A New Approach To Fizzbuzz

1. Introduction

- Agent-based modelling ☑
- Parallel computation ☑
- Scheduling algorithms ☑
- Microservices ☑

1.1 Claim

- Revolutionary breakthrough in fizzbuzz technique

2. Traditional Fizzbuzz

- Print numbers from 0 to 100
- Multiples of 3: print `fizz` instead
- Multiples of 5: print `buzz` instead
- Multiples of 3 and 5: print `fizzbuzz` instead

2. Traditional Fizzbuzz approach

```
package com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl;

import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.parameters.DefaultFizzBuzzUpperLimitParameter;
import com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.interfaces.FizzBuzz;
import com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.interfaces.parameters.FizzBuzzUpperLimitParameter;

public final class Main {

        public static void main(final String[] args) {
                final ApplicationContext context = new ClassPathXmlApplicationContext(Constants.SPRING_XML);
                final FizzBuzz myFizzBuzz = (FizzBuzz) context.getBean(Constants.STANDARD_FIZZ_BUZZ);
                final FizzBuzzUpperLimitParameter fizzBuzzUpperLimit = new DefaultFizzBuzzUpperLimitParameter();
                myFizzBuzz.fizzBuzz(fizzBuzzUpperLimit.obtainUpperLimitValue());

                ((ConfigurableApplicationContext) context).close();

        }

}
```

Figure 0: The industry-standard approach to Fizzbuzz.

2.1 Proof

Agents = { Counter, Fizzer, Buzzer, Fizzbuzzer }

2.1 Proof (code)

- Fearless concurrency

```
use std::thread;
use std::time::Duration;

fn main() {
    let counter = thread::spawn(|| {
        let mut i = 0u64;
        loop {
            i = i + 1;
            print!("\n{}", i);
            thread::sleep(Duration::from_secs(1));
        }
    });

    let fizzbuzz = thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_secs(15));
            print!("\rfizzbuzz");
        }
    });

    let buzz = thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_secs(5));
            print!("\rbuzz");
        }
    });

    let fizz = thread::spawn(|| {
        loop {
            thread::sleep(Duration::from_secs(3));
            print!("\rfizz");
        }
    });

    let _ = counter.join();
    let _ = fizzbuzz.join();
    let _ = buzz.join();
    let _ = fizz.join();
}
```

Figure 1: A memory-safe and concurrent program.

2.1 Proof (code)

```
➜  code git:(master) ✗ cargo run --release
   Compiling code v0.1.0
    Finished release [optimized] target(s) in 0.82 secs
     Running `target/release/code`

1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
```

Figure 2: Output from the program in Fig. 1.

3. Prior art

- This approach has been applied to other algorithms

3.1 “Genius sorting algorithm”

```
#!/bin/bash

function f() {
    sleep "$1"
    echo "$1"
}
while [ -n "$1" ]
do
    f "$1" &
    shift
done
wait
```

Figure 3: Original “Genius sorting algorithm”.

3.2 Human sleepsort

- Sleepsort proof

Q.E.D.

■

Funding for future work

- Integrate *Fluxcapacitor*[1] to optimise real execution time.

```
$ time ./fluxcapacitor -- sleep 12
real    0m0.057s
```

1. https://github.com/majek/fluxcapacitor