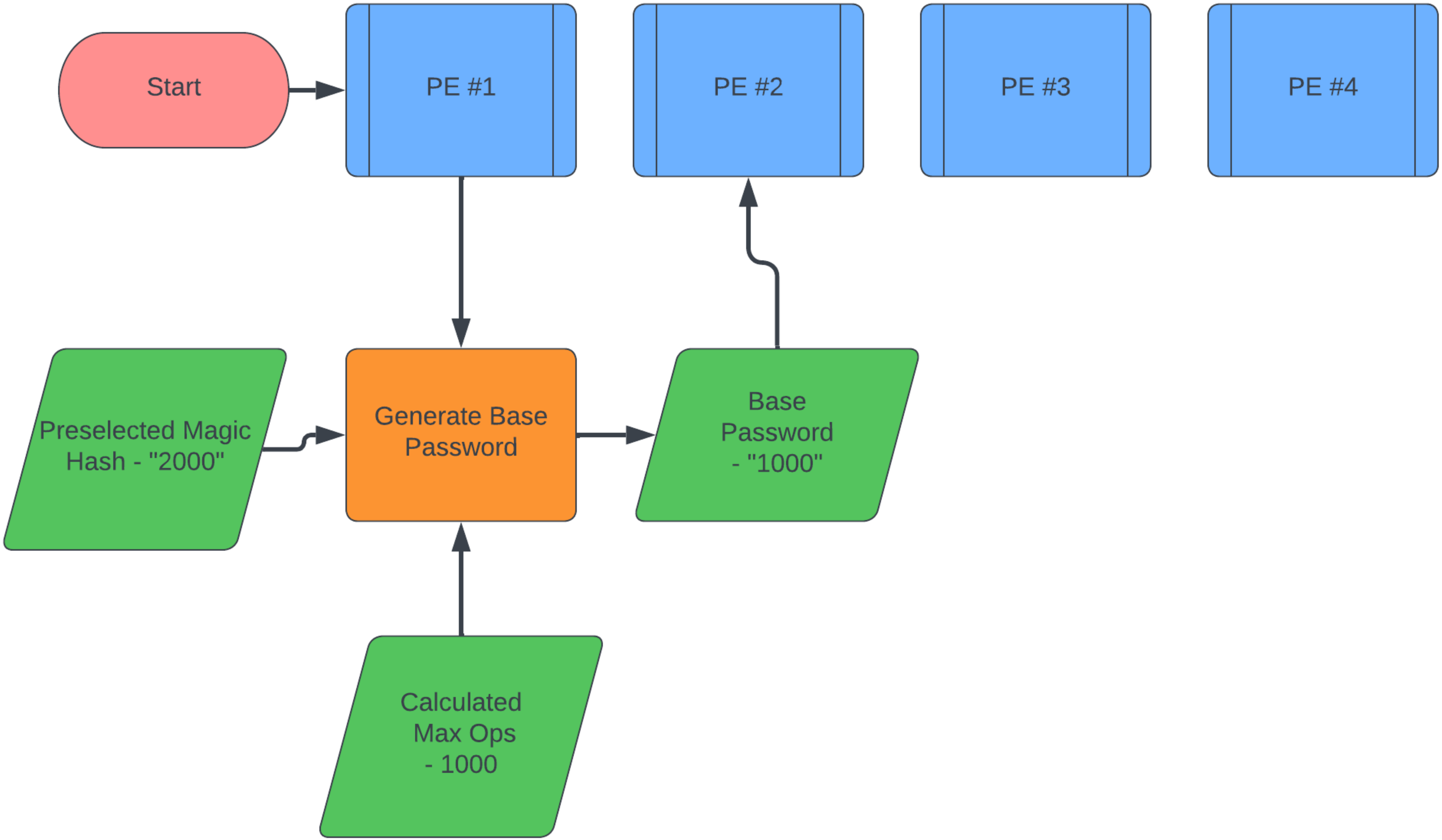
The Solution

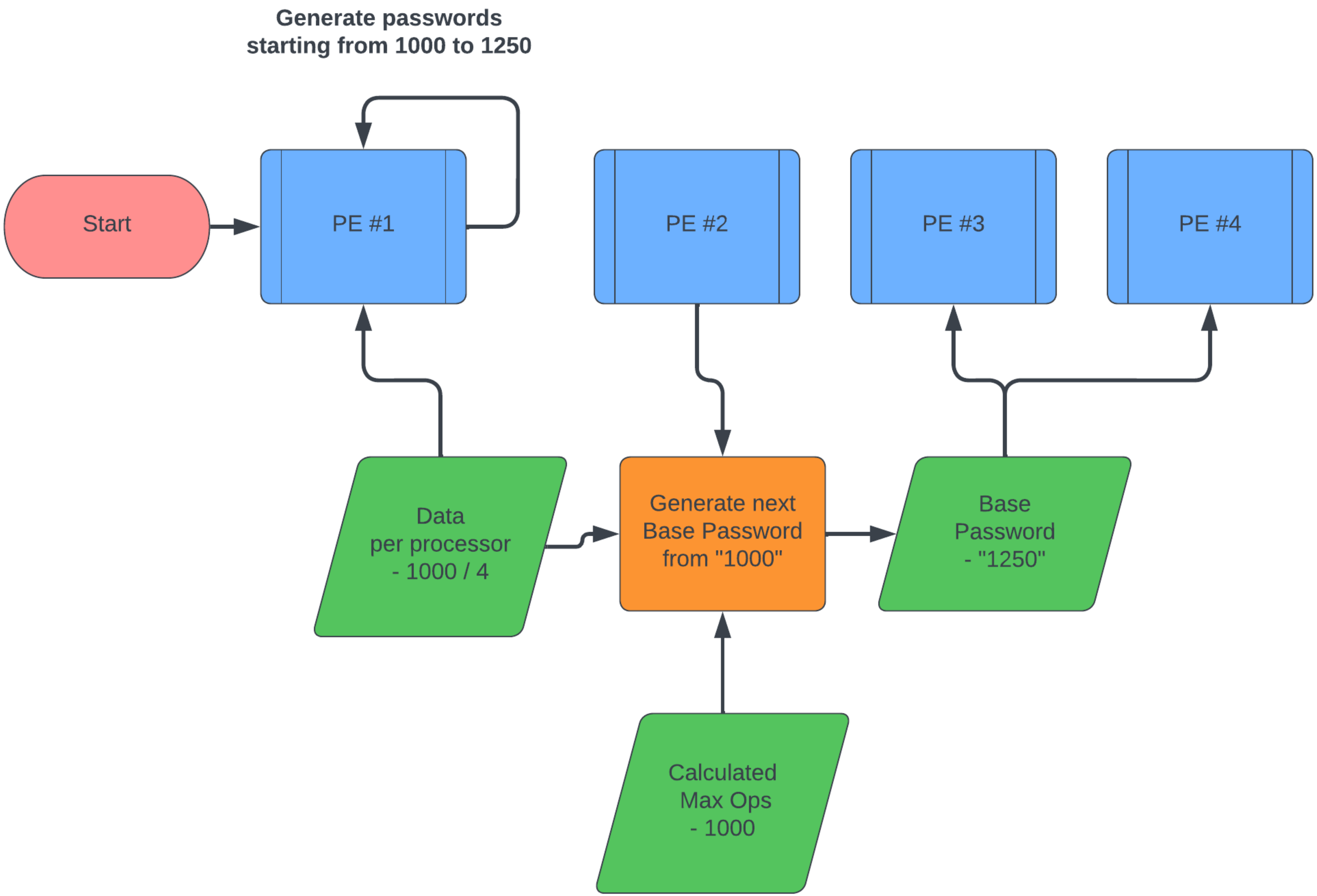
- Use a known Magic Hash
- Generate Upper bounds from this hash
- Uniform comparisons across each PE
- Runtime can be reduced to seconds or minutes based on architecture and hardware capabilities. (We control this)

