Data Mining in Rust

Chris Pearce
Rust AKL Meetup, 16 Oct 2018

about:me

- Software Engineer, Firefox, since 2007.
- http://github.com/cpearce
- Mostly worked on HTML5 <video> in Firefox in C++
- Married to Dr Yun Sing Koh, Senior Lecturer, CS @ University of Auckland
    - Data mining & machine learning expert
    - https://www.cs.auckland.ac.nz/~yunsing/
- Association rule mining implementor

Agenda

1. Rust borrow checker rules
2. Describe Association Rule Data Mining
3. Explain FP Growth Algorithm
4. Discuss challenges of using Rust
5. Describe optimizing implementation of FPGrowth
6. Parallelism & concurrency

Rust's Rules of References

[The Rust Programming Language, Chapter 4.2]

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.
- At any given time, you can have either (but not both of):
  - one mutable reference or,
  - any number of immutable references.
- References must always be valid.

Rules of References; Example

```
let mut s = String::from("hello");

let r1 = &s; // no problem

let r2 = &s; // no problem

let r3 = &mut s; // Error!



error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
```

Association Rule Mining in Rust

https://github.com/cpearce/arm-rs/

See also...

https://github.com/cpearce/armpy (Python3)

https://github.com/cpearce/arm-java (Java8)

https://github.com/cpearce/HARM (ugly C++)

Association Rule Mining: ~~Market basket~~ *Candy Bar* analysis

- Given data set of transactions, find associations between items.

Transaction ID	Items
1	popcorn, coke, choctop, ...
2	wine, jaffas, choctop...
3	coke, crisps, M&Ms, ...
…	...

Two sub problems

1. Generating frequent patterns
2. Generating associations rules

Generating Frequent Patterns

- Input: set of transactions
- Find items that occur together “frequently”.
- “Frequent” defined as more than “minimum support threshold”.
- Minimum support is a tunable parameter.
- Output: set of itemsets that co-occur, along with their frequencies.

Generating association rules

- Input: set of frequent itemsets; {{a,b,c}, {a,g,f}, … }
- Output: set of rules of the form {a -> bc, b -> ac, … }

Generating association rules

Given {popcorn, coke, choctop} is frequent, generate and test:

popcorn -> coke, choctop	choctop -> coke, popcorn
popcorn -> coke	choctop -> coke
popcorn -> choctop	choctop -> popcorn
coke -> popcorn, choctop	popcorn, choctop -> coke
coke -> choctop	coke, choctop -> popcorn
coke -> popcorn	coke, popcorn -> choctop

Both phases suffer combinatorial explosion...

Find frequent items sets with FP Growth

- Represent transaction data set as “Frequent Pattern Tree”;
- A trie, branches represent items occurring together
- Nodes contain a count
- Recursively build subtree for items to find co-occurrence & frequency.

