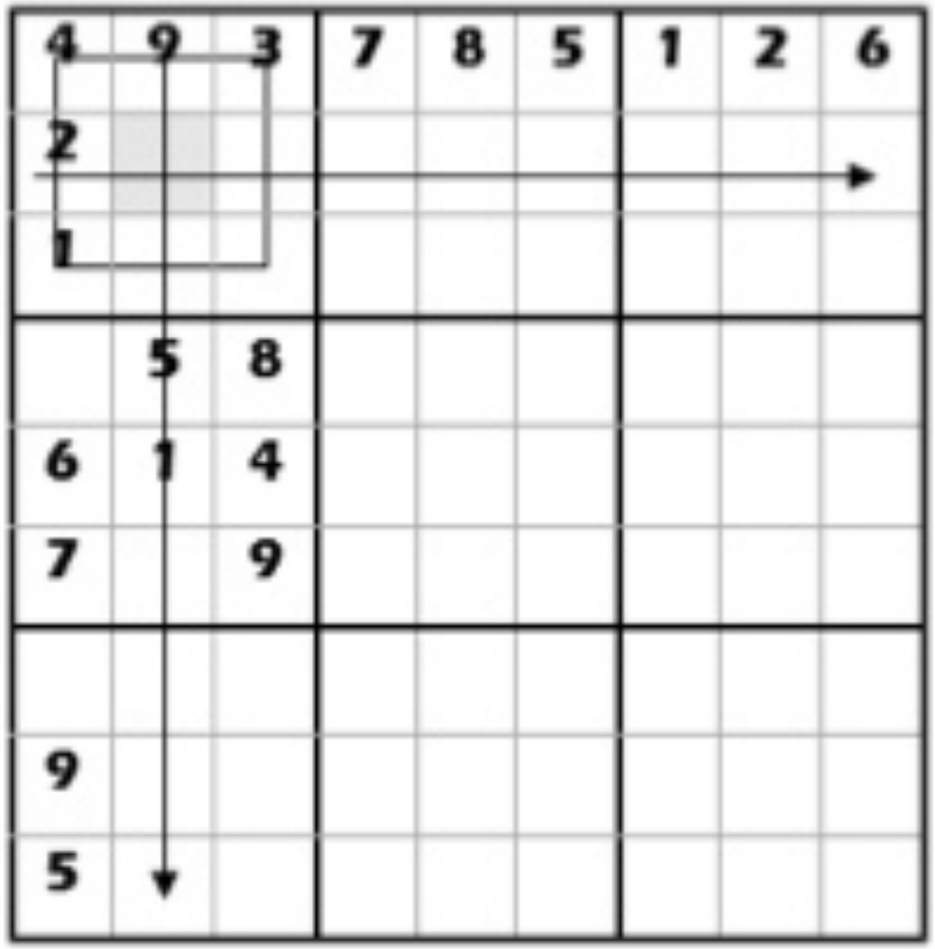
Functional meets parallel

Solving Sudoku puzzles

Maris Orbidans

@marutks

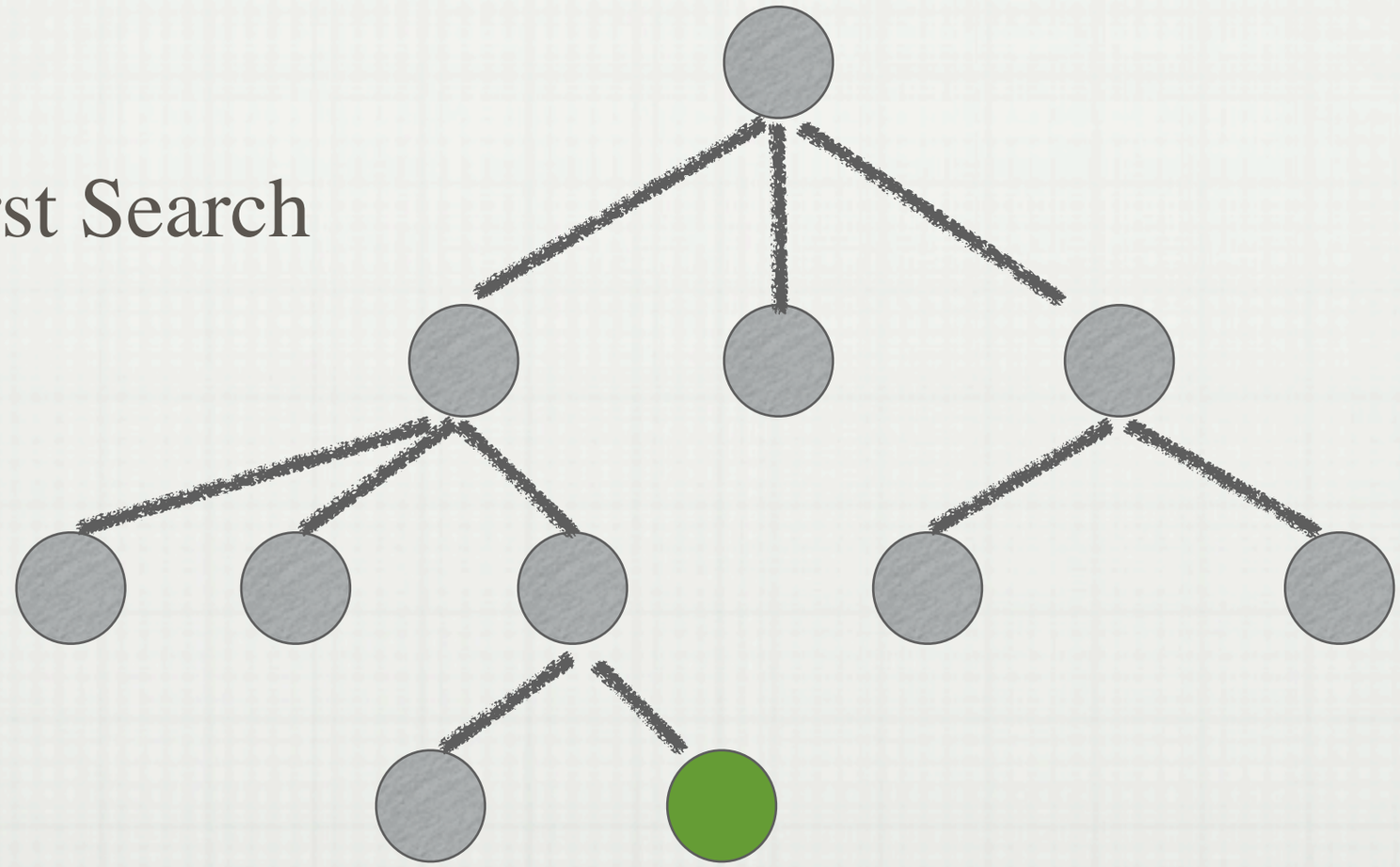
ELIMINATION TECHNIQUE



Sudoku is "a denial of service attack on human intellect"
computer security expert ***Ben Laurie***

RECURSIVE BACKTRACKING

Depth First Search



Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

Why functional ?

- Immutable collections.
- Laziness.
- Higher order functions.
- Result = less code (40 vs 600 LOC)

Another interesting effect of immutable, persistent data objects is that it is easy to maintain previous versions and roll back through them as necessary. This makes it extremely easy and efficient to implement things like undo histories or backtracking algorithms.

Practical Clojure

CLOJURE

- Modern dialect of LISP.
- Runs on JVM.
- Dynamically typed.

Protege: What is this seething mass of parentheses?!

