GridGain

Scalable Distributed Computing with Groovy Using Apache Ignite

or

"how to solve weird Rubik's cubes with brute force compute power"

Jeremy Meyer – Director Education and Professional Services

GridGain Systems, October 2023

Inc. 5000 · 500 Technology Fast 500 2020 NORTH AMERICA Deloitte. · DBTA 100 · RED HERRING GLOBAL 100 WINNER · RED HERRING N. AMERICA WINNER 100 · Fast Private · Gartner CoolVendor · GOLD 2023 STEVIE WINNER AMERICAN BUSINESS AWARDS

Speaker: Jeremy Meyer

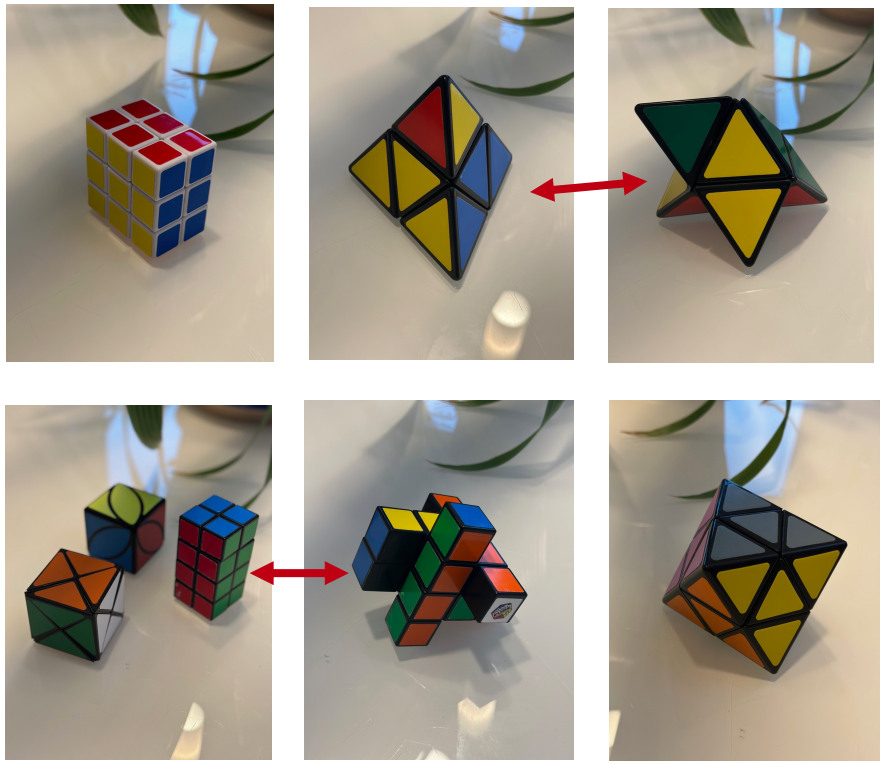
Jeremy Meyer heads up the Professional Services and Education team at GridGain Systems, the creators and sponsors of the Apache Ignite project.

He is a computer scientist, philosopher, coder, hobbyist, very rare blogger and writer as well as a lover of good design and good puzzles

Bad at solving the Rubik's cube

Introduction

- Record for solving the 3x3x3 rubik's cube - never
- I love the engineering and design of oddly shaped cubes, so I collect them.
- Permutations of these cubes are generally simpler than the original 3x3x3
- I wrote some naïve, recursive algorithms to help solve certain moves (like non-destructive corner swaps)



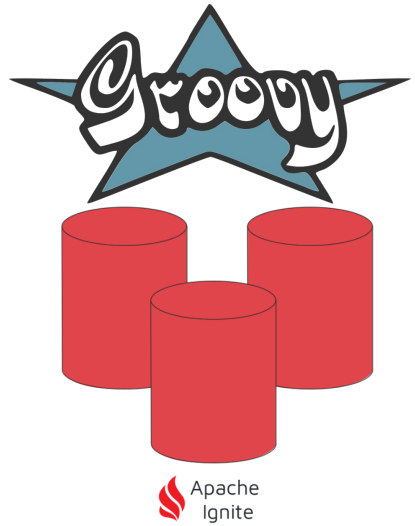
Why Groovy with Ignite?

Paul King (who you have just heard) gave me the idea with his Whisky presentation at the GridGain Apache Ignite summit

Ignite compute grid is quite Java focused, Groovy is very compatible

“What if we used the dynamic, easy to code and prototype aspect of Groovy..
..with the fantastically scalable compute power of Apache Ignite’s compute grid, and clever peer class loading?” *

I will share my journey with you..