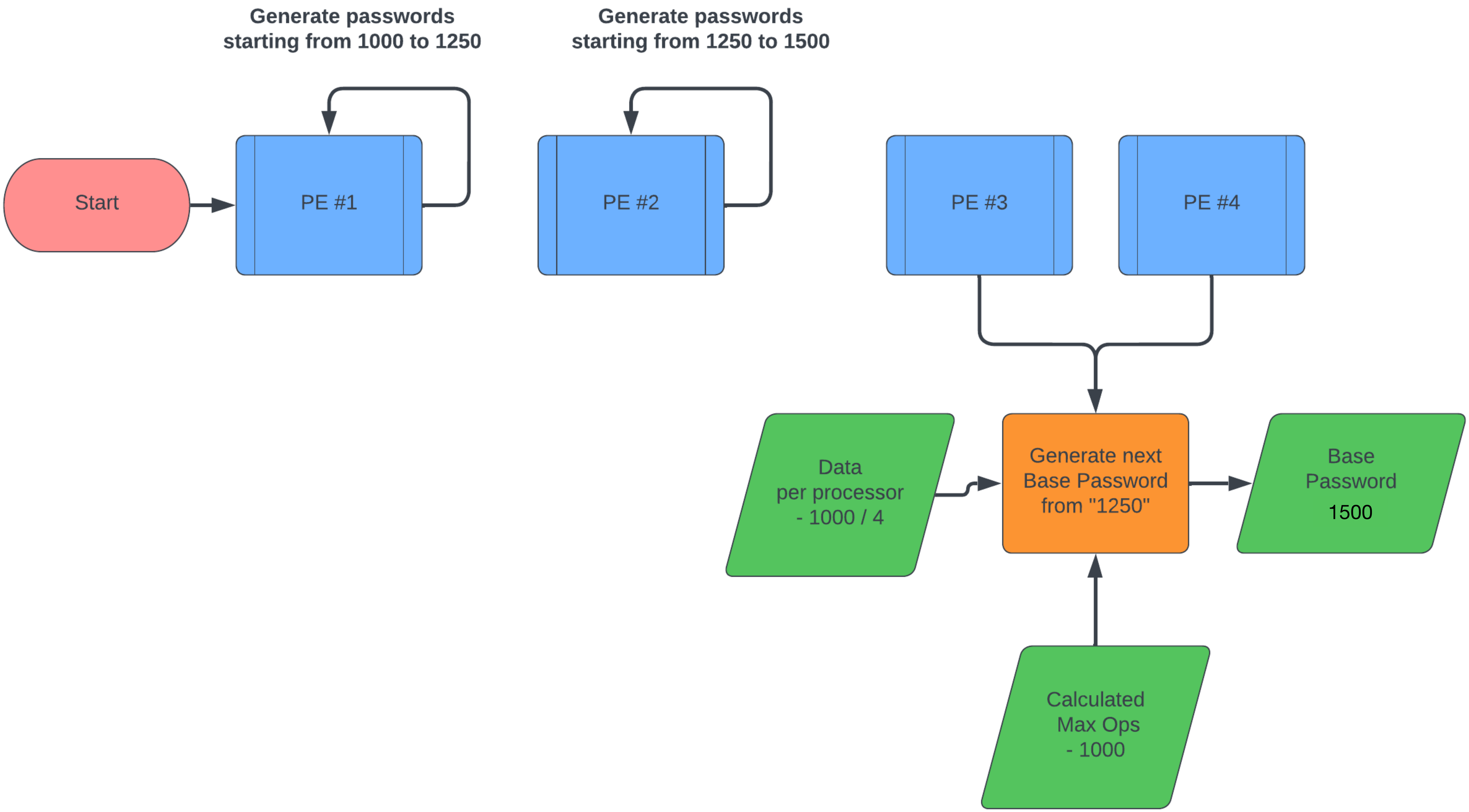


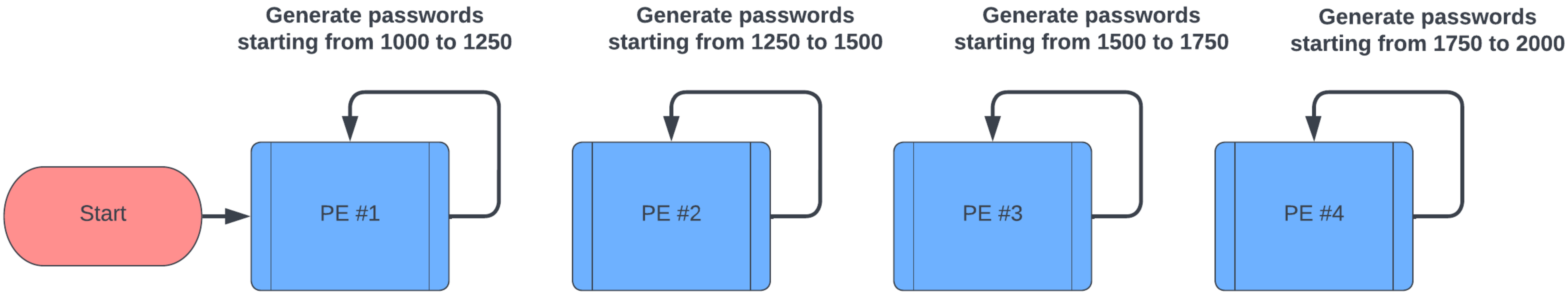
Recursive Doubling Example with 4 PEs and 1000 max ops

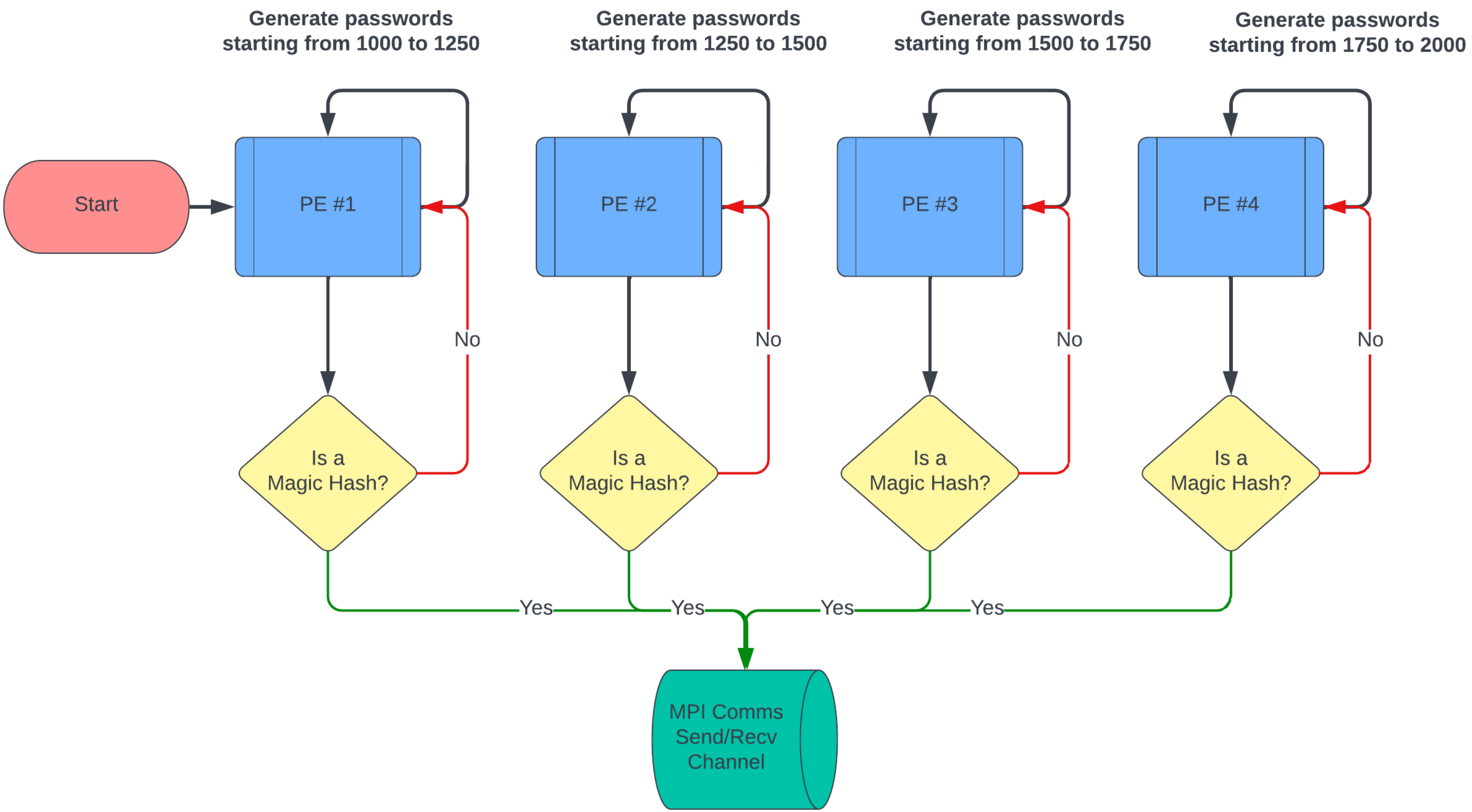


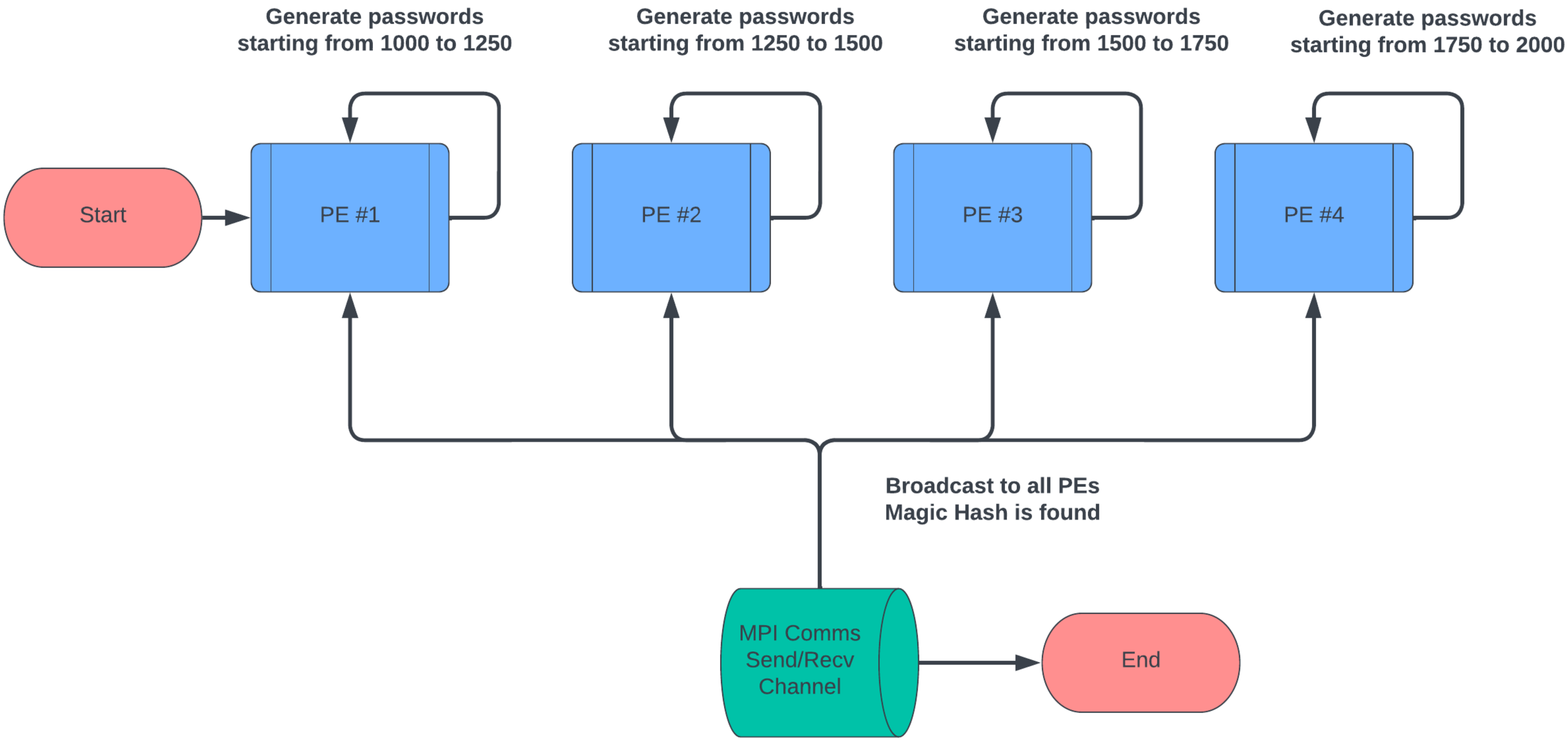

Start
PE #1
PE #2
PE #3
PE #4
Preselected Magic Hash - "2000"
Generate Base Password
Base Password - "1000"
Calculated Max Ops - 1000


Generate passwords
starting from 1000 to 1250
Start
PE #1
PE #2
PE #3
PE #4
Data
per processor
- 1000 / 4
Generate next
Base Password
from "1000"
Base
Password
- "1250"
Calculated
Max Ops
- 1000


Generate passwords starting from 1000 to 1250
Generate passwords starting from 1250 to 1500
Start
PE #1
PE #2
PE #3
PE #4
Data per processor - 1000 / 4
Generate next Base Password from "1250"
Base Password 1500
Calculated Max Ops - 1000


Generate passwords starting from 1000 to 1250
Generate passwords starting from 1250 to 1500
Generate passwords starting from 1500 to 1750
Generate passwords starting from 1750 to 2000
Start
PE #1
PE #2
PE #3
PE #4


Generate passwords starting from 1000 to 1250
Generate passwords starting from 1250 to 1500
Generate passwords starting from 1500 to 1750
Generate passwords starting from 1750 to 2000
Start
PE #1
PE #2
PE #3
PE #4
No
No
No
No
Is a Magic Hash?
Is a Magic Hash?
Is a Magic Hash?
Is a Magic Hash?
Yes
Yes
Yes
Yes
MPI Comms Send/Recv Channel