Master: Your father's Lisp REPL. This is the language of a true hacker. Not as clumsy or random as C++; a language for a more civilized age.

```clojure
(ns sudoku.core
  (:use clojure.set)
  (:use [clojure.contrib.seq :only [find-first]]))

(defn select-row [vek i]
  (let [start (- i (rem i 9))
        end (+ start 9)]
    (set (for [e (range start end)](vek e)))))

(defn select-column [vek i]
  (let [start (rem i 9)
        end (+ start 73)]
    (set (for [e (range start end 9)](vek e)))))

(defn select-square [vek i]
  (let [x (- (rem i 9)(rem (rem i 9) 3))
        y (- (quot i 9) (rem (quot i 9) 3))
        spos (+ x (* y 9))
        r (range spos (+ spos 3))
        rng (union (set r) (set (map #(+ % 9) r)) (set (map #(+ % 18) r)))]
    (set (for [e rng](vek e)))))

(defn smallest-change-set [vek]
  (let [sets (for [i (range (count vek)) :when (zero? (vek i))]
               [i (difference (set (range 1 10)) (select-row vek i) (select-column vek i) (select-square vek i))])]
    (if (seq sets)
      (reduce #(if (< (count (%1 1)) (count (%2 1))) %1 %2) sets))))

(defn solve [vek]
  (let [s (smallest-change-set vek)]
    (if (nil? s)
      vek
      (if (seq (s 1))
         (let [sol (for [i (sort (s 1))] (solve (assoc vek (first s) i))) ]
          (find-first #(not (nil? %)) sol))))))
```

```clojure
(defn smallest-change-set [vek]
  (let [sets (for [i (range (count vek)) :when (zero? (vek i))]
               [i (difference (set (range 1 10)) (select-row vek i)
                              (select-column vek i) (select-square vek i))])]
    (if (seq sets)
      (reduce #(if (< (count (%1 1)) (count (%2 1))) %1 %2) sets))))
```

```clojure
(defn solve [vek]
  (let [s (smallest-change-set vek)]
    (if (nil? s)
      vek
      (if (seq (s 1))
         (let [sol (for [i (sort (s 1))] (solve (assoc vek (first s) i))) ]
          (find-first #(not (nil? %)) sol))))))
```

List comprehension is lazy

Vector modification doesnt change original

PARALLELISM

- All known uniquely completable puzzles have at least 17 hints.

http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php

batch10.txt

```
0145000700002000000300000006000004028000100000000009002004000000000070100000000031000600000000084000000002000231000000000000600000008006000000830702040000000500000002004706009000000000000008405000002070000000000300000041600970000000300000000500970001040000000000000080000001043000000000002050020000060400081000000000090090004000000000001000000070080500000007000400000002900060000240400170000000300000800300000000040050000000400600008000007000105004000000700602000000503000010000000064000150300002000030008000021000060000300000000000000400700700000050020000000070030000000000001000006000010840000005000060000000300400710000200000650000000900010400000700001000000050002800000430000025000000600000000300190005060000000000000005400000602000000030000405006200000800000900000003000000016004020000007000000003
```

-:---batch10.txt All (11,0) Git-master (Text Spc Fill)

