Estudo Experimental sobre Paralelismo na Linguagem Go usando *Goroutines*

Luiz Alexandre de S. Freitas[1], Fernando Barbosa Matos[1], Paulo Eduardo Nogueira[2]

[1]Núcleo de Computação – Instituto Federal Goiano
Morrinhos – GO – Brasil

[2]Instituto Federal São Paulo
Hortolândia – SP – Brasil

luiz.alexandre@live.com, fernando.matos@ifgoiano.edu.br, paulocoronato@gmail.com

Abstract. *The best use of hardware and the solution of complex computational problem, requires a proposal of efficient parallel processing. The Go programming language implements several resources in order to streamline the writing and execution of programs. In this work, it is showed an experimental study on the performance of the parallelism implemented by the Go language through goroutines. In addition, it was also observed, the behavior of a genetic algorithm, when solving the problem of the traveling salesman, when implemented in Go language.*

Resumo. *O melhor aproveitamento do hardware e a solução de problemas computacionais complexos, requer uma proposta de processamento paralelo eficiente. A linguagem de programação Go implementa diversos recursos com o intuito de dinamizar a escrita e a execução dos programas. Neste trabalho, é apresentado um estudo experimental sobre o desempenho do paralelismo implementado pela linguagem Go através das goroutines. Além disso, foi observado também, o comportamento de um algoritmo genético, ao resolver o problema do caixeiro viajante, quando implementado em linguagem Go.*

1. Introdução

O avanço tecnológico para o desenvolvimento das linguagens de programação atuais visa otimizar o aproveitamento do hardware, principalmente, no que se refere a capacidade de processamento paralelo nativo. O uso de múltiplos paradigmas em uma mesma linguagem de programação é uma tendência nas linguagens modernas, como o Go, que implementa nativamente recursos e características predominantemente de linguagens do paradigma imperativo (linguagem C), Orientação a Objetos (linguagem JAVA) e funcional (Linguagem LISP) [Lisp 2017]. Essa arquitetura da linguagem proporciona ao programador explorar de diversas formas os recursos mais avançados de cada paradigma de forma a proporcionar uma maior facilidade na escrita do código e, ao mesmo tempo, a geração mais eficiente do mesmo.

Desenvolvida pela empresa Google, a linguagem Go foi criada por Robert Griesemer, Rob Pike e Ken Thompson como uma linguagem compilada, de uso geral, focada na produtividade e programação concorrente [Schamager et al. 2010].

O estudo do desempenho de linguagens de programação é de suma importância para a compreensão do comportamento de uma linguagem mediante a uma dada situação de processamento. Em seu trabalho, [Pompeu et al. 2015] comparou o desempenho de programas escritos em JAVA, PHP, C++, NODEJS, RUBY, PYTHON e GO, quando estes implementam API REST e JSON.

Neste trabalho, é apresentado um estudo experimental sobre o desempenho do paralelismo implementado pela linguagem Go através das *goroutines* [Aimonetti 2017]. Isso por que essa linguagem foi projetada tendo em mente o processamento massivo em paralelo, e por tanto observar seu comportamento frente a um cenário de execução sequencial é importante para avaliar até que ponto o paralelismo nativo dá uma vantagem a linguagem no desenvolvimento de um programa.

Para este estudo, foi escrito um programa que procura resolver o problema do caixeiro viajante, utilizando conceitos de algoritmos genético. Isso por que é um problema facilmente paralelizável e é fácil observar o comportamento da linguagem na execução do algoritmo. O restante do artigo está estruturado do seguinte modo, a Seção 2 apresenta uma breve descrição sobre a linguagem Go, a Seção 3 descreve o planejamento do experimento realizado, a Seção 4 discute os resultados encontrados, e a Seção 5 as conclusões e considerações finais.