Generate passwords starting from 1000 to 1250
Generate passwords starting from 1250 to 1500
Generate passwords starting from 1500 to 1750
Generate passwords starting from 1750 to 2000
Start
PE #1
PE #2
PE #3
PE #4
Broadcast to all PEs Magic Hash is found
MPI Comms Send/Recv Channel
End

Overview of Algorithm

1. Initialise MPI Processors
2. Generate Random String
3. Split String and broadcast/recv
4. Generate SHA1 Hash
5. Check for magic hash
6. Continue, within bounds, until found

```
int main(int argc, char *argv[])
{
    // To ensure consistent results - 1 billion ops
    const unsigned long int MAX_OPS = 1_00_000_000;
    int total_proc = init_MPI_comm();
    int curr_proc = init_my_MPI();

    // For each processor, only done once
    const int UPPER_LIMIT_BOUND = MAX_OPS / total_proc;
    std::string pwd = generate_random_password();
    pwd = split_pwd(curr_proc, UPPER_LIMIT_BOUND, pwd);

    // Loop until upper limit is reached
    while (true)
    {
        if (is_magic_hash(pwd))
        {
            notify_all_processors();
            return 1;
        }
        else if (within_upper_bound(pwd, UPPER_LIMIT_BOUND))
        {
            increment_char();
        }
        else
        {
            // Current processor has hit its upper limit
            break
        }
    }
    return 0;
}
```

Overview of Algorithm

Password Splitting

- Every logn processor in MPI_COMM_WORLD generates the next base password
- Broadcast to next logn processors so they can start generating and hashing
- Every other processor in [logn + 1, 2logn] receives base password to compute

```
std::string split_pwd(int curr_proc, int UPPER_LIMIT_BOUND, std::string pwd)
{
    // If processor is power of 2, it generates the next set of base pwd
    if ((curr_proc > 0) && ((curr_proc & (curr_proc - 1)) == 0))
    {

        // Recursively double the generated password
        // Send the generated password to all processors with ranks below
        int bounding_processor = curr_proc * 2 - 1;

        for (int i = curr_proc + 1; i <= bounding_processor; i++)
        {
            MPI_Send(splitPwd, split_pwd.size(), MPI_BYTE, i, 0, MPI_COMM_WORLD)
        }

        int number_of_places_away = curr_proc * UPPER_LIMIT_BOUND;
        std::string splitPwd = incrementStringFor(pwd, number_of_places_away);
    }
    else
    {
        std::string newPwd;
        MPI_Recv(&newPwd, PWD_LEN, MPI_BYTE, nearestPowerOfTwo(curr_proc), 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
}
```

Overview of Algorithm

Character 'Incrementer'

- Increment each character until the boundary of alphanumeric characters and reset.
- Two For Loops means potential for shared memory optimisation

```
std::string incrementStringFor(std::string value, int count)
{
    std::string newStr = value;
    for  int i = 0; i < count; i++)
    {  (
        int carry = 1;
        std::string res = "";
        for  int j = value.length() - 1; j >= 0; j--)
        {  (
            char c = value[j];
            int charCode = static_cast<int>(c);
            charCode += carry;
            if  charCode > 127)
            { (
                charCode = 0;
                carry = 1;
            }
            else
            {
                carry = 0;
            }
            res = static_cast<char>(charCode) + res;
            if  !carry)
            { (
                res = value.substr(0, j) + res;
                break;
            }
        }
        if  carry)
        { (
            res = '\0' + res;
        }
        newStr = res;
        value = newStr;
    }
    return newStr;
}
```

Overview of Algorithm

Magic Hash Checker

- Generates and checks hash
- Magic Hashes only need to start with any amount of 0s and one count of ‘e’
- Fastest way to match string patterns is regex

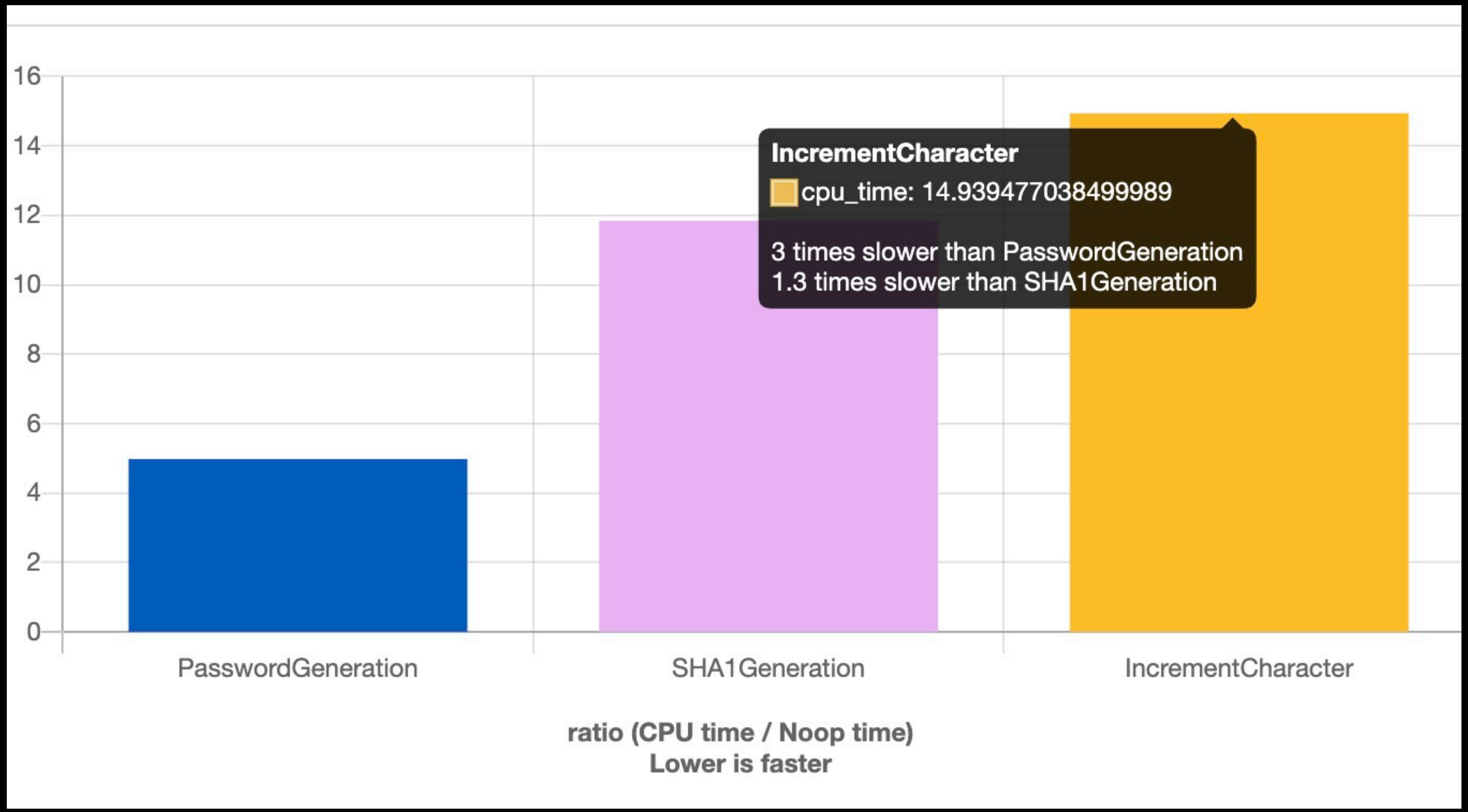
```
bool is_magic_hash(std::string pwd)
{
    std::string hash = generate_hash(pwd);
    std::regex const regExp{"~^0+e\\d*$~"};
    std::smatch matched_arr;
    if  std::regex_match(hash, matched_arr, regExp))
    { (
        return true
    }
    else
    {
        return false
    }
}
```

Where are my cycles?

- The 3 Big Functions for our flow

1. Password Generator
2. SHA1 Hash Generator
3. Increment Character

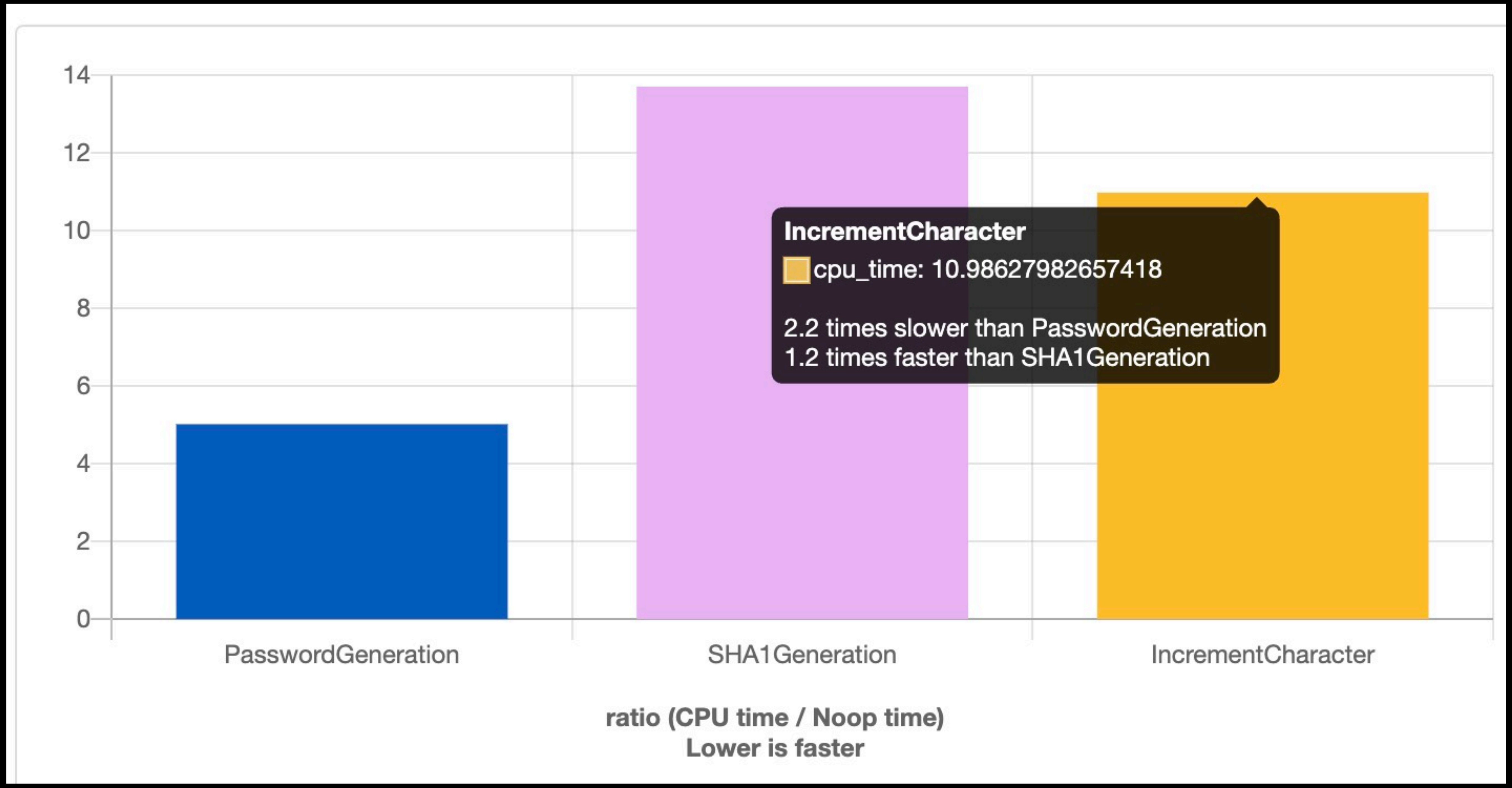
- Password Generation happens only once per node/PE