Quit

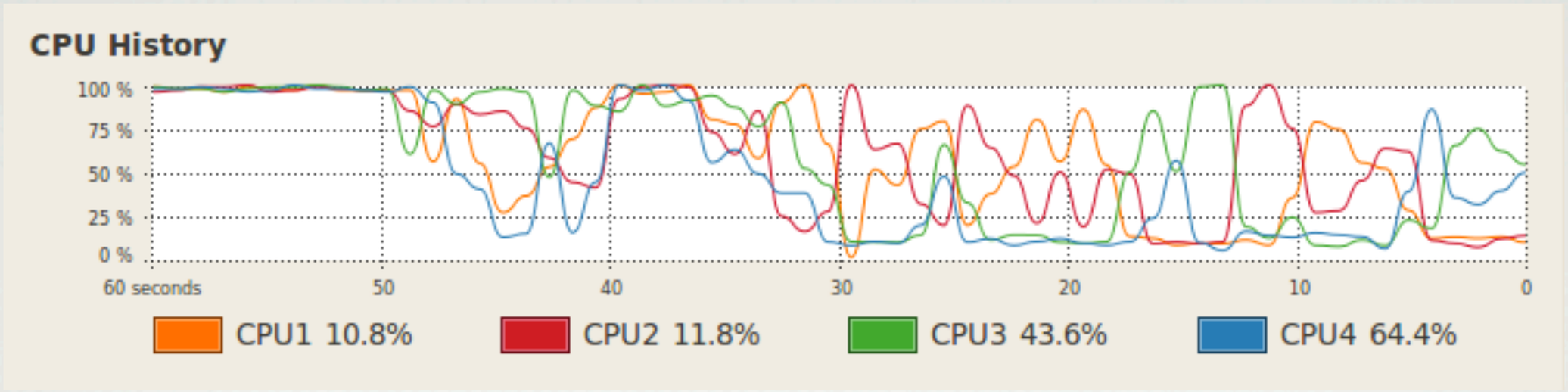
Let’s solve them all !

sequential

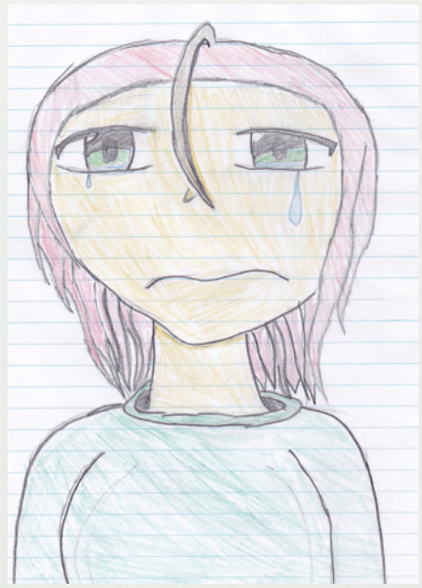
```
(defn -main [& args]
  (let [boards (read-batch-file (first args))
        solutions (map #(solve %) boards)]
    (doseq [solution solutions] (write-file solution))))
```

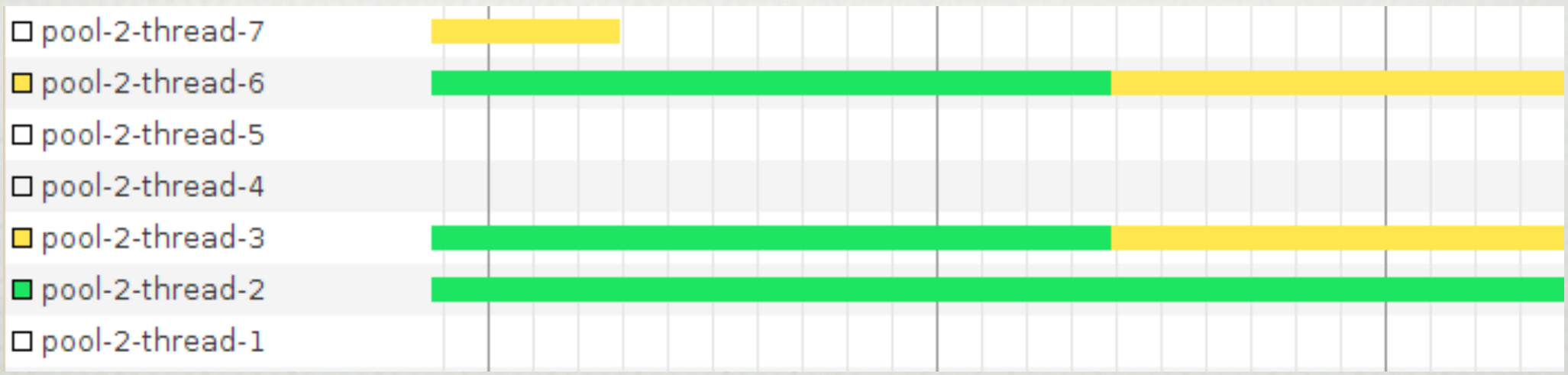
```
(defn -main [& args]
  (let [boards (read-batch-file (first args))
        solutions (pmap #(solve %) boards)]
    (doseq [solution solutions] (write-file solution))
    (shutdown-agents)))
```

