Experiments in Using Google's Go Language for Optimization Research

John W. Chinneck
Systems and Computer Engineering
Carleton University, Ottawa, Canada
Motivation

- Challenges for optimization algorithms:
  - *Always*: faster solutions for bigger problems
  - *New*: massive scale up to handle big data

- Hardware has evolved:
  - *Multiple processors are everywhere*
  - Even *phones* have quad core processors!
  - Recent purchase: 16-core machine for $2000

- Conclusion:
  - *New optimization algorithms must be parallel*
  - Must handle big data problems
  - Must take advantage of parallel hardware
Language Selection Criteria

- **Shortest distance between idea and implementation**
  - I’m an algorithms guy, not a programming specialist
  - Easy to learn and program
  - Parallelism built-in and easy to use

- **Fast execution**
  - Needed for comparisons to commercial solvers
  - Compiled language execution speed

- **Nice to have:**
  - Multi-platform (Windows, linux, Apple)
  - Fast compilation
  - Integrated Development Environment (IDE)
  - Low cost / free
  - Active user community (especially optimizers)
Go Language Design Criteria

- Language specification simple enough to hold in a programmer's head.
- Built-in concurrency
- Others
  - Automatic garbage collection
  - Fast compilation and execution
  - Simple system for dependencies
    - I hate header files
Helpful features in Go

- **Simplicity**
  - No header files!
  - Simple scoping. E.g. externally visible package-level variable: just capitalize the first letter
  - No type inheritance
  - No method or operator overloading
  - No circular dependencies among packages
  - No pointer arithmetic

- **Very strict compiler** prevents common errors
  - No mixed-type arithmetic: you must explicitly cast types.

- **Enforced efficiency**
  - Unused variables are an *error*

- **Enforced common format**
  - Just run `gofmt`: takes care of indenting etc. in a standard way

- **Call C code directly**
  - Use `cgo` or `gccgo`

- **Debugger**
Example optimization application

- **Goal:** quickly find a feasible solution for a system of linear equalities and inequalities
- **Concurrent Constraint Consensus (CC) projection**
  - Very fast initial movement towards feasibility, but bogs down later
- **Main idea:**
  - Define an initial *launch box* for random sampling
  - Repeat:
    - Randomly select multiple CC start points by Latin Hypercube sampling in the launch box
    - Multiple parallel CC runs for limited number of iterations
    - Update incumbent (point closest to feasibility)
    - Reinitialize a smaller launch box centred around incumbent
- **Semi-successful...**
package solver
// Controls the solution process

import (  
    "fmt"  
    "lp"  
    "math"  
    "math/rand"  
    "sort"  
    "strconv"  
    "time"
)

// Package global variables  
var PrintLevel int  // controls the level of printing. Setting it equal to zero turns printing off  
var FinalBox int // Captures the last box commenced so it can be printed out

// Structures needed for sorting the impact list  
type IMPACT struct {  
    Row int  
    Sum int
}

func Solve(AlphaIn float64, BetaIn float64, MaxItnsIn int, MaxSwarmPtsIn int, plinfyIn float64, ...  
...  

External Reference to a Package Variable:

solver.PrintLevel = 0 // PrintLevel = 0 turns off the printing so you can run through a set of files

External Reference to a package routine:

Point, Status = solver.Solve(Alpha, Beta, MaxItns, MaxSwarmPts, plinfy, featol)
Language Elements

• Statements:
  ◦ Only one kind of loop: *for*
    • Index over a range, or over the length of a vector
    • Can act like a while loop
  ◦ If-then-else
  ◦ Select / Case
  ◦ Etc.

• General data structures

• Arrays and “slices” (vectors)

• *Generally simple and intuitive*
Functions

// Given an input point at which some of the variables may violate their bounds, this
// routine returns an output point in which all of the variables have been reset onto their
// closest bound, if necessary.

func EnforceBounds(PtIn []float64) (PtOut []float64) {
    PtOut = make([]float64, len(PtIn))
    for j := 0; j < lp.NumCols; j++ {
        if PtIn[j] < lp.LP.Cols[j].BndLo {
            PtOut[j] = lp.LP.Cols[j].BndLo
            continue
        }
        if PtIn[j] > lp.LP.Cols[j].BndUp {
            PtOut[j] = lp.LP.Cols[j].BndUp
            continue
        }
        PtOut[j] = PtIn[j]
    }
    return
}
Concurrency

- Make any routine concurrent by the `go` keyword
  - Spawns a new asynchronous thread
- Communication is via `channels`
  - Channels have defined types
    - Could be a structure holding many items
  - Return results via channels
- Channels allow:
  - Blocking to wait for something to be received
  - Receive something from one of several channels
  - Etc.
- There is also a `sync` package
  - Mutex, lock, wait, etc.
Concurrent example

```go
NumCPUs := runtime.NumCPU()
...
MaxPts := 2 * NumCPUs
...
chPoint := make(chan []float64)
...

for itn := 0; itn < MaxItns; itn++ {
    // Get new set of CC start points
    NewPoints(itn)

    // Run CC in parallel to improve each start point
    for i := 0; i < MaxPts; i++ {
        go CC(Point[i], chPoint, i)
    }

    // Retrieve the CC output points
    for i := 0; i < MaxPts; i++ {
        Point[i] = <-chPoint
    }
}
} // end of large iteration loop
```
Concurrency: observations

- Even identical processes may return results in a different order than they were instantiated!
  - Interruptions from other processes, etc.
- Go takes care of everything
  - You can have many simultaneous threads
Packages

- Many built-in, see http://golang.org/pkg/
  - E.g. sorting, database, etc.
- External projects:
  - https://code.google.com/p/go-wiki/wiki/Projects
  - E.g. Mathematics, machine learning
  - CVX (ported from the CVX python package)
  - A particle swarm optimizer
  - Linear algebra routines, e.g. BLAS
  - Graph theory algorithms
Learning Go is easy

- Start at the tour of Go: http://tour.golang.org/#1
- Go documentation: http://golang.org/doc/ includes video tours, docs, examples
- The Go playground: http://play.golang.org/
- Go home: http://golang.org/
- Searching online for Go information: search on “golang”
IDEs for Go

- See [http://geekmonkey.org/articles/20-comparison-of-ides-for-google-go](http://geekmonkey.org/articles/20-comparison-of-ides-for-google-go)
- I like Eclipse (called Goclipse): [https://code.google.com/p/gocclipse/](https://code.google.com/p/gocclipse/)
Golang for Optimization
Things that tripped me up

- Copying a vector (called a slice in Go)
  - `copy(BestPt, CCPoint)`

- Concurrency
  - Concurrent processes usually return (or write on the channel) in a different order than they were launched
Conclusions

- Easy to learn
  - Mostly intuitive
  - Good online learning, reference, and practice tools
- Concurrency easy to program
  - Takes some practice if new to concurrency
- Very fast compilation, fast execution
- Multi-platform (Windows, linux, Apple)
- Good IDEs
- Free
- But relatively little supporting software for optimization (yet)
- Bottom line:
  - Good language for general coding of parallel algorithms for optimization
    - Supported by Google, so likely to be around for a while
- Potential alternative: Julia
Julia


- Fast

- Concurrency built-in
  - More complicated to use?

- Larger optimization user community
  - Built-in matrix routines (Matlab-like)
  - Many optimization interfaces already:
    - [https://jump.readthedocs.org/en/release-0.4/jump.html](https://jump.readthedocs.org/en/release-0.4/jump.html)