Micro-Optimisation #1

Password Generation

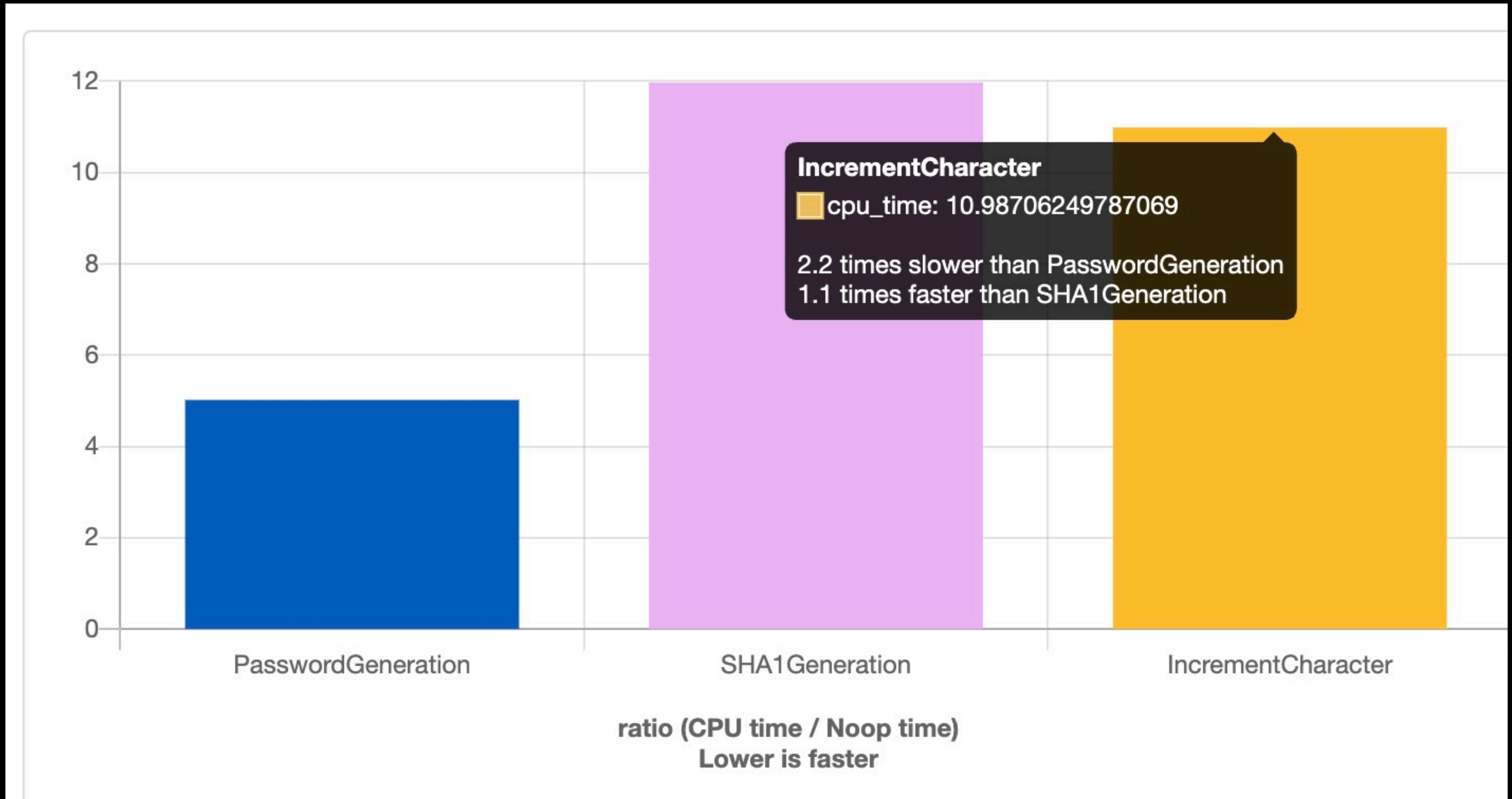
- Use array of bytes instead of strings for passwords
- Requires less allocations if not using a heavy class like std::string
- Less allocations = less wasted cycles = faster runtimes



Micro-Optimisation # 2

SHA1 Hash Generation

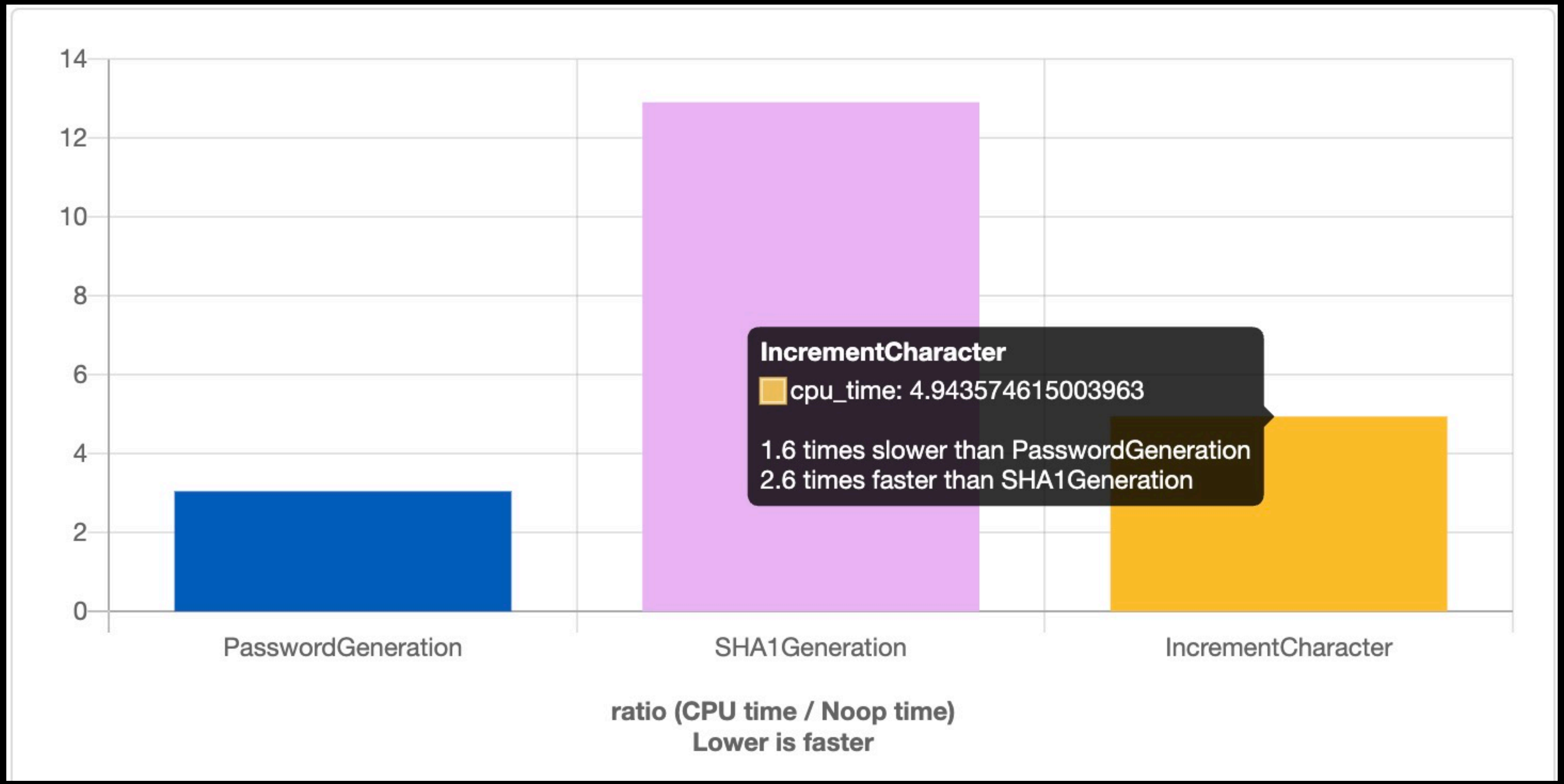
- Use Intel’s SHA function that is already baked into most modern CPUs
- Uses CPU Instructions to compute hashes efficiently
- (Don’t reinvent the wheel)
- Doesn’t work for SHA224 (Couldn't find one)



Micro-Optimisation # 3

String <-> Byte conversion

- Replace string comparisons everywhere with byte comparisons
- No more string conversions or manipulations means CPU registers are better utilised
- Huge gains!



Micro-Optimisation # 4

OpenMP Shared Memory

- Use OpenMP for Password Generation
- Pragma OMP Reduction clause works very well with FOR loops