*(and fixed the problem of embarrassingly unsolved cubes on my coffee table?)

What is Apache Ignite?

Apache Ignite is a distributed database management system for high-performance computing and can be used to power in-memory apps, as a cache, or an in-memory database, or datagrid sitting between an application and third-party databases.

Apache Ignite can help to:

Build real-time and event-driven solutions that process data with in-memory speed

Scale up and out across available memory and disk capacity

Take advantage of built-in SQL, high-performance computing and real-time processing APIs

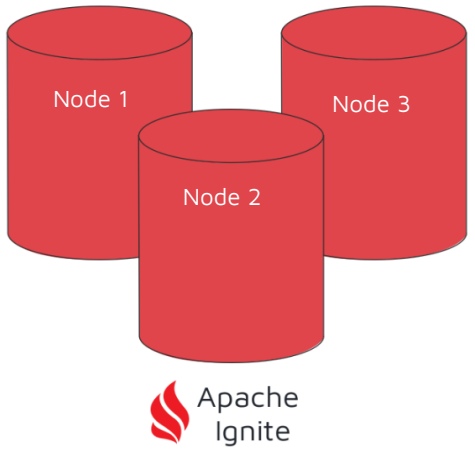
A little more about Ignite

Ignite automatically distributes data across partitions and nodes. Compute tasks can be distributed, too. Collocation of computations and in-memory data is Ignite's secret sauce.

A thick client connects to the cluster as its own node, and can run tasks, thin clients can connect via JDBC and more lightweight protocols

Tasks can be Java closures, or Runnable classes

Key Concept: Class files can be copied to server nodes..or.. serialized automatically via "peer class loading"



Some numbers to start..

Don't worry - in depth combinatorial theory or Rubik's cube math/efficiency are not needed here

World record for a human solving the entire regular rubik's cube - **3.8 secs**.

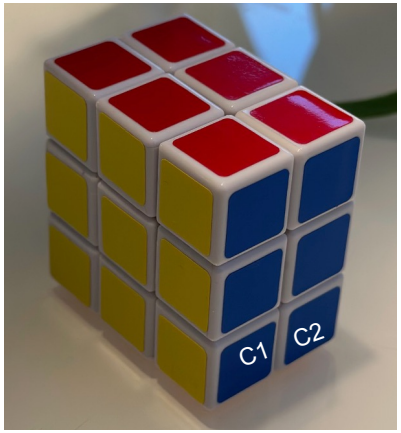
World record for a robot solving the regular rubik's cube - **0.38 secs**

The 3x3x2 has 12 moves.. So with 10 recursive moves = **62 billion** permutations (with no pruning)

My code in Python for swapping two final corners of a 3x3x2 without messing up the rest of the cube .... **27 hours**

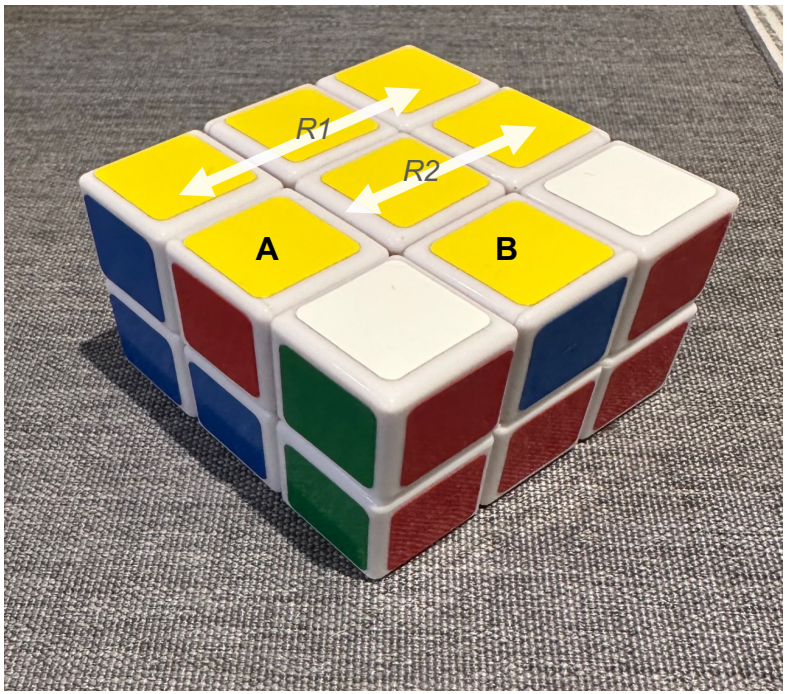
My code in Groovy for swapping two edge pieces (simpler) without messing up the top layers .... **5 minutes**