2. Linguagem Go

A linguagem de programação Go é uma linguagem de propósito geral, compilada, concorrente, com *garbage-collected*, com tipagem forte e estática [Pike 2012b], foi desenvolvida e lançada pela Google em novembro de 2009. A linguagem Go foi concebida para resolver problemas de eficiência na criação, compilação, execução e escalabilidade de software da Google. Um dos principais pontos fortes da linguagem é sua abordagem em relação a concorrência e ao paralelismo, baseado no trabalho *Communicating Sequential Processes* [Filipini 2014].

Uma proposta de implementação de programas concorrentes em Go é através de uma abstração eficiente e leve de *thread*, conhecida como *goroutine* [Aimonetti 2017]. As *goroutines* executam no mesmo espaço de endereçamento e, portanto, é necessário que haja sincronização entre elas para o acesso a memória compartilhada. Essa sincronização é feita através de *channels*, os quais são usados para enviar e receber valores entre *goroutines* [Aimonetti 2017] dispensando o uso de travas, semáforos e outras técnicas de sincronização de processos. Isso ocorre porque o próprio ambiente de execução garante que um *channel* será acessado apenas por uma *goroutine* em determinado momento. Um ponto que vale ressaltar é que concorrência não é paralelismo [Pike 2012a], embora um programa concorrente bem escrito em Go pode executar de forma eficiente em paralelo em poucos passos.

Há diversas ferramentas integradas ao ambiente que auxiliam no desenvolvimento de softwares na linguagem Go, um bom exemplo são as ferramentas para testes automatizados. Em relação a concorrência, uma das ferramentas que se destaca é a *race-detector* que auxilia na detecção de *Race Conditions*. *Race Condition* é uma condição em que duas ou mais rotinas requisitam acesso ao mesmo recurso, como uma variável ou estrutura de dados, esse tipo de condição é comum em programas concorrentes e é um erro difícil de se detectar [Vyukov and Gerrand 2017].

3. Planejamento Experimental

3.1 Método

Neste estudo, tanto o planejamento e execução do experimento, quanto a análise dos resultados, foram realizados conforme o método estatístico DOE (*Design of Experiment*) [Montgomery 2000]. O DOE instrui a modificar de forma controlada os fatores sendo estudados, permitindo assim, observar os efeitos dessas alterações sobre a variável resposta. Para este estudo, foi definido o tempo de execução do programa TESTE, obtido em cada cenário de teste (tratamento), como a variável resposta. Além disso, os tratamentos foram definidos através do método da matriz de sinais [Montgomery 2000], sendo que os sinais (+ e -) representam os possíveis níveis que um fator pode assumir. A matriz de sinais foi configurada de acordo com a ordem de Yates [Jain 1991]. Em relação aos tratamentos, é importante informar que é obtido pela combinação de fatores e níveis [Montgomery 2000]. Como uma forma de evitar que a execução e um tratamento influencie nos resultados do próximo tratamento, o sistema operacional foi reinicializado a cada novo tratamento.

Ao final da execução de cada tratamento, os tempos obtidos são comparados. Se houver diferença estatística significativa, nos tempos de execução, isso significa que há diferença no desempenho do programa TESTE. Os tempos de execução do experimento foram obtidos através do programa *time* [Kerrisk 2010].

A análise estatística foi realizada seguindo os passos sugeridos pelo protocolo proposto por Nogueira and Matias (2015). Esse protocolo propõe uma sequência de análise estatística de modo a garantir que os resultados obtidos são confiáveis e exatos segundo um nível de confiança pré-estabelecido. Em um primeiro momento, o protocolo sugere avaliar se as amostras obtidas no experimento seguem uma distribuição Gaussiana, caso isso ocorra, o protocolo sugere o uso de um *teste-t* de hipótese [Barbetta et al 2010]. Por outro lado, se as amostras não seguirem uma distribuição Gaussiana, o protocolo sugere o uso do teste de *Wilcoxon-Mann-Whitney* [Barbetta et al 2010], o qual exige apenas que as amostras apresentem a mesma distribuição de dados. Após a aplicação de um dos testes é possível identificar se as amostras obtidas, no experimento, possuem ou não uma diferença estatística significante entre elas.