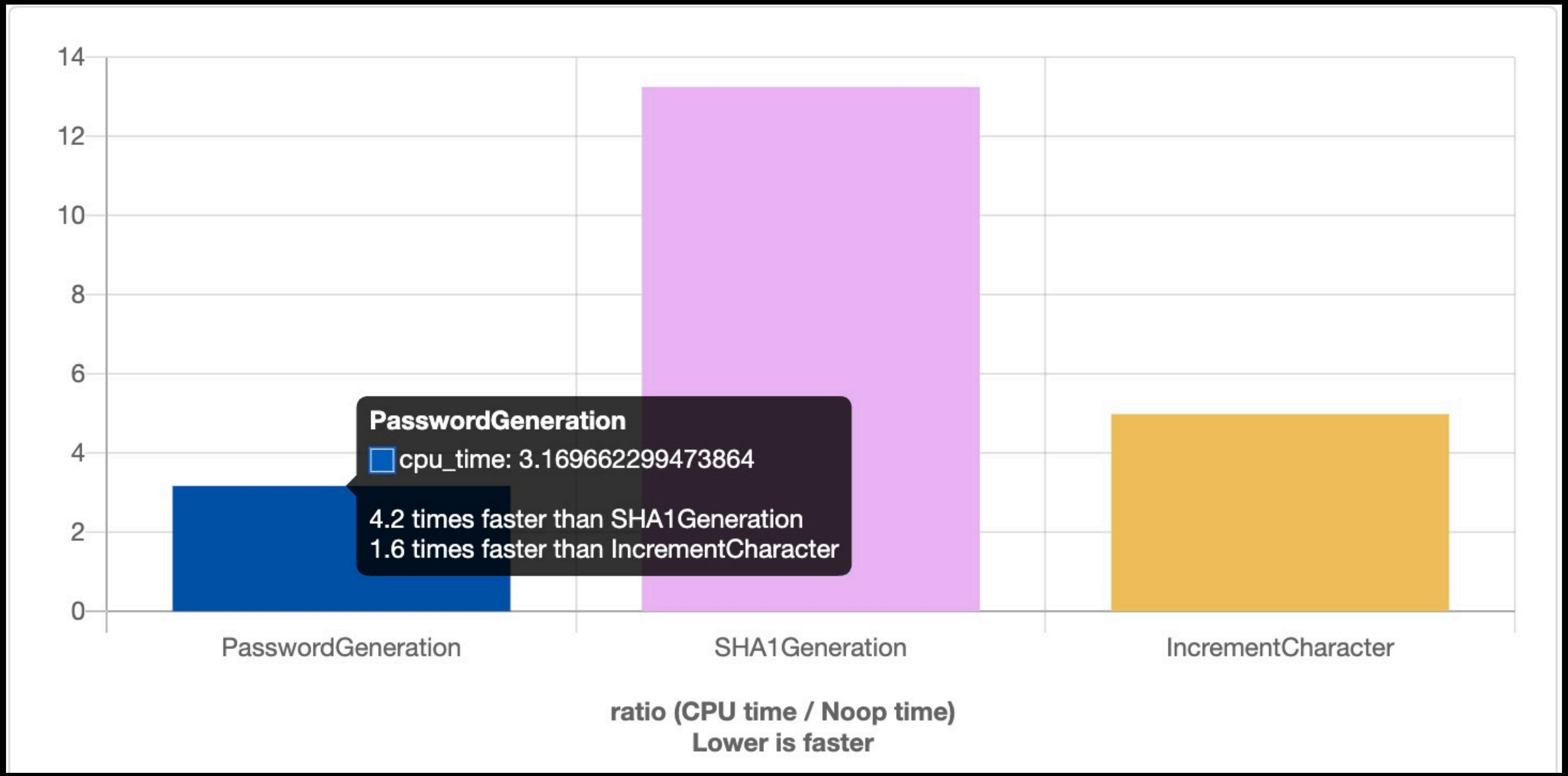
```
std::string incrementStringFor(std::string value, int count)
{
    std::string newStr = value;
    for (int i = 0; i < count; i++)
    {
        int carry = 1;
        std::string res = "";

        #pragma omp parallel for reduction(+ : res)
        for (int j = value.length() - 1; j >= 0; j--)
        {
            char c = value[j];
            int charCode = static_cast<int>(c);
            charCode += carry;
            if (charCode > 127)
            {
                charCode = 0;
                carry = 1;
            }
            else
            {
                carry = 0;
            }
            res = static_cast<char>(charCode) + res;
            if (!carry)
            {
                res = value.substr(0, j) + res;
                break;
            }
        }
        if (carry)
        {
            res = '\0' + res;
        }
        newStr = res;
        value = newStr;
    }
    return newStr;
}
```

Micro-Optimisation # 4

OpenMP Shared Memory

- Use OpenMP for Password Generation
- Pragma OMP Reduction clause works very well with FOR loops
- Tiny gains but gains nonetheless



Results

Timeline

Previous progress

- Run a sequential version of the final algorithm on device
  ~11 hours
- Run the sequential version on cluster (without any optimisation)
  ~13 hours
- Split the string generation and hash verifier pieces of code (no other forms of communication between processors)
  ~9 hours
- Split the string generation into chunks (communication between pairs of processors)
  ~16 hours

Timeline

Current Results

- Run a sequential version of the final algorithm on device
  [30 - 300] seconds

- ~~Run the sequential version on cluster (without any optimisation)~~
  ~~~13 hours~~

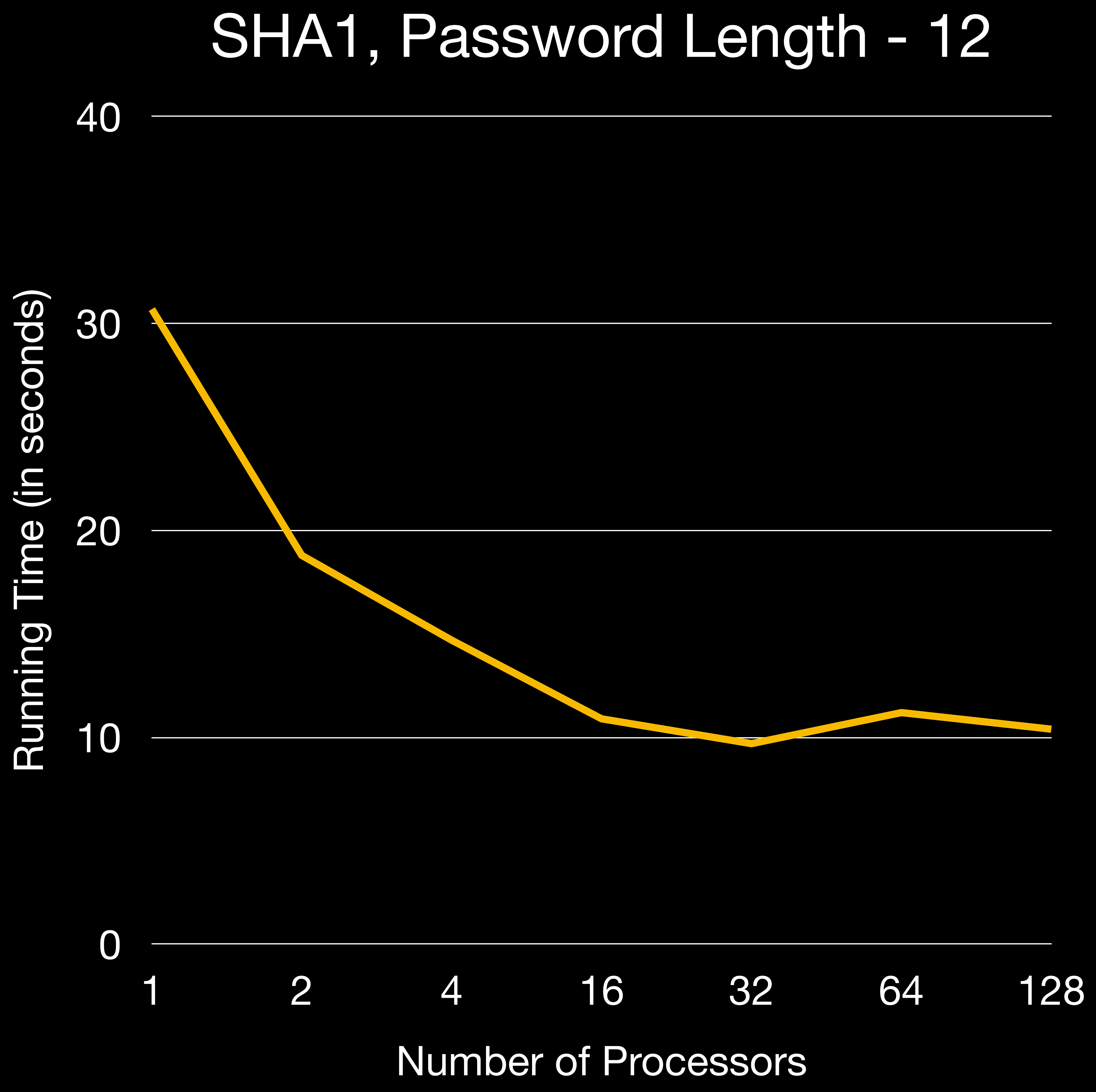
- ~~Split the string generation and hash verifier pieces of code (no other forms of communication between processors)~~
  ~~~9 hours~~

- ~~Split the string generation into chunks (communication between pairs of processors)~~
  ~~~16 hours~~

- Run the final algorithm on clusters
  [10 - 150] seconds
  ~~~

