UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

RANIERE FERNANDES DE MEDEIROS

UM ESTUDO SOBRE A EFICIÊNCIA DOS COMPILADORES DA LINGUAGEM GO COM O AUXILIO DE ALGORITMOS GENÉTICOS

JOÃO PESSOA
2018

RANIERE FERNANDES DE MEDEIROS

UM ESTUDO SOBRE A EFICIÊNCIA DOS COMPILADORES DA LINGUAGEM GO COM O AUXILIO DE ALGORITMOS GENÉTICOS

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Centro de Informática da Universidade Federal da Paraíba, como requisito parcial para obtenção do grau de Mestre em Informática

Orientador: Prof. Dr.Clauirton de Albuquerque Siebra

JOÃO PESSOA
2018

Catalogação na publicação
Seção de Catalogação e Classificação

M488e Medeiros, Raniere Fernandes de.
Um estudo sobre a eficiência dos compiladores da linguagem Go com o auxilio de algoritmos genéticos / Raniere Fernandes de Medeiros. - João Pessoa, 2019.
72 f.

Orientação: Clauirton de Albuquerque Siebra.
Dissertação (Mestrado) - UFPB/CI.

1. Compilador. 2. Go Compiler. 3. GCC. I. Siebra, Clauirton de Albuquerque. II. Título.

UFPB/BC

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de Raniere Fernandes de Medeiros, candidato ao título de Mestre em Informática na Área de Sistemas de Computação, realizada em 12 de dezembro de 2018.

Aos doze dias do mês de dezembro, do ano de dois mil e dezoito, às dezesseis horas, no Centro de Informática da Universidade Federal da Paraíba, em Mangabeira, reuniram-se os membros da Banca Examinadora constituída para julgar o Trabalho Final do Sr. Raniere Fernandes de Medeiros, vinculado a esta Universidade sob a matrícula nº 20161020434, candidato ao grau de Mestre em Informática, na área de "Sistemas de Computação", na linha de pesquisa "Sinais, Sistemas Digitais e Gráficos", do Programa de Pós-Graduação em Informática, da Universidade Federal da Paraíba. A comissão examinadora foi composta pelos professores: Clauirton de Albuquerque Siebra (PPGI-UFPB) Orientador e Presidente da Banca, Natasha Correia Queiroz Lino (PPGI-UFPB), Examinadora Interna, Daniel Lacet de Faria Fireman (IFAL), Examinador Externo à Instituição. Dando início aos trabalhos, o Presidente da Banca, cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse a exposição oral do trabalho de dissertação intitulado "Um Estudo sobre a Eficiência dos compiladores da Linguagem GO com o Auxilio de Algoritmos Genéticos". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "***aprovado***". Do ocorrido, eu, Alisson Vasconcelos de Brito, Vice-Coordenador do Programa de Pós-Graduação em Informática, lavrei a presente ata que vai assinada por mim e pelos membros da banca examinadora. João Pessoa, 12 de dezembro de 2018.

Prof. Dr. Alisson Vasconcelos de Brito

Prof. Dr. Clauirton de Albuquerque Siebra
Orientador (PPGI-UFPB)

Prof. Dr. Natasha Correia Queiroz Lino
Examinadora Interna (PPGI-UFPB)

Prof. Dr. Daniel Lacet de Faria Fireman
Examinador Externo ao Programa (IFAL)

À Josefa, Rayssa e Clara.

Agradecimentos

Primeiro à Deus por conceder a mim a oportunidade de respirar o ar criativo no aspecto terreno. Gratidão!

À minha Mãe Josefa Fernandes de Medeiros por exemplificar em vida as virtudes de uma autentica alquimista. Sua prole é grato por todos os ensinamentos em atitudes. Gratidão!

À minha Esposa Rayssa Martins por todo amor e suporte em todas as areas da minha vida. Eu não estaria aqui se não fosse pelo seu amor, encorajamento e parceria. Gratidão!

À Clara e Lina pelo simples fato de existirem. Vocês são a minha alegria em todas as manhãs nas quatro estações. Gratidão!

Ao meu irmão Romualdo Fernandes de Medeiros por segurar sozinho muito dos problemas familiares para que eu pudesse voar. Gratidão!

Ao meu orientador Dr. Clauirton de Albuquerque Siebra por transcender na paciência, conhecimento, dedicação, isentivo e confiança. Gratidão!

A todos os incontáveis familiares e amigos que de alguma forma ajudaram na minha caminhada até este momento. Gratidão!

A todas as regalias e privilégios que usufrui direto ou indiretamente por causa da minha espécie, cor, raça e gênero. A minha esperança é contribuir de alguma forma na equiparação de oportunidades independente de espécie, cor, raça e gênero. Punhos fechados!