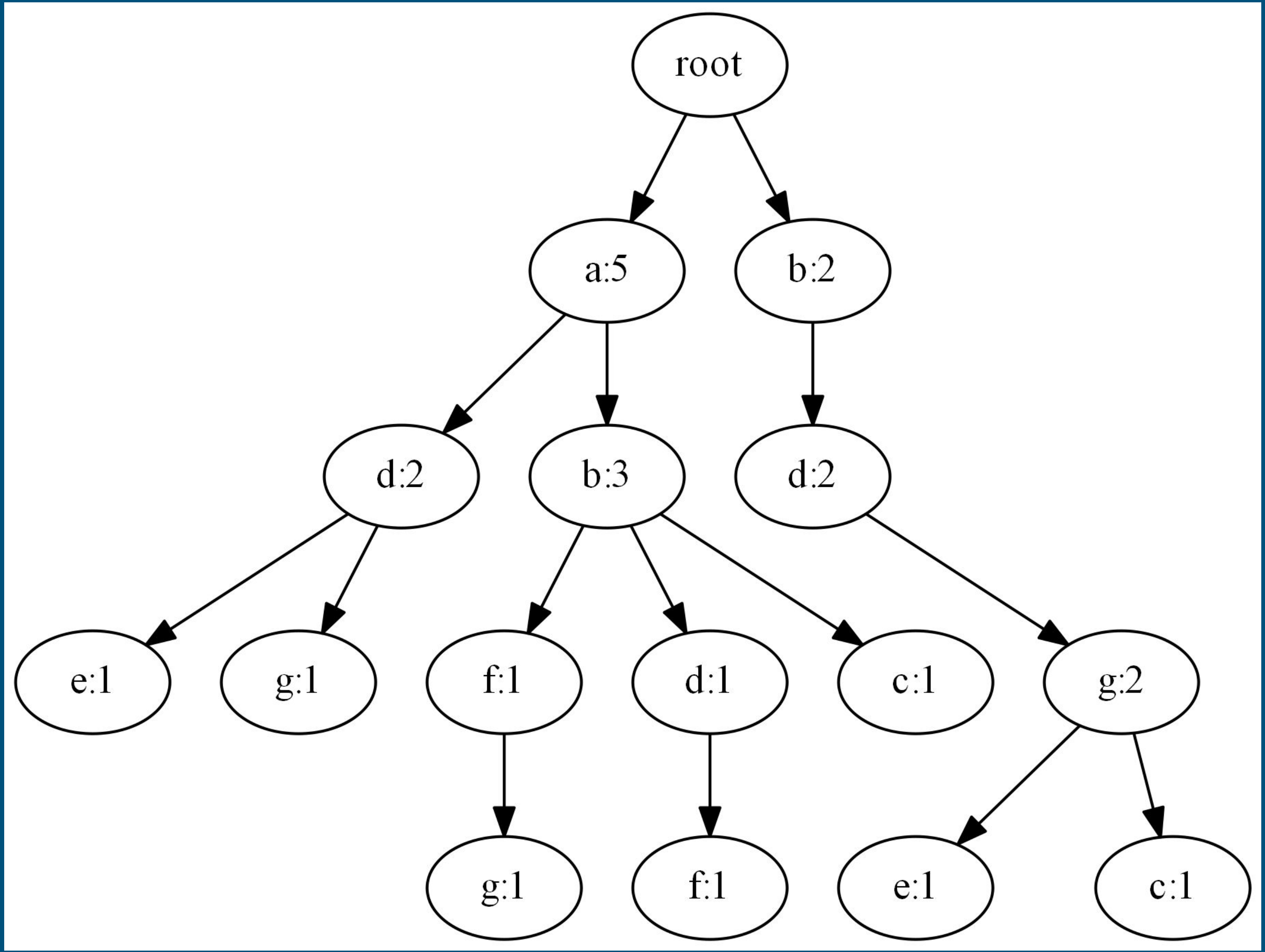
FPGrowth: Pseudo code, initial tree build