parallel



WHY ARE MY CPU CORES IDLE ?





```clojure
(defn pmap
  "Like map, except f is applied in parallel. Semi-lazy in that the
  parallel computation stays ahead of the consumption, but doesn't
  realize the entire result unless required. Only useful for
  computationally intensive functions where the time of f dominates
  the coordination overhead."
  {:added "1.0"}
  ([f coll]
   (let [n (+ 2 (.. Runtime getRuntime availableProcessors))
         rets (map #(future (f %)) coll)
         step (fn step [[x & xs :as vs] fs]
                (lazy-seq
                 (if-let [s (seq fs)]
                   (cons (deref x) (step xs (rest s)))
                   (map deref vs))))]
     (step rets (drop n rets))))
  ([f coll & colls]
   (let [step (fn step [cs]
                (lazy-seq
                 (let [ss (map seq cs)]
                   (when (every? identity ss)
                     (cons (map first ss) (step (map rest ss)))))))]
     (pmap #(apply f %) (step (cons coll colls))))))
```

AGENTS

- Create agent.
  `(def a (agent s))`

- Change state.
  `(send a funct)`
  `(send-off a funct)`

- Read state.
  `(deref a)`
  `@a`

- Wait on agent.
  `(await a)`

Agents provide independent, asynchronous change of individual locations.

slime-repl clojure

```
; SLIME 20100404
user> (use 'sudoku.core 'sudoku.io)
nil
user> (def p (read-file "test/resources/puzzle7.txt"))
#'user/p
user> p
[0 0 0 0 0 0 0 1 0 4 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 5 0 6 0 4
 0 0 8 0 0 0 3 0 0 0 0 1 0 9 0 0 0 0 3 0 0 4 0 0 2 0 0 0 5 0 1 0 0 0 0
 0 0 0 8 0 7 0 0 0]
user> (def a (agent p))
#'user/a
user> (send a solve)
#<Agent@78a40f0e: [0 0 0 0 0 0 0 1 0 4 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0
 0 0 0 5 0 6 0 4 0 0 8 0 0 0 3 0 0 0 0 1 0 9 0 0 0 0 3 0 0 4 0 0 2 0 0
 0 5 0 1 0 0 0 0 0 0 0 0 8 0 7 0 0 0]>
user> (deref a)
[0 0 0 0 0 0 0 1 0 4 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 5 0 6 0 4
 0 0 8 0 0 0 3 0 0 0 0 1 0 9 0 0 0 0 3 0 0 4 0 0 2 0 0 0 5 0 1 0 0 0 0
 0 0 0 8 0 7 0 0 0]
user> (await a)
nil
user> (deref a)
[7 9 3 6 8 4 5 1 2 4 8 6 5 1 2 9 3 7 1 2 5 9 7 3 8 4 6 9 3 2 7 5 1 6 8 4
 5 7 8 2 4 6 3 9 1 6 4 1 3 9 8 7 2 5 3 1 9 4 6 5 2 7 8 8 5 7 1 2 9 4 6 3
 2 6 4 8 3 7 1 5 9]
```

PARALLEL MAP USING AGENTS

```
(defn parmap [f coll] (let [agents (map agent coll)]
                        (doseq [a agents]
                          (send a f))
                        (apply await agents)
                        (map deref agents)))
```