This is what I made the focus of the study



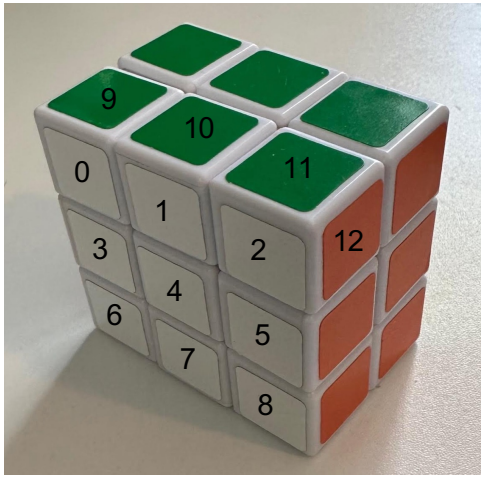
The problem space..

- This cube – 3x3x2
- Swapping piece A with piece B
- Rows R1 and R2 must remain the same
- Can be solved in 7 moves
  - (Max recursion depth of 6)



The Algorithm

- Represent cube as array
  cube = (0..41).toList()
- Define “moves”, transforms which turn the cube and map pieces to different places
  moves = ['front_anticlock', 'left_col_rot', 'back_anticlock', 'front_180.. etc.
- Implement moves as a series of swaps
  'front_clock': [[0, 6], [1, 3], [2, 0], [3, 7], [4, 4], [5, 1], [6, 8], [7, 5] etc.
  'front_anticlock': [[0, 2], [1, 5], [2, 8], [3, 1], [5, 7] etc.
- Define Source and Target positions
  - Start
  - Test for target position
  - Apply all moves methodically
  - Check for loops and silly (back and forth) moves
  - Recurse to maximum specified depth.



The Algorithm in pseudo code

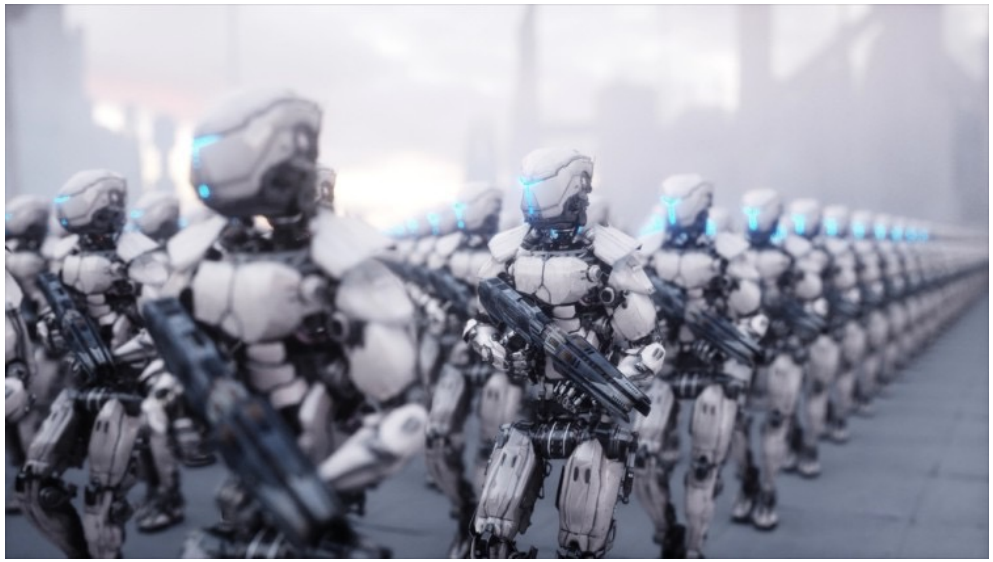
```
def findPaths(depth, source, target, move) {
   if (depth > MAX_DEPTH or isSilly(move)) {
     return false
   }
    def res = applyMove(move_dict[move], source)
    if (compare(res, target)) {
     solutions.add(solution)
     return true
     }
    for (each_move in move_dict) {
     findPaths(depth + 1, res, target, move_dict[each_move])
    }
}
```