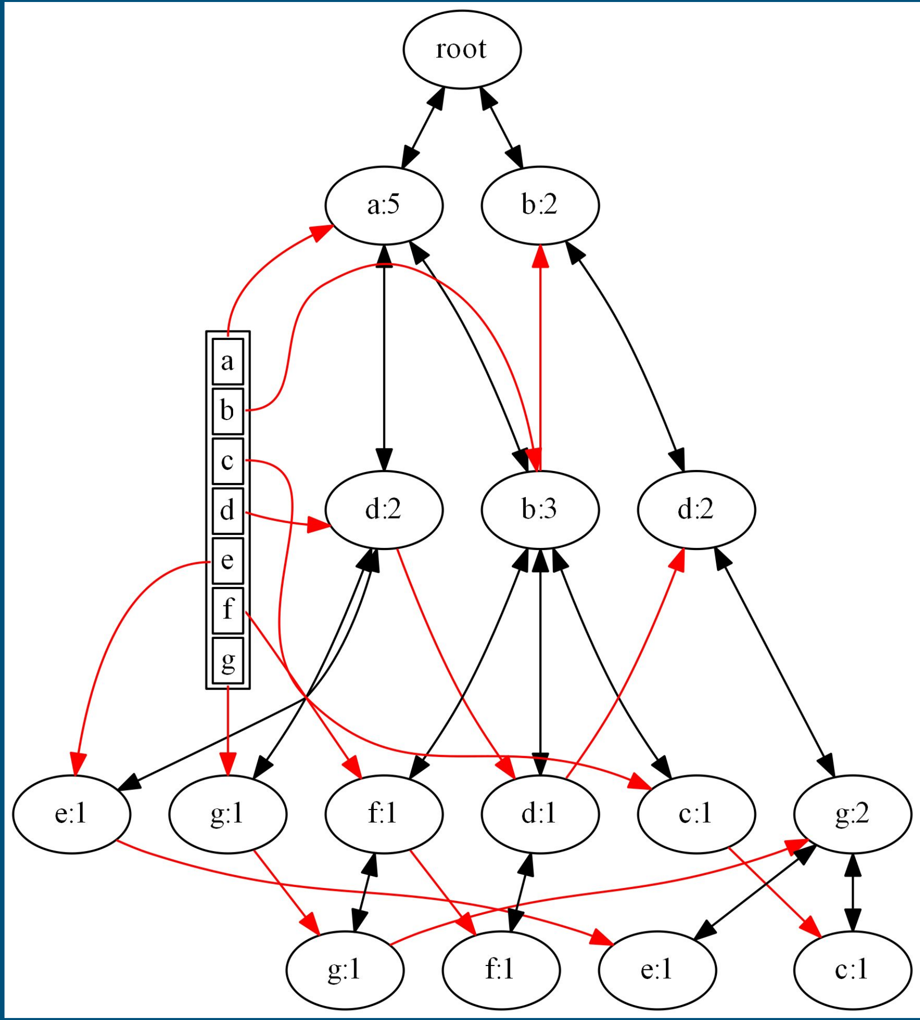
```
For each transaction

     Count item frequencies

For each transaction

     Sort transaction by decreasing frequency, discard infrequent items

     Insert into initial FP tree

minimum_count = num_transactions * minimum_support

FPGrowth(initial FP tree, item_set=[], minimum_count)
```