NOT LAZY

```clojure
(ns sudoku.main
  (:use sudoku.core)
  (:use sudoku.io)
  (:gen-class))

(defn parmap [f coll] (let [agents (map agent coll)]
                        (doseq [a agents]
                          (send a f))
                        (apply await agents)
                        (map deref agents)))

(def functions {:s map :p pmap :a parmap})

(defn -main [& args]

  (cond (= (count args) 1) (let [board (read-file (first args))
                                 solution (solve board)]
                             (write-file solution))
        (= (count args) 2) (let [boards (read-batch-file (first args))
                                 funct (functions (keyword (nth args 1)))
                                 solutions (funct #(solve %) boards)]
                             (doseq [solution solutions] (write-file solution))
                             (shutdown-agents))
        :else (println "Usage: SudokuSolver <file name>")))
```

SCALA

```
def solve(p: Vector[Int]): Option[Vector[Int]] = {
  smallestChangeSet(p) match {
    case None => Some(p)
    case Some(set) if set._2.isEmpty => None
    case Some(set) => {
      val sol = for (i <- set._2.toSeq.sorted.view)
        yield solve(p.updated(set._1, i))
      sol.find(_.isDefined).getOrElse(None)
    }
  }
}

def smallestChangeSet(p: Vector[Int]): Option[(Int, Set[Int])] = {
  val sets = for (i <- 0 to p.size - 1; if (p(i) == 0))
    yield (i, (1 to 9).toSet -- (selectRow(p, i) ++
    selectColumn(p, i) ++ selectSquare(p, i)))
  if (sets.isEmpty) None
  else Some(sets.reduceRight((a, b) => if (a._2.size < b._2.size) a else b))
}
```

PARALLEL MAP

```
object Main {
  def main(args: Array[String]): Unit = {

    args match {
      case Array(f) => {
        val p = SudokuIO.readFile(new File(f))
        val sol = SudokuSolver.solve(p)
        SudokuIO.writeFile(System.out, sol)
      }
      case Array(f, a) => {
        val v = SudokuIO.readBatchFile(new File(f))
        val s = if (a == "p") v.par.map(SudokuSolver.solve(_))
        else v.map(SudokuSolver.solve(_))
        s.foreach(SudokuIO.writeFile(System.out, _))
      }
      case _ => println("Usage SudokuSolver <filename>")
    }
  }
}
```

?

```
val v = SudokuIO.readBatchFile(new File(f))
val p = if (a=="p") v.par else v
val s = p.map(SudokuSolver.solve(_))
```

NOT PARALLEL !