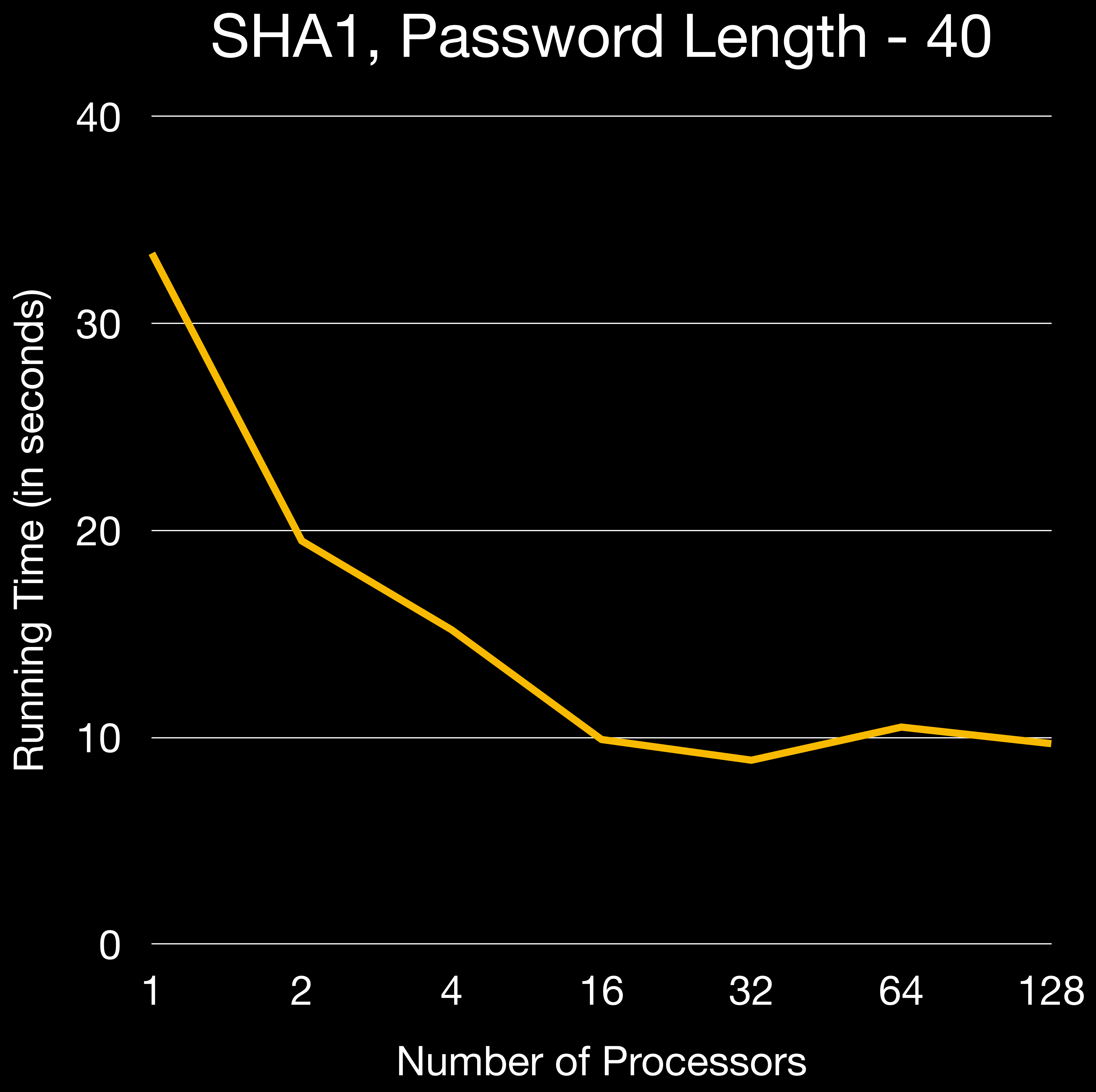
Results - Running Time

PEs	Time (secs)
1	30.7
2	18.8
4	14.7
16	10.9
32	9.7
64	11.2
128	10.4



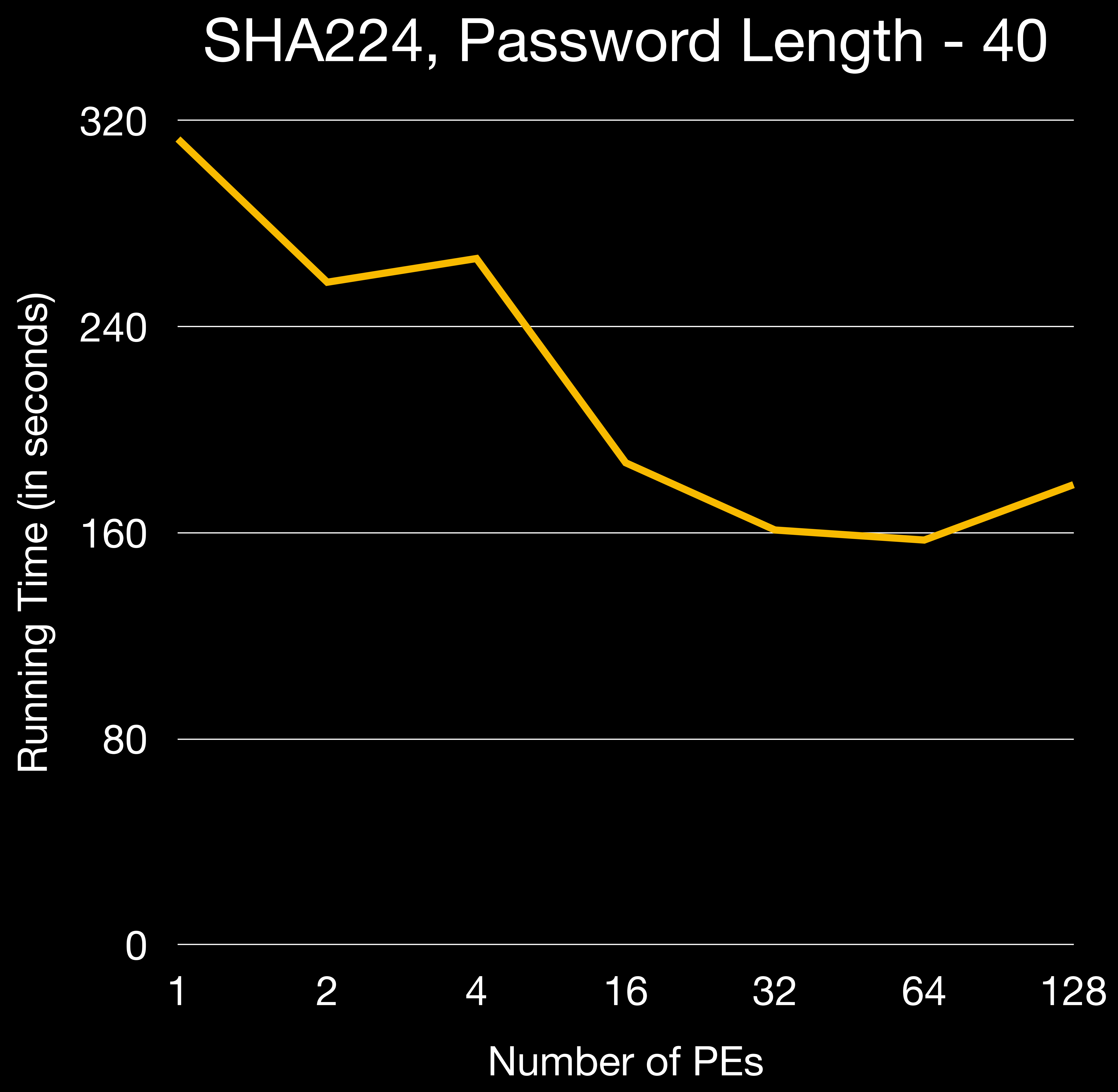
Results - Running Time

PEs	Time (secs)
1	33.4
2	19.5
4	15.2
16	9.9
32	8.9
64	10.5
128	10.2



Results - Running Time

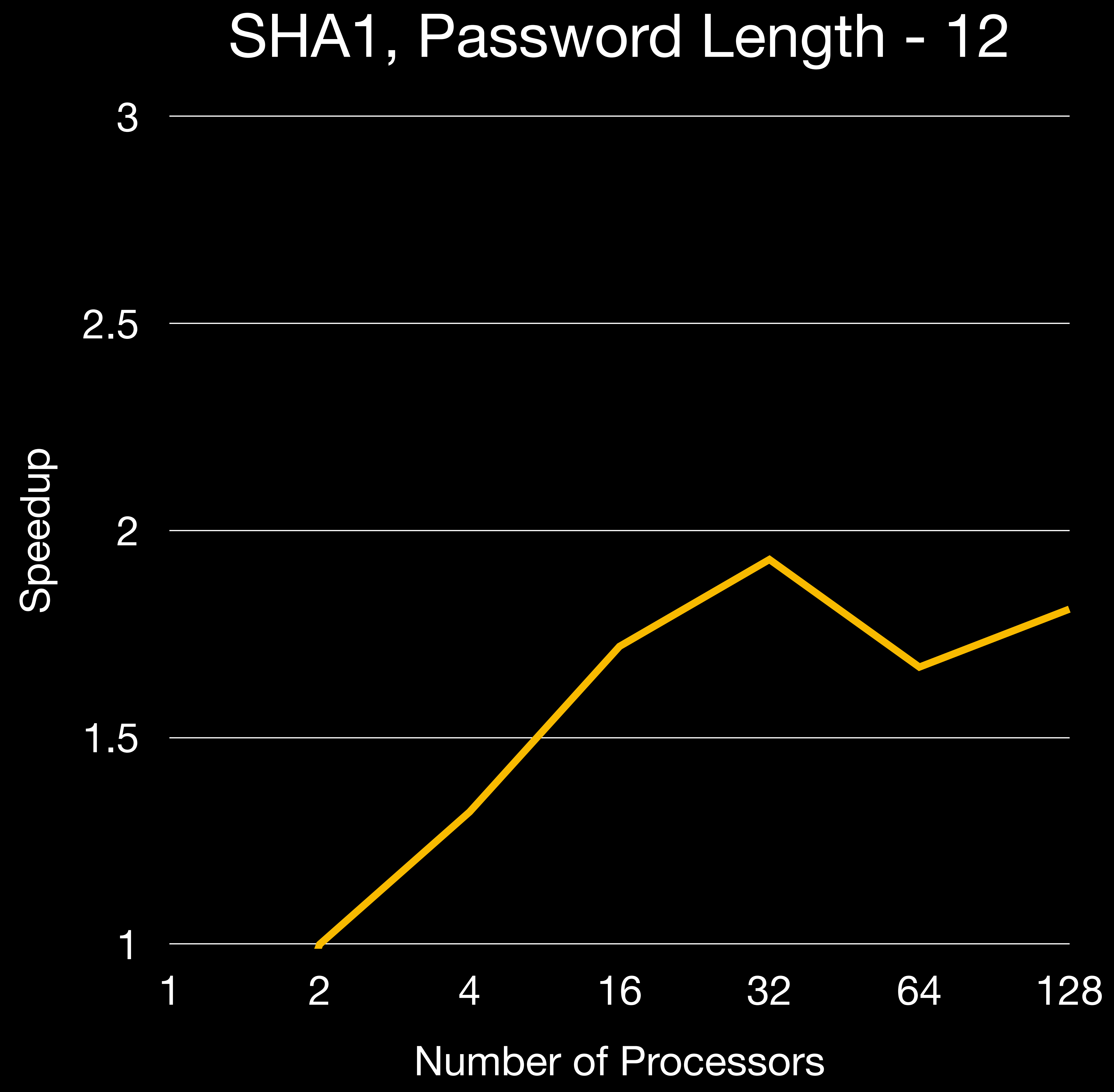
PEs	Time (secs)
1	312.9
2	257.3
4	266.5
16	187.2
32	161.1
64	157.2
128	178.7



Results - Speedup

Amdahl's Law

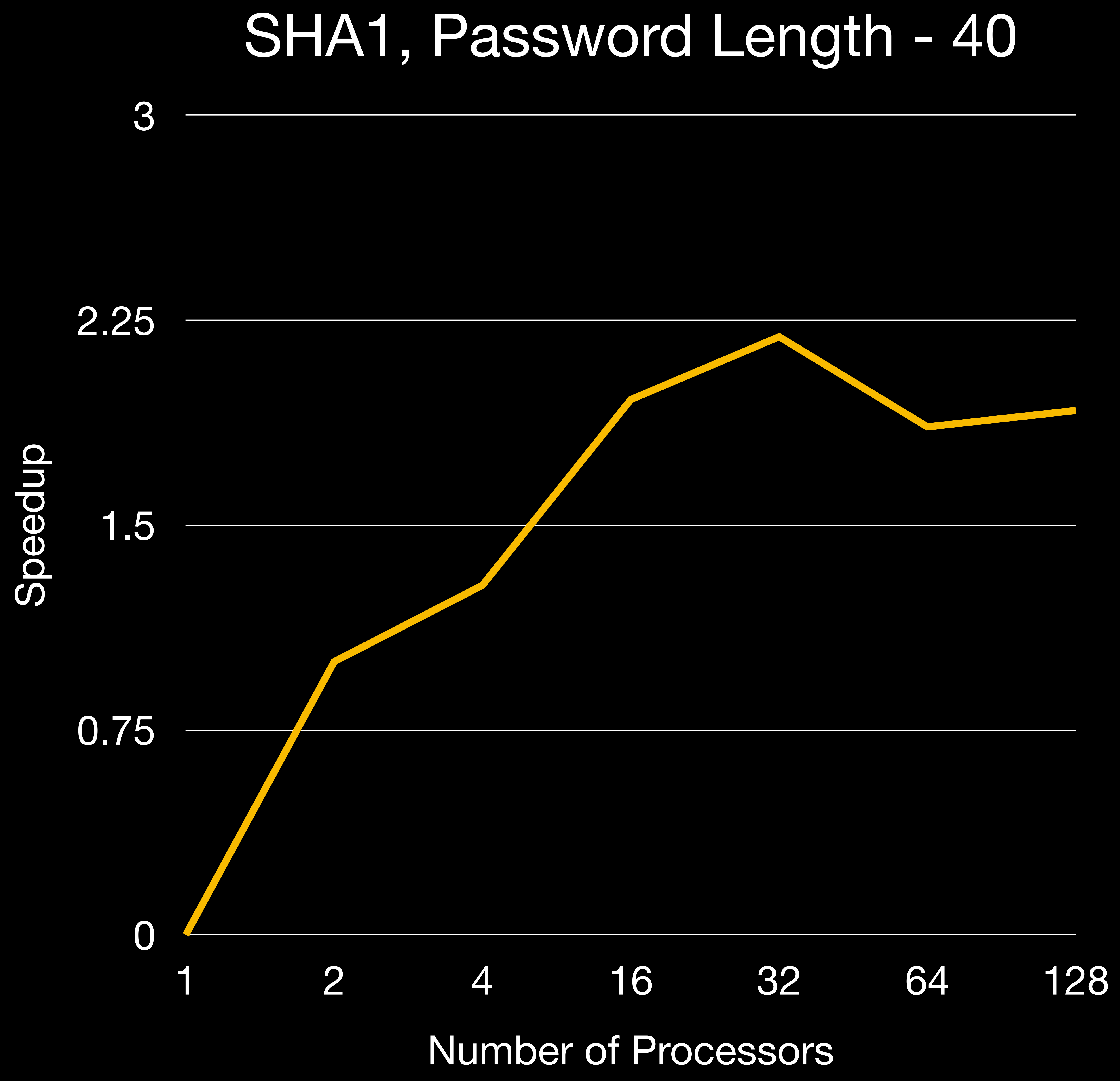
PEs	Speedup
1	-
2	1
4	1.32
16	1.72
32	1.93
64	1.67
128	1.81