The plan.. vs. the approach

I wanted the clean, dynamic, easy to write, prototyping approach of Groovy running in an almost continuous deployment dev environment, with an easy workflow to high-performance code in QA/Live

1. Creating a thick Ignite client in Groovy
   - Easy.
2. Passing Groovy scripts, or just a Groovy closure snippet to an Ignite compute task for distribution to the cluster
   - I couldn't do that without precompiling the Groovy with **groovyc**
   - I didn't want to do that
3. Finally settled on an Ignite Java "harness" task which took a Groovy script as a parameter and parsed and ran it on each node
   - Gave me dynamic script writing /loading/prototyping
   - But .. gives an overhead to a task – could be improved

Testing – The fantasy version



- 100 nodes running on ThinkSystem SR665 V3 Rack Mounted Servers
- Budget - US$600k
- Not approved by Finance

Testing – The reality

Cobbled together some machines I had.

- An old Lenovo Laptop, Intel I5, Dual Core, 1.6ghz 8Ghz, running Ubuntu Linux
- A “Beelink” Celeron J3455 Quad Core 2.3 GHz, mini PC, 4GB RAM running Windows
- A Raspberry Pi 4 Quad Core Arm V8 1.5 Ghz, 4GB RAM, running Raspbian
- My old MacBook Air, Quad Core 1.6 Ghz, 16GB RAM, MacOS
- My work MacBook Pro M1, 8 Core, 16GB RAM, MacOS

Budget: $0

Finance approved!

Testing – The real reality.

Cobbled together some machines I had.

- An old Lenovo Laptop, Intel I5, Dual Core, 1.6ghz 8Ghz, running Ubuntu Linux
- A “Beelink” Celeron J3455 Quad Core 2.3 GHz, mini PC, 4GB RAM running Windows
- A Raspberry Pi 4 Quad Core Arm V8 1.5 Ghz, 4GB RAM, running Raspbian
- My old MacBook Air, Quad Core 1.6 Ghz, 16GB RAM, MacOS
- My work MacBook Pro M1, 8 Core, 16GB RAM, MacOS

Budget: $0

Finance approved!

Starting it all up…

Very simple:

- Install the same version of Java on all hardware
- Install the same version of Ignite on all hardware
- Specify Ignite config. on all nodes (e.g. loadbalancing etc.)
- Copy Groovy.jar to classpath of all nodes
- Start them all up from SSH terminals
  - They connect to each other automatically
  - `compute.applyAsync(new IgniteClosure(){..}, parameters)` sends off the tasks to all nodes, using your specified distribution strategy
  - Aggregate answers and that's it!

The GroovyRunner class

```
class GroovyRunner implements IgniteClosure<RemotableGroovyScript, Object> {
    @IgniteInstanceResource
    Ignite ignite;
    public Object apply(RemotableGroovyScript remoteableScript) {
        System.out.println("Starting Groovy Script in Ignite Task:");
        Object res;
        try {
            GroovyShell shell = new GroovyShell(remoteableScript.getBindings());
            String scriptText = remoteableScript.getScript();
            Script script = shell.parse(scriptText);
            res = script.run();
            System.out.println("Script Done." + res);
        } catch (Exception e) {
       // Handling any exceptions
            e.printStackTrace();
            return e.getMessage();
        }
    return res;
    }
}
```