ensime-inferior-scala

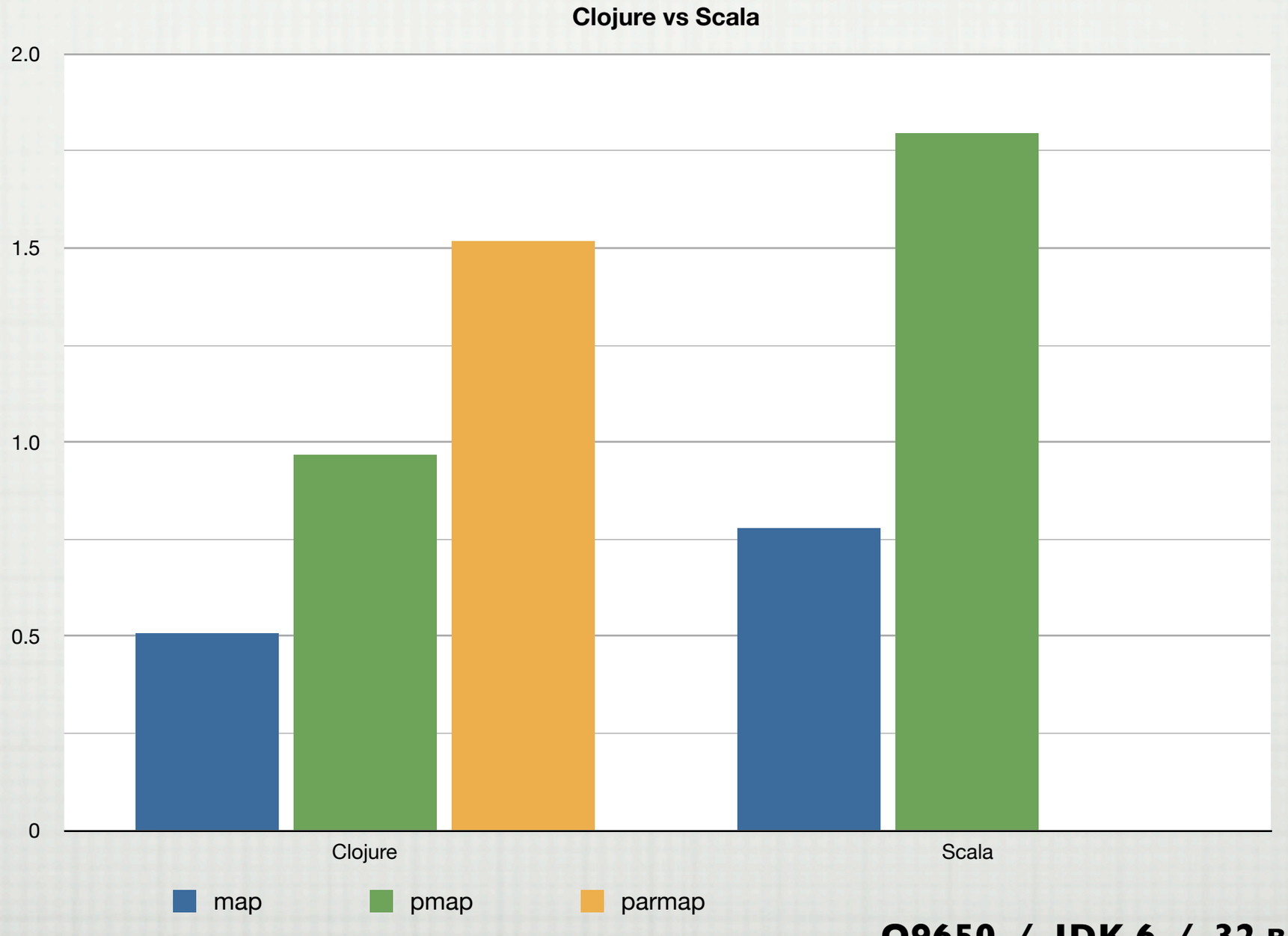
```
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Serv
er VM, Java 1.6.0_26).
Type in expressions to have them evaluated.
Type :help for more information.

scala> (1 to 4).par map { _ => Thread.currentThread.getName }
res0: scala.collection.parallel.immutable.ParSeq[java.lang.String]
 = ParVector(ForkJoinPool-1-worker-1, ForkJoinPool-1-worker-1, For
kJoinPool-1-worker-1, ForkJoinPool-1-worker-1)

scala> val numbers: scala.collection.GenSeq[Int] = (1 to 4).par
numbers: scala.collection.GenSeq[Int] = ParRange(1, 2, 3, 4)

scala> numbers map { _ => Thread.currentThread.getName }
res1: scala.collection.GenSeq[java.lang.String] = ParVector(Thread
-8, Thread-8, Thread-8, Thread-8)

scala>
```

NOT PARALLEL !

-:**-*ensime-inferior-scala* All (14,8) (ENSIME Inferior Scala:run)

Font set for ensime-inf-mode-default face.

PERFORMANCE



Q9650 / JDK 6 / 32 bit Linux

SUMMARY

- Clojure's pmap works better with equal load function.
- Not a problem for non-lazy parallel maps.
- Fork/join enabled map (`pvmap/pvreduce`).

http://blip.tv/clojure/david-liebke-from-concurrency-to-parallelism-4663526

- Always profile your program!

```
git://github.com/maruks/sudoku-clojure.git

git://github.com/maruks/sudoku-scala.git
```