Results - Speedup

Amdahl's Law

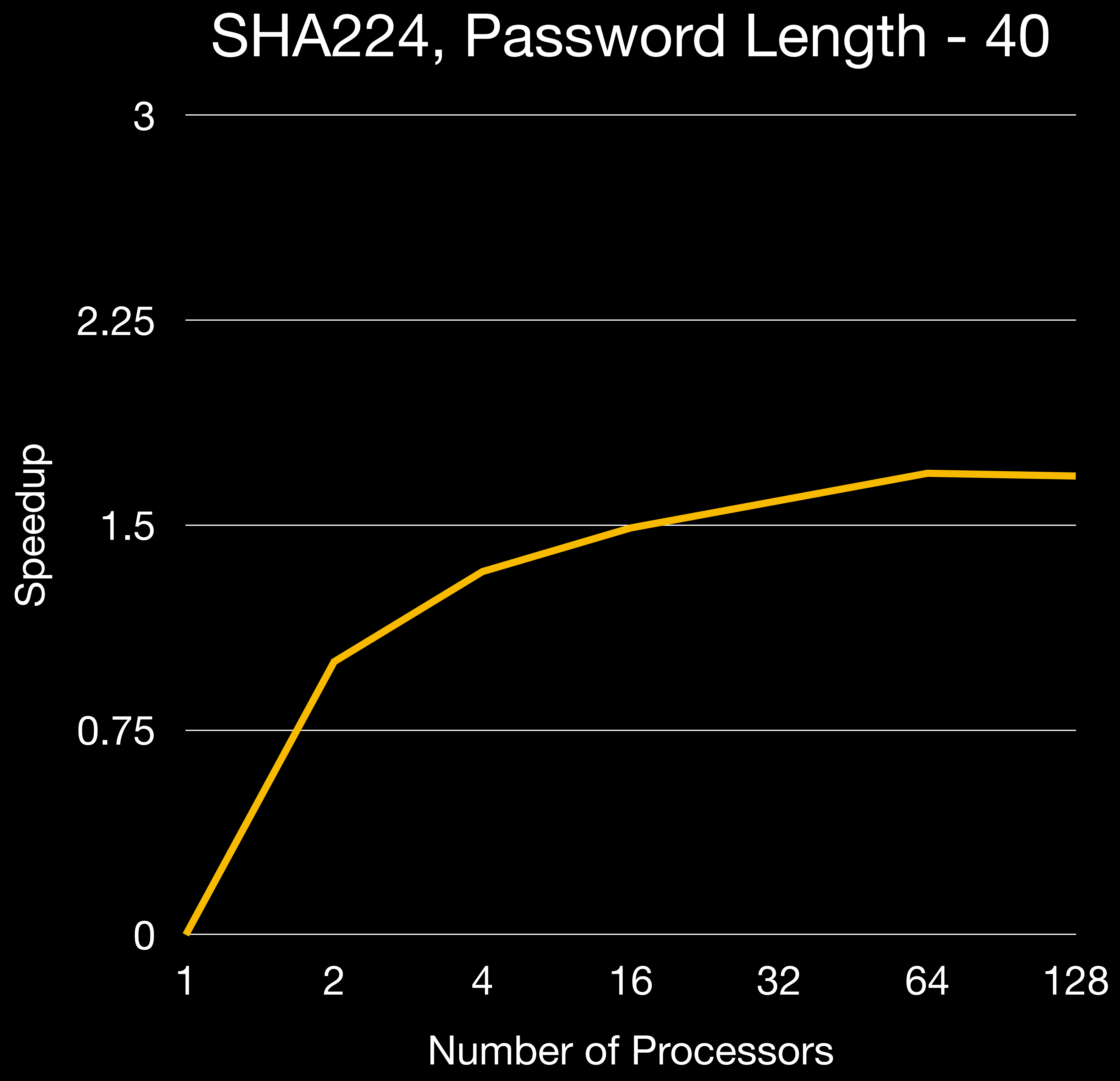
PEs	Speedup
1	-
2	1
4	1.28
16	1.96
32	2.19
64	1.86
128	1.92



Results - Speedup

Amdahl's Law

PEs	Speedup
1	-
2	1
4	1.33
16	1.49
32	1.59
64	1.69
128	1.68

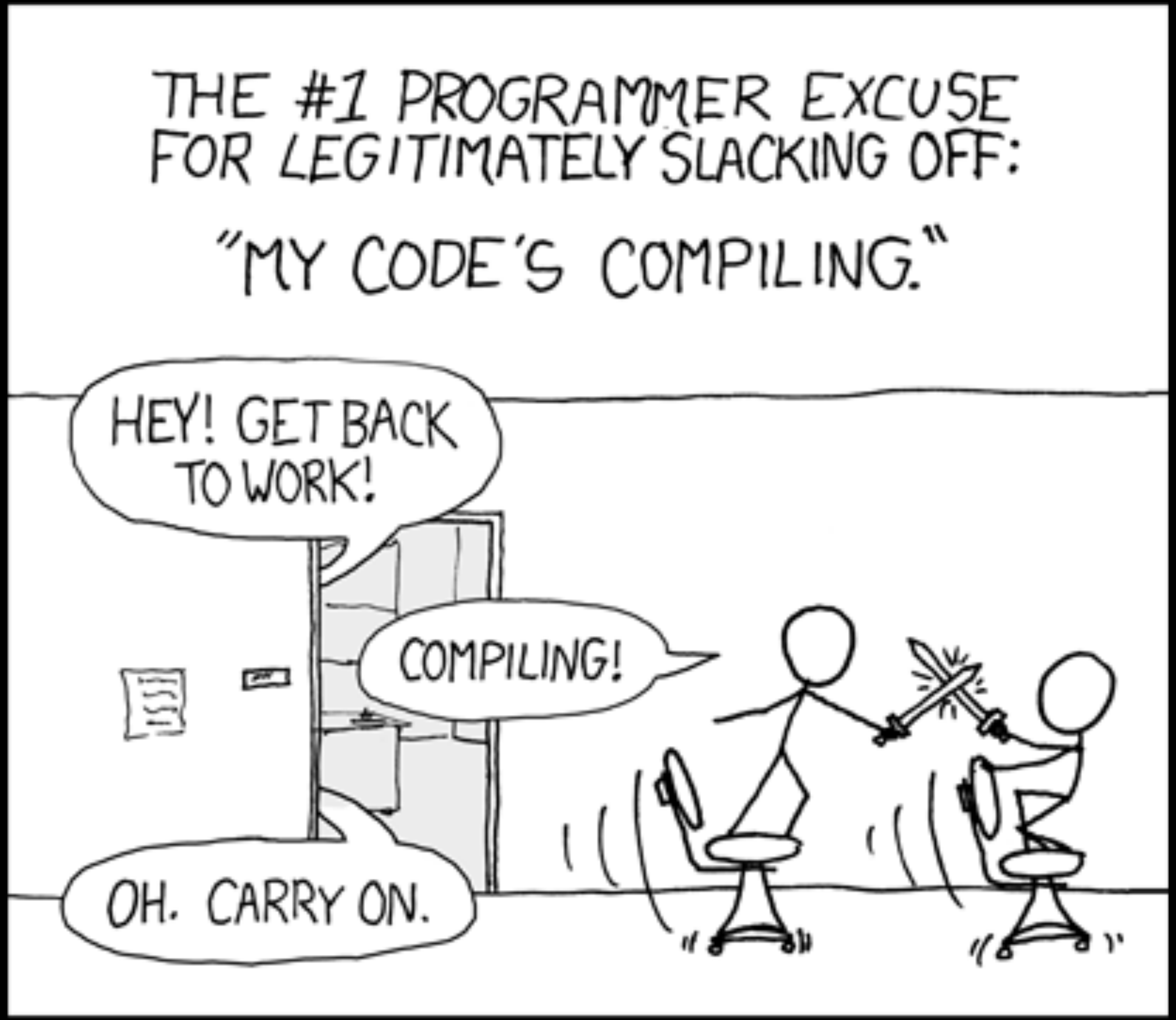


Discourse

Takeaway #1

Parallelisation is not always a silver bullet

- The initial algorithm's approach of splitting into chunks and exchanging communication resulted in wasted operations and idle cycles.
- Even with batching results, idle time was high
- Long compile times meant not enough time to run diverse experiments