FPGrowth: Pseudo code

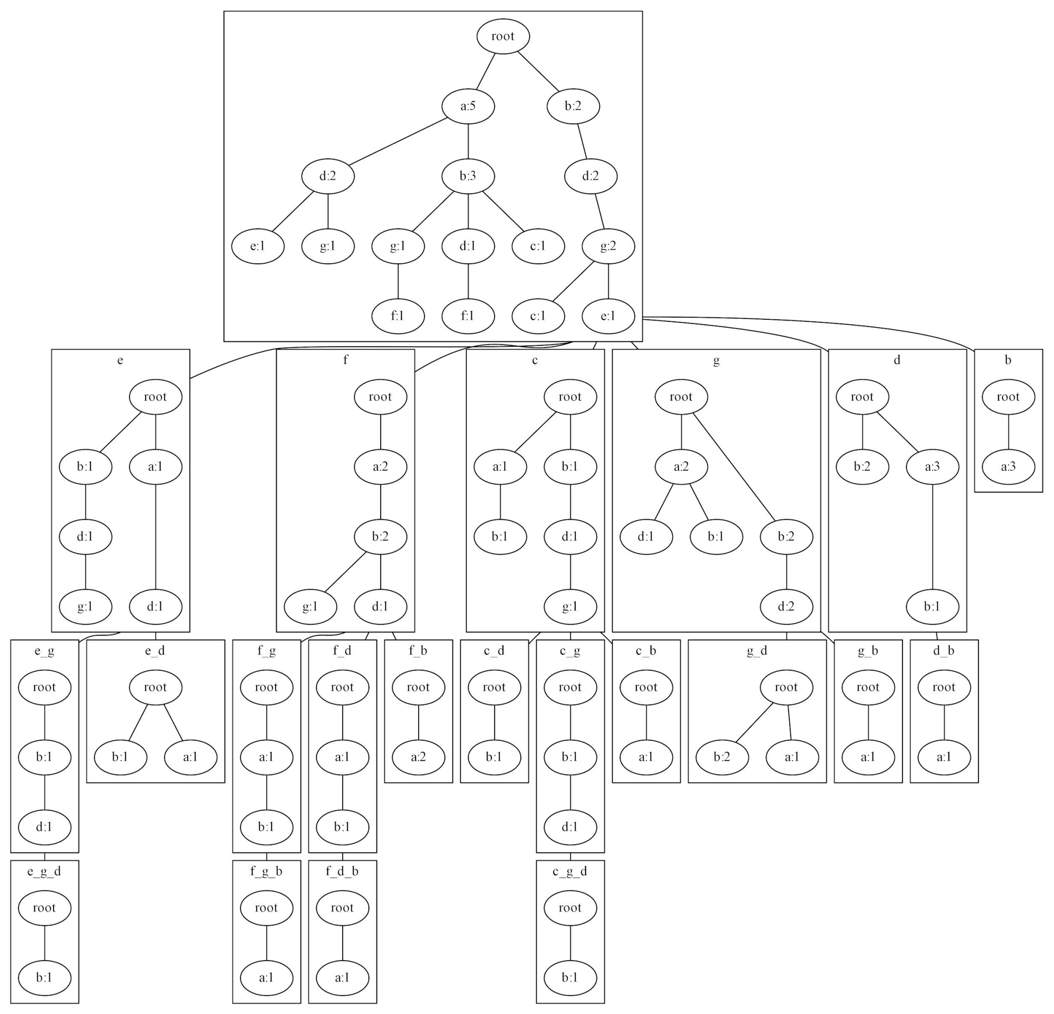
```
FPGrowth(tree, item_set, minimum_count)

    for_each item in tree with count >= minimum_count

        conditional_tree = new FPTree()

        for_each path in tree from item to root

            Insert path excluding item into conditional_tree

        Output(item_set + item, count=conditional_tree.root.count)

        FPGrowth(conditional_tree, item_set + item, minimum_count)
```

TID	Items
1	a, d, e
2	a, b, f, g
3	a, b, d, f
4	a, b, c
5	a, d, g
6	b, c, d, g
7	b, d, e, g





root
a:5
b:2
a
b
c
d
e
f
g
d:2
b:3
d:2
e:1
g:1
f:1
d:1
c:1
g:2
g:1
f:1
e:1
c:1



root
a:5 b:2
d:2 b:3 d:2
e:1 g:1 g:1 d:1 c:1 g:2
f:1 f:1 c:1 e:1
e
root
b:1 a:1
d:1
g:1 d:1
f
root
a:2
b:2
g:1 d:1
c
root
a:1 b:1
b:1 d:1
g:1
g
root
a:2
d:1 b:1 b:2
d:2
d
root
b:2 a:3
b:1
b
root
a:3
e_g
root
b:1
d:1
e_d
root
b:1 a:1
f_g
root
a:1
b:1
f_d
root
a:1
b:1
f_b
root
a:2
c_d
root
b:1
c_g
root
b:1
d:1
c_b
root
a:1
g_d
root
b:2 a:1
g_b
root
a:1
d_b
root
a:1
e_g_d
root
b:1
f_g_b
root
a:1
f_d_b
root
a:1
c_g_d
root
b:1

Naive tree node Rust code...

```
struct FPNode {
    item: Item,
    count: u32,
    children: Vec<FPNode>,
    parent: &FPNode,
}
```

```
struct FPTree {
    root: FPNode,
    item_list: HashMap<Item, Vec<&FPNode>>,
}
```

Naive tree node Rust code...

```
struct FPNode {
    item: Item,
    count: u32,
    children: Vec<FPNode>,
    parent: &FPNode,
}
```

```
struct FPTree {
    root: FPNode,
    item_list: HashMap<Item, Vec<&FPNode>>,
}
```

Problem: can’t borrow nodes immutably during building phase!