3.2 Programa TESTE

Para realização do experimento, foi desenvolvido o programa TESTE que utiliza uma heurística para calcular a distância entre as cidades. Foram elaboradas duas versões, uma sequencial e outra paralela, implementadas na linguagem Go. Tendo em vista que o foco desse trabalho é avaliar o comportamento da linguagem Go quando utiliza os recursos de paralelismo nativo. Para a execução paralela do programa, configurou-se a variável GOMAXPROCS, com o intuito de indicar a quantidade máxima de núcleos que o *runtime* [Runtime 2017] poderá utilizar na execução do programa.

Com o objetivo de gerar carga de processamento para a realização do experimento, ambas as versões do programa TESTE foram implementadas com um algoritmo genético, com os mesmos passos e operações, para calcular uma possível rota entre 123 e 246 cidades goianas, sem repetições e voltando para a cidade de partida. Sendo que a única diferença entre as versões é o fato da versão paralela ter sido implementada

utilizando *goroutines* e *channels* [Filipini 2014]. Os parâmetros referentes a população, função *fitnnes,* cruzamento e mutação são idênticas em ambos os casos.

O indivíduo utilizado no algoritmo genético é composto pelo seu valor de aptidão (*fitness)* que é a medida de sucesso reprodutivo do indivíduo e por um *array* de *structs* que contém a identificação, a latitude e a longitude da cidade, a sequência que o *array* está disposto é a mesma que será percorrida a rota. O cálculo do *fitness* do indivíduo foi obtido através da soma da distância entre todas as coordenadas na sequência que está disposta no indivíduo. A seleção dos melhores indivíduos para a próxima geração foi realizada por meio de *elitismo,* estratégia esta que seleciona os melhores indivíduos de uma população e garante que a qualidade da solução obtida pelo algoritmo genético não diminuirá na geração seguinte. Para a recombinação genética (*crossover*) foi utilizada a técnica *order crossover operator* (OX) que é uma das técnicas existentes para a recombinação para cromossomos ordenados [Sehrawat and Singh 2011].

As cidades goianas foram separadas em dois arquivos contendo 123 e 246 cidades, respectivamente. Cada linha dos arquivos corresponde a uma cidade e, cada cidade possui uma identificação e suas respectivas latitude e longitude, convertidas em decimal. Como entrada para a execução dos experimentos foi passado os seguintes argumentos: 100 indivíduos por população, 1000 gerações e 0,1 de taxa de mutação. Esses valores foram definidos como padrão para comparar o desempenho entre a execução sequencial e a paralela, sendo que para a obtenção da menor distância é necessário a variação destes valores.

3.3 Experimento

O objetivo desse experimento foi avaliar o desempenho de um programa escrito em Go [Aimonetti 2017], compilado de forma sequencial e de forma paralela. A Tabela 1 sumariza os fatores e níveis utilizados no experimento.

Tabela 1. Fatores e níveis avaliados

		Nível(-)	**Nível(+)**
Fatores	*Cidades* (C)	246	123
	Paralelismo (P)	Sequencial	Paralelo

No nível (-) o fator *Cidades* assume o valor 246, o que indica que o programa deverá resolver o problema para o cálculo de distância entre as 246 cidades; o nível (+) configura este fator para 123 cidades, reduzindo assim, a quantidade de trajetos que o programa deverá calcular. Ao variar esse fator, busca-se aumentar e diminuir a quantidade de trajetos que o programa deverá calcular. Em relação ao fator *Paralelismo*, busca-se observar se há diferença estatística significante entre o programa sequencial e o programa paralelo, compilado com Go. Assim, no nível (-), o programa foi compilado de forma sequencial, sem o uso de *goroutines* e, no nível (+), o programa foi compilado de forma paralela, utilizando *goroutines*. Neste estudo, a variável de ambiente GOMAXPROCS foi configurada, de modo, a permitir o uso máximo de processadores disponíveis e, o programa TESTE foi compilado de forma paralela sem especificar o número de threads a *priori*, assim, o próprio Go define quantas *threads* usar.