Source: xkcd

Takeaway #2

Overengineering = Spaghetti Code

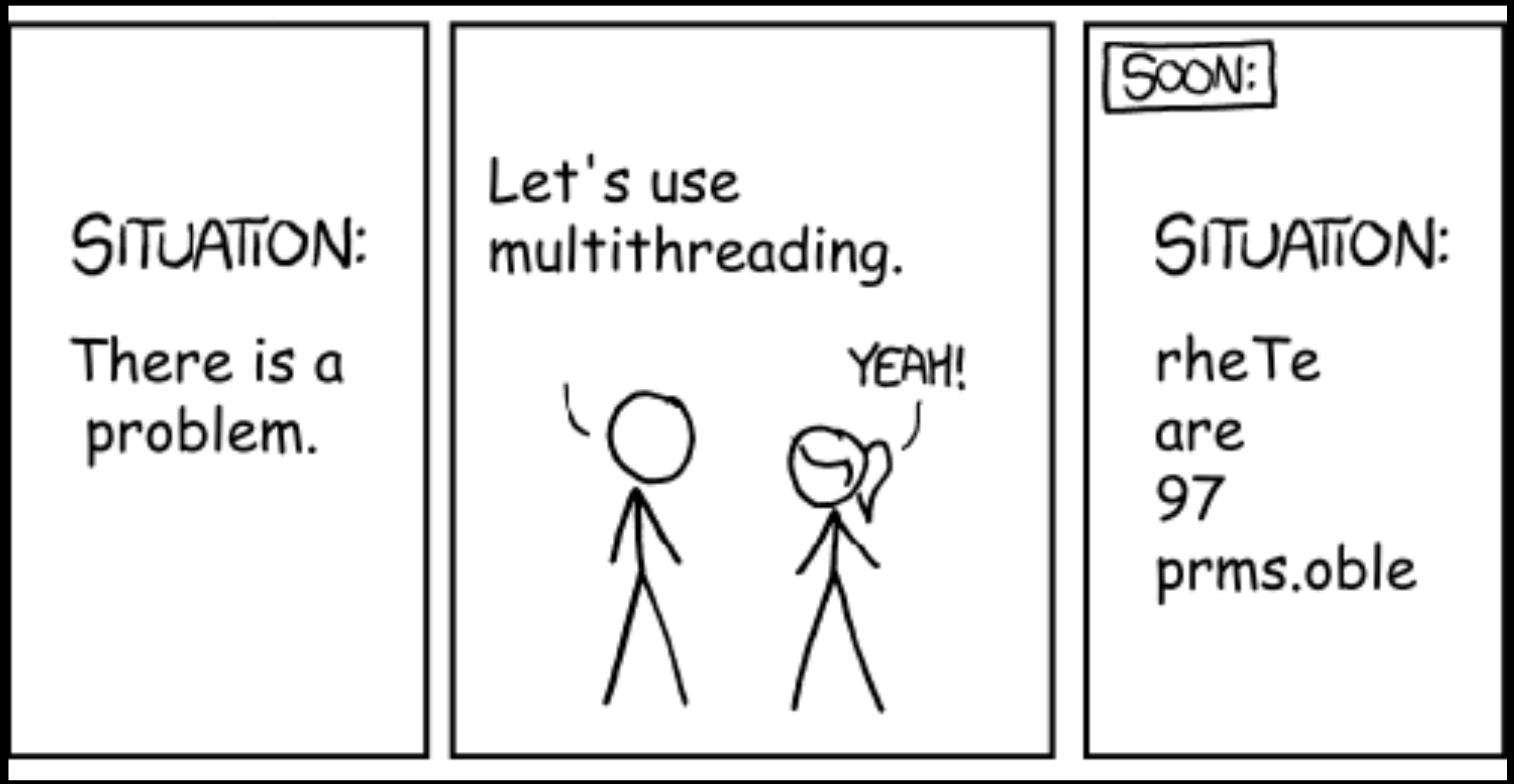
- The attempt to cleverly avoid idle time ended up introducing additional idle time in unintended ways.
- Over-complicating a straightforward algorithm inevitably leads to the development of convoluted and tangled spaghetti code.
- K.I.S.S prevails.

Source: xkcd

Takeaway #3

More Cores, Same Problems

- A subpar speedup was observed when utilising a maximum of 128 cores, despite minimal communication between the processors.
- The introduction of threads resulted in the emergence of synchronisation issues.
- Gated by SHA1 hash generation not being ‘parallelisable’.

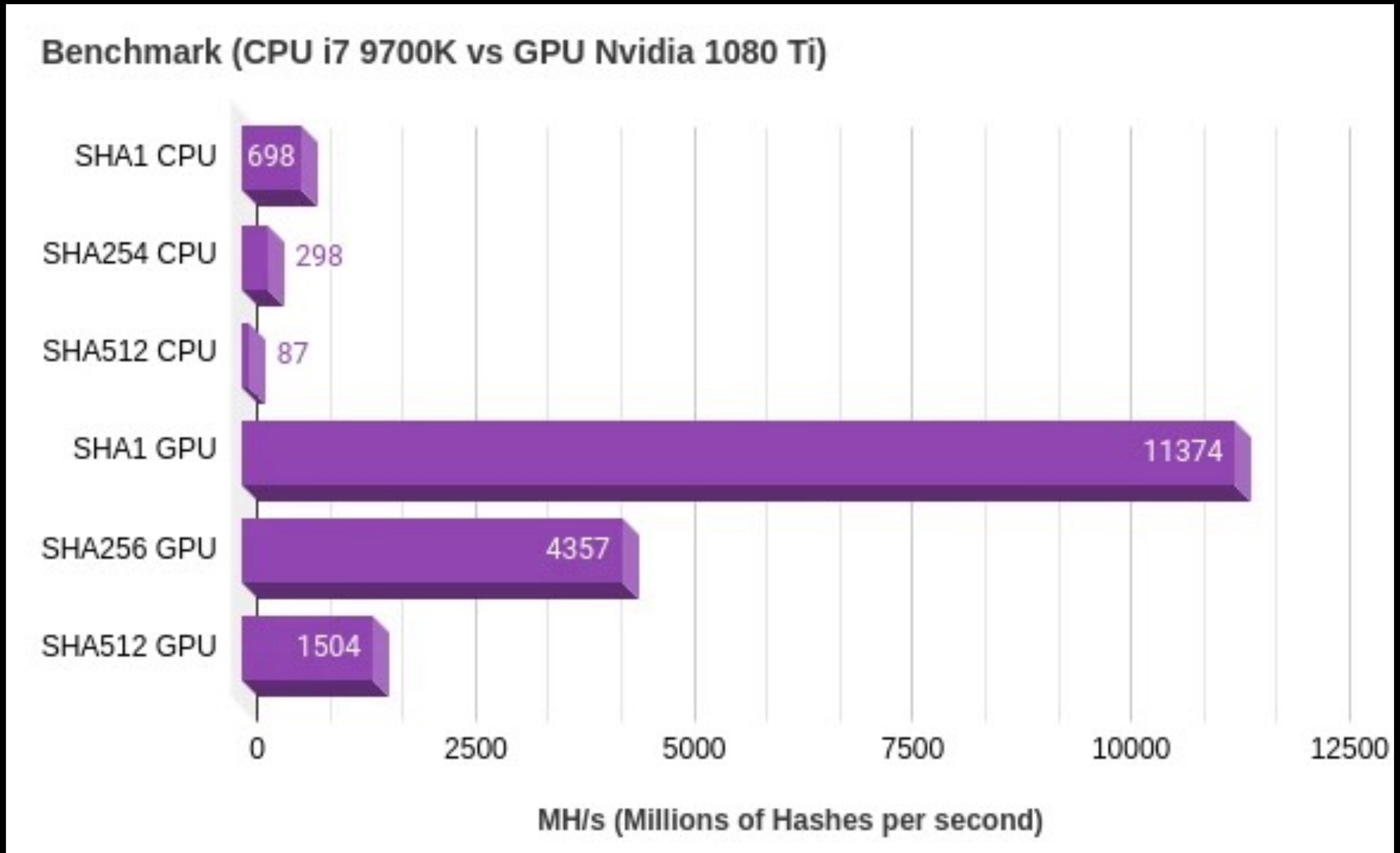


Source: xkcd

Extension #1

GPUs

- GPUs exhibit hash rates that are 20 times greater than CPUs when it comes to generating hashes.
- CUDA Cores further simplify and optimise the hashing process, out of the box (OOTB).

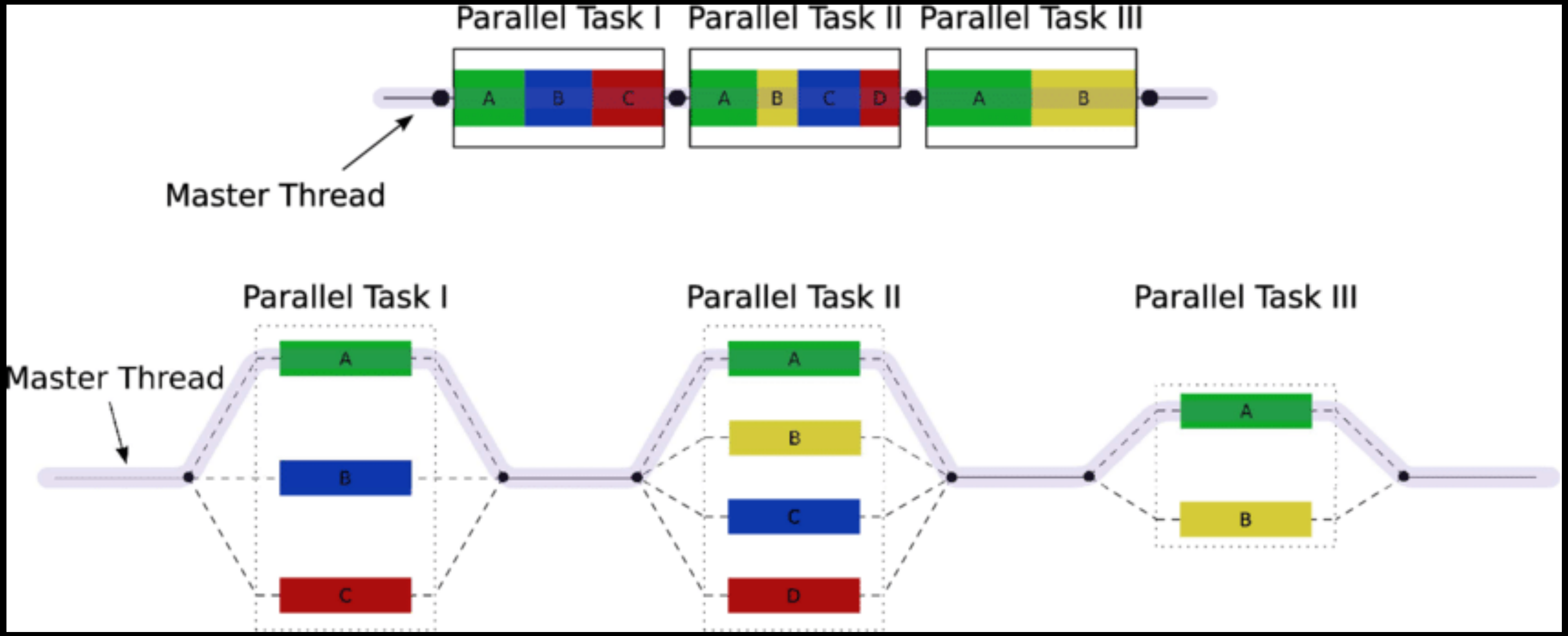


Source: Hashcat

Extension #2

OpenMP

- Pragma OMP Reduction only optimises one variable and only supports few basic ops
- By virtualising both password generation and password hashing, the occurrence of idle or no-op cycles is further minimised or even eliminated.



Source: ResearchGate

Questions?

and Thank You

References

- Super Magic Hashes
- Chick3nMan and Spaze F0rze - Twitter
- PHP Magic Hashes
- SHA1 - Auth.0
- SHA-1 Collision
- SHA-1 CPU Extensions - Intel
- CCR Batching and Open MPI reference
- CCR Batch Jobs and Clusters
- ResearchGate - OpenMP