Solution #1...

- Don’t store reference to node’s parent in node.
- Don’t maintain item list.
- After building tree, traverse tree, create index nodes’ parents and item list.
- Actually not terrible for performance...

Solution #2

- Store tree as a Vec<FPNode>.
- FPNodes store their parent's index.
- FPTree stores list of FPNodes' indicies for each Item.
- Performance about the same as solution #1.
- No borrow checker shenanigans.
- Code is not as simple.

FPTree backed by an array

```
struct FPNode {
    item: Item,
    count: u32,
    children: Vec<usize>,
    parent: usize,
}
```

```
struct FPTree {
    nodes: Vec<FPNode>,
    item_list: HashMap<Item, Vec<usize>>,
}
```

Other challenges…

Hash tables!

- Allocates lots of spare capacity
  - Reduced memory by 10X when switched nodes’ children HashMap<Item,FPNode> to Vec<FPNode>!
- Very slow hashing function for primitive types.
  - FnvHash helps…
  - Replaced some HashMap<Item, T> with Vec<T>, indexed by Item as usize.
    - (May be bad idea on very big data sets)

Hash tables! (cont.)

- Use sorted Vec<Item> instead of HashSet<Item>.
  - O(N) union/intersection.
    - 10% speed up by pre-calculating output Vec<Item> capacity!
- Can reorder Item indices by ItemName lexicographically
  - Sorted output for free!

Strings! Oh my!

- Reducing string allocations was key to improving output performance.
- Rust made it easy to create (and remove!) unnecessary allocations.
- Java implementation was actually quite fast.
  - I think due to strings & objects being passed by reference by default.

Parallelism!

Fearless concurrency

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized

https://doc.rust-lang.org/beta/nomicon/races.html

Concurrency

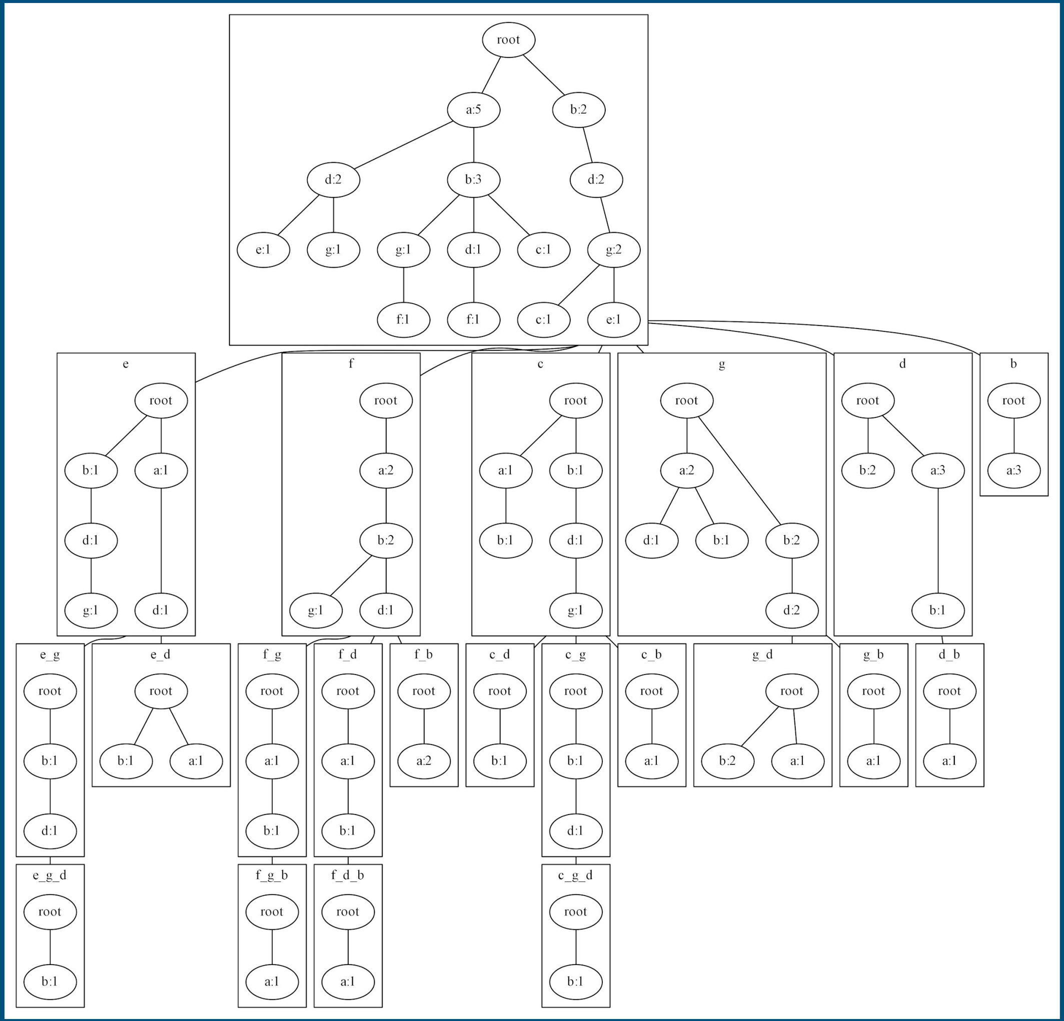
- Rust supports Go-style channels
  - “Do not communicate by sharing memory; instead, share memory by communicating.”
- Rust’s std::threads are OS threads
- Rust’s std::Mutex locks data not code