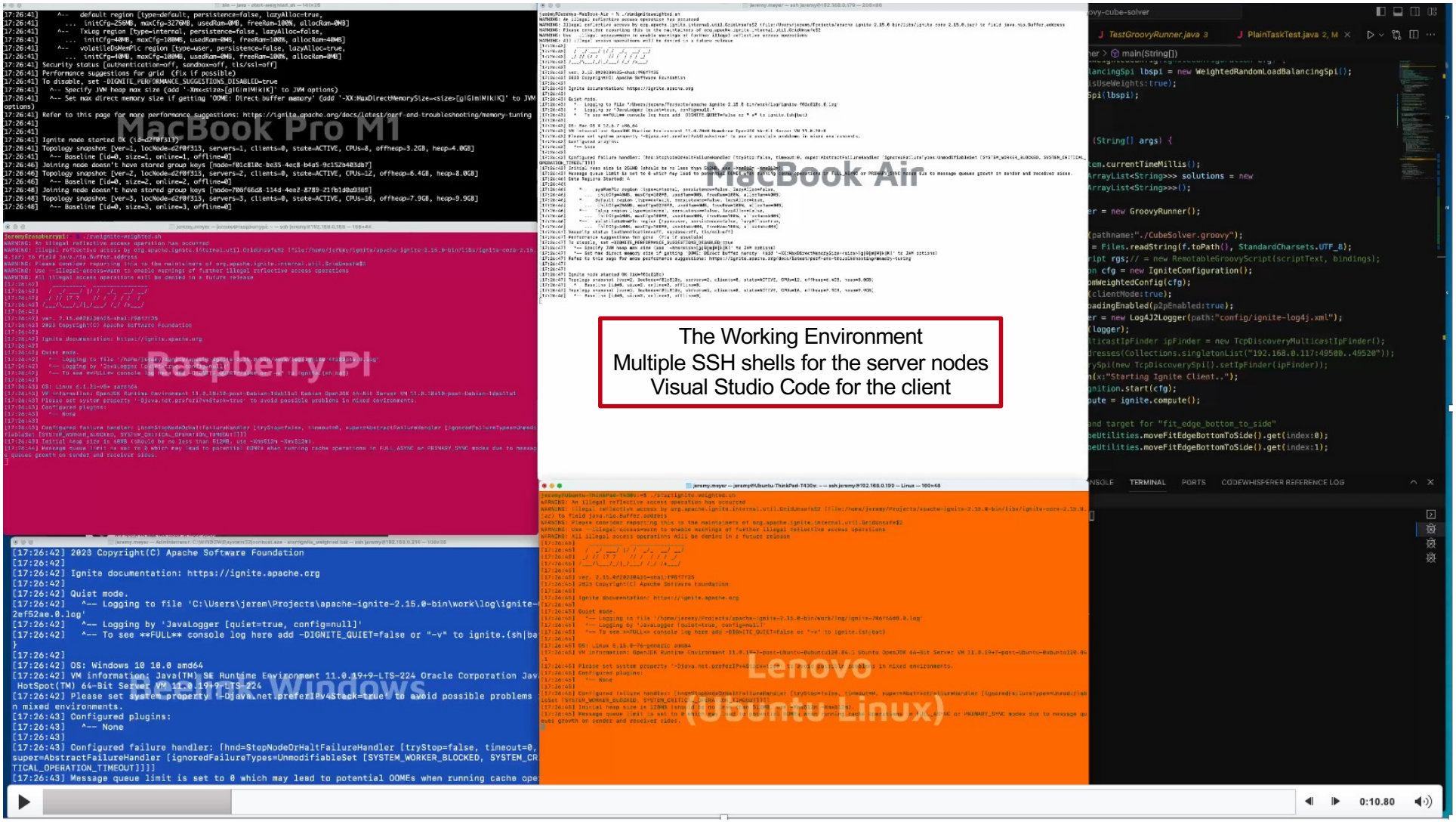
The RemotableGroovyScript

```
import java.util.Map;
import groovy.lang.Binding;

public class RemotableGroovyScript {
    private String script;
    private Binding bindings;
    public RemotableGroovyScript(String script, Map<String,Object> bindings) {
        this.script = script;
        this.bindings = new Binding(bindings);
    }
    public String getScript() {
        return this.script;
    }
    public Binding getBindings() {
        return this.bindings;
    }
    public void setVariable(String variable, Object val) {
            bindings.setVariable(variable, val);
    }
}
```

A snippet of the Groovy Cube Solver ‘moving’

```
move_map = [
'front_clock': [[0, 6], [1, 3], [2, 0], [3, 7], [4, 4], [5, 1], [6, 8], [7, 5], [8, 2], [9, 18],
[10, 19], [11, 20], [12, 9], [13, 10], [14, 11], [15, 12], [16, 13], [17, 14], [18, 15], [19, 16], [20, 17]],
'front_anticlock': [[0, 2], [1, 5], [2, 8], [3, 1], [5, 7], [6, 0], [7, 3], [8, 6], [9, 12], [10, 13],
[11, 14], [12, 15], [13, 16], [14, 17], [15, 18], [16, 19], [17, 20], [18, 9], [19, 10], [20, 11]],
'front_180': [[0, 8], [1, 7], [2, 6], [3, 5], [5, 3], [6, 2], [7, 1], [8, 0], [9, 15], [10, 16],
[11, 17], [12, 18], [13, 19], [14, 20], [15, 9], [16, 10], [17, 11], [18, 12], [19, 13], [20, 14]],
'back_clock': [[21, 27], [22, 24], [23, 21], [24, 28], [25, 25], [26, 22], [27, 29], [28, 26], [29, 23], [30, 39],
[31, 40], [32, 41], [33, 30], [34, 31], [35, 32], [36, 33], [37, 34], [38, 35], [39, 36], [40, 37], [41, 38]],
'back_anticlock': [[21, 23], [22, 26], [23, 29], [24, 22], [26, 28], [27, 21], [28, 24], [29, 27], [30, 33], [31,
34],
[32, 35], [33, 36], [34, 37], [35, 38], [36, 39], [37, 40], [38, 41], [39, 30], [40, 31], [41, 32]],
'back_180': [[21, 29], [22, 28], [23, 27], [24, 26], [26, 24], [27, 23], [28, 22], [29, 21], [30, 36], [31, 37],
[32, 38], [33, 39], [34, 40], [35, 41], [36, 30], [37, 31], [38, 32], [39, 33], [40, 34], [41, 35]].. ETC. ]
]
def applyMove(move, currentPos) {
    def newPos = currentPos.clone()
    move_map[move].each { swap ->
        newPos[swap[0]] = currentPos[swap[1]]
    }
    return newPos
}
```