Todos os tratamentos deste experimento foram replicados 31 vezes, assim, procurou-se reduzir a possibilidade de ocorrência de um erro experimental. Além disso, a primeira replicação foi descartada, deste modo, evitou-se que a análise fosse realizada com ruído devido ao carregamento dos dados do programa na *cache* e sem maior influência do *buffer* de disco.

O experimento foi realizado em um computador com processador Intel I5 3570 Quad-core de 1,60 GHz, com três níveis de cache e 8 GB de memória RAM (ver Figura 1). O sistema operacional adotado foi o openSUSE Leap 42.2 kernel 4.4.27-2. A versão da linguagem Go utilizada foi a go1.8.linux-amd64.

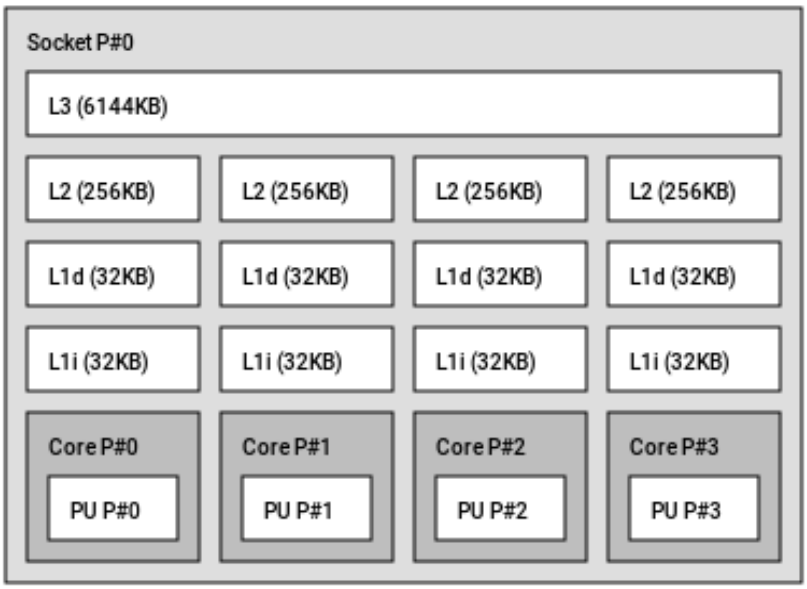

Figura 1. Topologia do processador utilizado no Experimento

4. Resultados

Após a coleta dos tempos de execução, foi realizada uma análise descritiva dos dados. Ao comparar os tempos de execução médios e medianos dos tratamentos, observou-se que a execução paralela obteve tempos menores do que a execução sequencial. Além disso, é possível identificar que a dispersão dos dados é menor na execução paralela do que na execução sequencial. A Tabela 2 apresenta a compilação dos dados obtidos. As colunas Seq123 e Seq246, indicam os tratamentos sequenciais com 123 e 246 cidades respectivamente e, as colunas Par123 e Par246, indicam os tratamentos paralelos com 123 e 246 cidades, respectivamente.

Tabela 2. Estatística descritiva obtida no Experimento

	Seq123	**Seq246**	**Par123**	**Par246**
Média	5,147s	8,100s	2,219s	3,456s
Mediana	5,150s	8,095s	2,220s	3,460s
Desvio Padrão	0,033s	0,053s	0,004s	0,007s
Variância	0,001	0,003	0,000	0,000
Maior Tempo	5,210s	8,200s	2,230s	3,470s
Menor Tempo	5,080s	7,980s	2,210s	3,440s

s = segundos

Com os dados obtidos foi possível analisar se houve uma diferença estatisticamente significante, entre os tempos obtidos em cada tratamento. Ao aplicar o protocolo, proposto por Nogueira and Matias (2015), observou-se que com um nível de confiança de 95%, apenas os tratamentos Seq123 e Seq246 seguem uma distribuição Gaussiana e, além disso, confirmou-se que há uma diferença estatisticamente significante quando, esses dois tratamentos, foram comparados com os tratamentos Par123 e Par246.