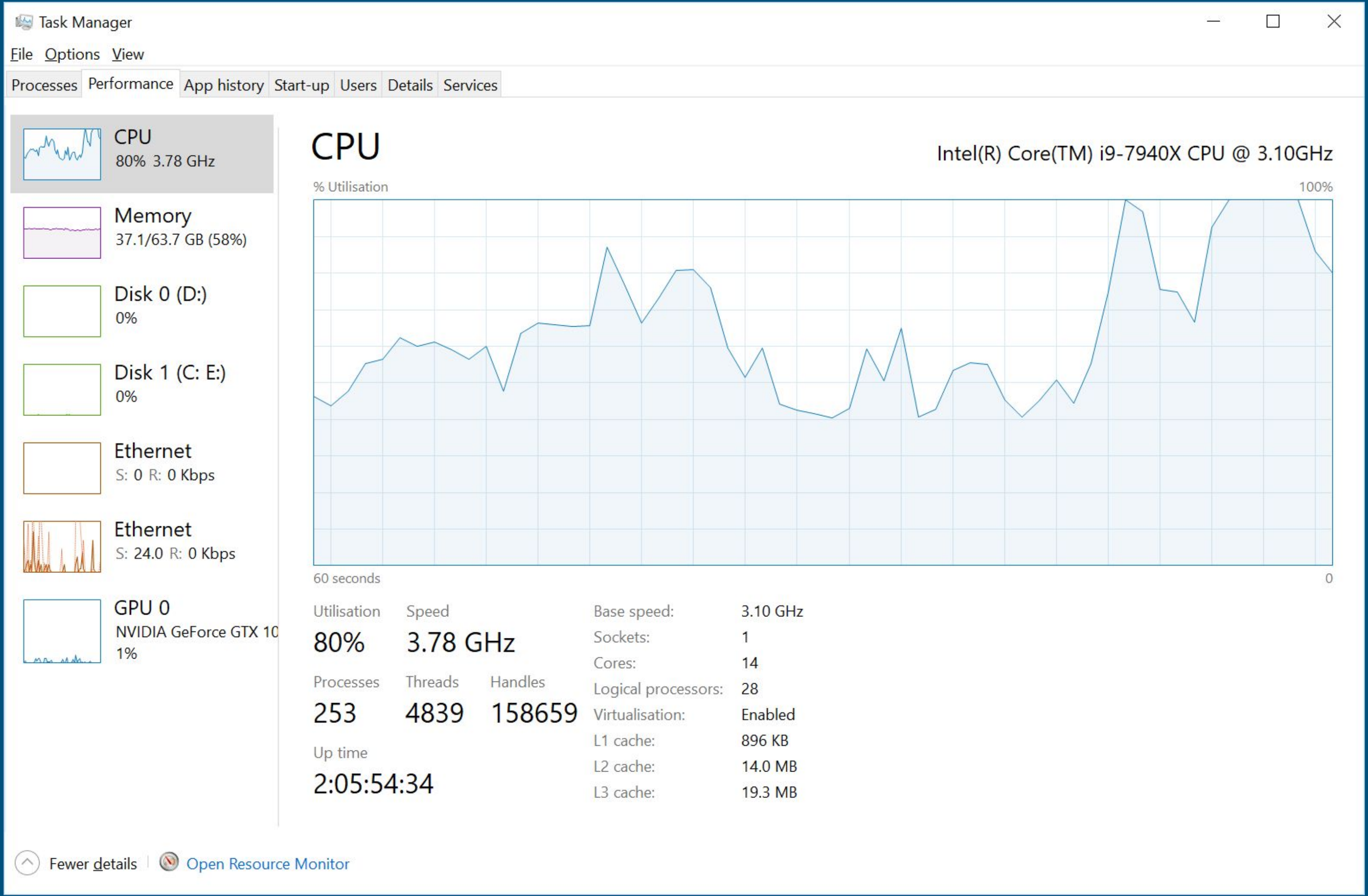
Rayon: Rust’s super power

Safe foundations

- Parallel iterators make it easy to parallelize sequential code.
- Possible because everything there is Sync;
  - Can be shared across threads
- Rayon implements functional for par_iter();
  - map, for_each, filter, fold

Pick the right iter() to paralellize


root
a:5
b:2
d:2
b:3
d:2
e:1
g:1
g:1
d:1
c:1
g:2
f:1
f:1
c:1
e:1
e
root
b:1
a:1
d:1
g:1
d:1
f
root
a:2
b:2
g:1
d:1
c
root
a:1
b:1
b:1
d:1
g:1
g
root
a:2
d:1
b:1
b:2
d:2
d
root
b:2
a:3
b:1
b
root
a:3
e_g
root
b:1
d:1
e_d
root
b:1
a:1
f_g
root
a:1
b:1
f_d
root
a:1
b:1
f_b
root
a:2
c_d
root
b:1
c_g
root
b:1
d:1
c_b
root
a:1
g_d
root
b:2
a:1
g_b
root
a:1
d_b
root
a:1
e_g_d
root
b:1
f_g_b
root
a:1
f_d_b
root
a:1
c_g_d
root
b:1


Task Manager
File Options View
Processes
Performance
App history
Start-up
Users
Details
Services
CPU
80% 3.78 GHz
Memory
37.1/63.7 GB (58%)
Disk 0 (D:)
0%
Disk 1 (C: E:)
0%
Ethernet
S: 0 R: 0 Kbps
Ethernet
S: 24.0 R: 0 Kbps
GPU 0
NVIDIA GeForce GTX 10
1%
CPU
Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz
% Utilisation
100%
60 seconds
0
Utilisation
80%
Speed
3.78 GHz
Processes
253
Threads
4839
Handles
158659
Up time
2:05:54:34
Base speed: 3.10 GHz
Sockets: 1
Cores: 14
Logical processors: 28
Virtualisation: Enabled
L1 cache: 896 KB
L2 cache: 14.0 MB
L3 cache: 19.3 MB
Fewer details
Open Resource Monitor

Summary

- Rust makes concurrency/parallelism easy and safe.
- Rayon is awesome.
- Rust can make the nodes in your cluster faster.
- Use Rust!

Questions?

https://github.com/cpearce/arm-rs/
chris@pearce.org.nz