MacBook Pro M1
MacBook Air
Raspberry PI
Windows
Lenovo
(Ubuntu Linux)
The Working Environment
Multiple SSH shells for the server nodes
Visual Studio Code for the client
0:10.80

Some Lessons – The “Goldilocks Compute Principle”

Not quite Amdahl’s law..

Dividing up tasks between very different resources needs special consideration, as results may not be what was expected.

Consider:

- A forest has 100 trees that need felling (invasive species)
- Mama bear can push down 50 in 1 hour
- Papa bear can push down 50 in 1 hour
- Baby bear can push down 20 in 1 hour
- Goldilocks can push down 1 in 1 hour

Almost counter intuitive..

- Mama bear working alone can push down all 100 trees in 2 hours.

- Divide up all trees between the parent bears, and the whole task will take 1 hour (2 x as fast as Mama Bear)
- If you divide the task up between the three bears equally (33 each), everyone has to wait for baby bear to finish 33 trees, and it takes about 1.5 hours (only 1.5 x as fast as Mama Bear)
- If you divide up the tasks between Goldilocks *and* the three bears (25 each), everyone has to wait for Goldilocks to push down 25 trees and the task will take 25 hours. (12.5 times as slow as Mama Bear)
- Adding slower resources naively, can give a slower result than having a single, fast resource (that doesn't even include overhead)

Aha.. Ignite does Load balancing

- Round Robin
  - distributed evenly across nodes
  - default
- Weighted Random
  - Distributed randomly but nodes can be assigned a weight
  - Nodes with a bigger weight get proportionally more tasks assigned
- Jobstealing
  - Distributed evenly across nodes
  - Nodes which are freer can “steal” jobs from nodes which have full queues*

* Mama bear saying to Goldilocks “Let me help you with those extra trees”

With Jobstealing and Weighted Random, time taken should tend towards (50+50+20+1)/100, but overhead and different task sizes (some of the moves are loops, and optimized for redundancy) makes this theoretical.

Jobstealing has a little more overhead, but Weighted Random needs pre-knowledge about resources

Some Results

Cluster	Nodes	Average Time	Chart
MacBook Pro only	1	70 secs	
MacBook Air only	1	155 secs	
Lenovo with Linux only	1	238 secs	
Beelink Windows only	1	430 secs	
Raspberry Pi Only	1	583 secs	
Two Macs only	2	55 secs*	
Non - Macs only	3	171 secs	
All – Random Weighted	5	60 secs*	

* surprising!!

Conclusions and work for the future

A possibly unrealistic use case, but very interesting all the same.

This is a great environment for testing and playing with Ignite clusters and Groovy.

Peer class loading working with the Groovy classloader would really mean that we could just pass Groovy closures as tasks

Failing that, some libraries for easy passing of Groovy tasks (as I have done here) would be a worthwhile investment

Compute tasks are very useful even without data, but there is a world of data querying to be explored.

Ignite's high performance queries handle the bottleneck of data querying performance, so a scripting language is quite feasible in a high performance environment

Reach Out

Check out these useful resources:

- Free Apache Ignite Training
- Free Apache Ignite Online Learning at GridGain University
- Ignite Summit – Virtual Community Conference
- Product Demo: The GridGain Unified Real-Time Data Platform
- Apache Ignite Community
- Source code? e-mail me!

E-mail:
jeremy.meyer@gridgain.com