Ao comparar o tratamento Par123 com Par246 foi observado que não há uma diferença estatisticamente significante entre as amostras, porém esse resultado pode não estar correto para o nível de confiança de 95%. Isso porque o teste estatístico, indicado no protocolo [Nogueira 2015], tem como premissa o fato das duas amostras seguirem uma mesma distribuição de dados, o que não foi observado quando o teste de comparação de distribuição foi realizado.

Uma outra informação interessante, obtida ao analisar os dados, foi o *speedup* [Lilja 2005] obtido quando se faz uma comparação entre os tratamentos com o mesmo número de cidades, mas com o tipo de compilação diferente. Ao calcular o *speedup* da mediana, entre os tratamentos Seq123 e Par123, observou-se uma melhora de 2,32 e quando foi analisado os tratamentos Seq246 e Par246, identificou-se que o *speedup* foi de 2,34. Esses resultados indicam que um programa em linguagem Go, compilado de forma paralela, apresenta um desempenho melhor do que um programa compilado de forma sequencial.

Em relação ao cálculo das rotas realizado pelo algoritmo proposto, foi identificado que o programa TESTE, compilado de forma sequencial, apresentou valores melhores do que quando esse mesmo programa foi compilado de forma paralela. A Tabela 3 apresenta os resultados obtidos para o cálculo de menor distância de acordo com os parâmetros pré-estabelecidos para o experimento (100 indivíduos por população, 1000 gerações e 0,1 de taxa de mutação). As colunas DistSeq123 e DistSeq246, indicam os resultados de distância obtidos nos tratamentos sequenciais com 123 e 246 cidades respectivamente e, as colunas DistPar123 e DistPar246, indicam os tratamentos paralelos com 123 e 246 cidades, respectivamente

Tabela 3. Estatística descritiva para as distâncias obtidas no programa TESTE

	DistSeq123	**DistSeq246**	**DistPar123**	**DistPar246**
Média	9950,95 km	23237,35 km	12690,56 km	27160,84 km
Mediana	9936,94 km	23353,52 km	12883,36 km	27098,59 km
Desvio Padrão	633,19 km	1554,79 km	856,95 km	2014,26 km
Variância	400935,21	2417376,00	734358,61	4057261,00
Maior Distância	11156,34 km	25356,55 km	14457,39 km	32238,19 km
Menor Distância	8676,91 km	19693,13 km	11073,94 km	22788,97 km

km = quilômetro

Uma vez que o programa TESTE é o mesmo e apenas foi compilado como sequencial e paralelo, esperava-se que os resultados para as distâncias fossem próximos. No entanto, ao observar os dados obtidos referentes as distâncias, foi possível identificar que no tratamento com 123 cidades, a melhor rota obtida na execução paralela foi 27,62% maior do que na execução sequencial e 15,72% maior quando se observa os tratamentos com 246 cidades. Ao comparar a média observou-se a mesma diferença, a execução

paralela foi 27,53% maior, no tratamento com 123 cidades e para 16,88% maior no tratamento com 246 cidades.

5. Conclusões

Os instrumentos oferecidos por uma linguagem de programação são de suma importância para dinamizar a escrita dos programas, bem como, alcançar melhor desempenho em sua execução. Quando se trabalha com processamento paralelo busca-se usufruir ao máximo do hardware, por isso, uma linguagem que oferece bons instrumentos para o desenvolvimento dos programas é imprescindível. Neste trabalho, foram apresentadas informações sobre o desempenho da linguagem Go, quando um algoritmo é escrito e executado utilizando *goroutines*. Ao escrever um programa na linguagem Go, utilizando *goroutines*, o programador não precisa se preocupar com a sincronização entre os processos, isso é realizado de forma dinâmica dentro do ambiente de execução da linguagem.

Os resultados aqui apresentados foram obtidos seguindo uma metodologia para eliminar ruídos que pudessem interferir em sua análise. Esses resultados, indicam que utilizar *goroutines*, melhoram o desempenho de execução de um programa. Porém, apesar do programa utilizado para o experimento ter sido o mesmo, foi possível observar uma diferença nos resultados de processamento exibidos por ele. Neste caso, o cálculo de rotas.

Infelizmente, o experimento proposto, foi insuficiente para responder o porquê desse ocorrido, sendo necessário a mudança de outros parâmetros de execução do algoritmo genético, como a quantidade de indivíduos na população para a averiguar se a variabilidade genética da população é afetada quando se utiliza o *goroutines* para paralelizar o código em detrimento do código sequencial. Além disso, é proposto também como trabalho futuro um estudo experimental comparando o desempenho da linguagem Go, utilizando *goroutines*, com o desempenho de outras implementações que empregam o paralelismo, como é o caso da linguagem C com a biblioteca *pthreads* [Buttlar et al 1996] e da OpenMP [Chapman et al 2007] que é uma interface de programação *multithreading* multiplataforma.

Referências

Aimonetti, M. (2017) “Go Bootcamp”, http://www.golangbootcamp.com/book, Março.

Barbetta, P.A, Bornia, A.C e Reis, M. M (2010), Estatística para Cursos de Engenharia e Informática, 3ª edição.

Buttlar, D., Farrell, J. and Nichols, B. (1996), PThreads Programming. 1th edition.

Chapman, B., Jost, G. and Ruud V. (2007), Using OpenMP - Portable Shared Memory Parallel Programming. Scientific and Engineering Computation Series.

Filipini, C. (2014), Programando em Go: Crie aplicações com a linguagem do Google, Casa do Código, 1ª edição.

Jain, R. (1991), The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley, 1st edition.

Kerrisk, M. (2010), The Linux Programming Interface, San Francisco: No Starch Press, 1st edition.

Lilja, D. J. (2005), Measuring Computer Performance: A Practitioner's Guide, Cambridge University Press, 1st edition.

Lisp (2017) "Common Lisp", http://lisp-lang.org/, Março.

Montgomery, D. C. (2000), Design and Analysis of Experiments, John Wiley, 3rd edition.

Nogueira, P. E. and Matias, R. (2015). A quantitative study on execution time variability in computing experiments. In *Proceedings of the 2015 Winter Simulation Conference* (WSC '15), pages 529-540, IEEE Press, Huntington.

Pike, R. (2012a) "Concurrency is not Parallelism", https://talks.golang.org/2012/waza.slide#1, Março.

Pike, R. (2012b) "Go at Google: Language Design in the Service of Software Engineering", https://talks.golang.org/2012/splash.article#TOC_13, Março.

Pompeu, I. F., Matos, F. B. e Melo, M. S. (2015). Testando a performance de tecnologias especialistas em Rest API JSON uma abordagem em JAVA, PHP, C++, NODEJS, RUBY, PYTHON e GO. Em Encontro Anual de Computação (ENACOMP 2015), p. 147-154, Catalão.

Runtime (2017) "The Go Programming Language: Package runtime", https://golang.org/pkg/runtime/, Março.

Schamager, F., Cameron, N. and Noble, J. (2010). GoHotDraw: Evaluating the Go Programming Language with Design Patterns. In Evaluation and Usability o Programming Languages and Tools, p. 10:1-10:6, New York.

Sehrawat, M. and Singh, S. (2011) "Modified Order Crossover (OX) Operator", International Journal on Computer Science & Engineering, Engg Journals Publications, Vol. 3 Issue 05, p. 2019-2023.

Vyukov, D. and Gerrand, A. (2017) "Introducing the Go Race Detector", https://blog.golang.org/race-detector, Março.