Resumo

A linguagem Go é uma das linguagens mais novas da atualidade com um crescente aumento de popularidade na indústria de software. Ela é definida por uma especificação e implementada por dois compiladores com propostas diferentes para assegurar que a especificação esteja correta e completa. O compilador gc foca na compilação rápida e poucas otimizações enquanto que o gccgo foca na utilização das otimizações do GCC sem se preocupar com o tempo gasto na compilação. Este trabalho propõe um estudo com objetivo de construir um algoritmo genético que auxilie na identificação de situações em que o gccgo possa ser mais performático que o Go Compiler. Resultados mostram que o Go Compiler é em média 24,7 vezes mais performático no tempo de execução e com tamanho de executável em média de 33,86 vezes maior que o executável gerado no gccgo utilizando um subconjunto de opções de otimização.

Palavras-chave: Compilador, Go Compiler, GCC.

Abstract

The Go language is one of the newer current languages with an increasing popularity in the software industry. It is defined by a specification and implemented by two compilers with different proposals to ensure that the specification is correct and complete. The gc compiler focuses on quick compilation and few optimizations while gccgo focuses on using GCC optimizations without worrying about the compiling time that is spent. This work proposes a study with the objective of constructing a genetic algorithm that assists in the identification of situations that gccgo can be more efficient than the Go Compiler. Results show that the Go Compiler is on average 24.7 times more performance at runtime and with an average executable file size of 33.86 times greater than the generated gccgo executable using a subset of optimization options.

Keywords: Compiler, Go Compiler, GCC.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

GC : *Go Compiler*

GNU : *GNU is Not Unix*

GCC : *GNU Compiler Collection*

GCCGo : *Frontend Go para GCC*

CSP : *Communicating Sequential Processes*

API : *Application Programming Interface*

GB : *Gigabyte*

TBB : *Threading Building Blocks*

LLVM : *Low Level Virtual Machine*

EEMBC : *Embedded Microprocessor Benchmarks*

DFG : *Data Flow Graph*

Capítulo 1

Introdução

Nesse capítulo será apresentada a motivação para esse estudo, o objetivo geral e os objetivos específicos e por fim a estrutura do trabalho.

1.1 Motivação

A linguagem de programação Go foi reconhecida em 2016 como a linguagem do ano baseado na pesquisa que mede a popularidade das linguagens de programação (ZDNET, 2017). Segundo os autores da pesquisa, os principais impulsionadores para a popularidade da linguagem foram a facilidade na curva de aprendizado e a sua natureza pragmática. Validando a informação do ano de 2016, um popular serviço de hospedagem de código fonte divulgou no início de 2017 que Go ocupa a nona posição no ranking de uso de linguagem de programação em projetos de código aberto hospedados no serviço (GITHUB, 2017).

A linguagem Go é definida por uma especificação, implementada de forma diferente e com foco distinto em dois compiladores modernos com a intenção de assegurar que a especificação da linguagem esteja correta e completa. O compilador gc (Go compiler) implementa a especificação da linguagem Go (GOOGLE, 2016) com objetivo de traduzir de forma extremamente rápida independentemente do tamanho da estrutura de código fonte. Em contrapartida, essa característica do compilador dificulta a inclusão de otimizações mais complexas que poderiam auxiliar na geração do código mais eficiente. Diferentemente da implementação do gc, o gccgo (Frontend Go para gcc) utiliza o gcc como backend que disponibiliza mais de uma centena de opções de otimizações para o uso à critério do usuário como entrada na compilação do código fonte.

Em compiladores, diversas etapas são necessárias para que um código fonte possa

ser traduzido para um formato que possa ser executado por um computador. Uma das etapas mais importantes é a fase de otimização de código, pois esta realiza transformações com objetivo de produzir um código mais eficiente. A quantidade dessas transformações ou otimizações podem variar de compilador para compilador e o tempo total gasto para traduzir o código pode variar dependendo da quantidade de transformações utilizada no compilador. Linguagens modernas impõem novas demandas para os projetistas de compiladores por disponibilizar facilidades nas abstrações dos problemas. Dessa forma é necessário criar algoritmos e representações para traduzir e dar suporte aos novos recursos das linguagens.

De acordo com Taylor (2012), o código fonte compilado no gccgo com determinadas opções de otimizações poderia ser 30% mais performático que o código fonte compilado no gc. Entretanto, não foi encontrado trabalho na literatura que validasse essa afirmação ou que realiza algum experimento de benchmark entre os dois compiladores. Identificar o compilador performático entre os disponíveis da linguagem Go, quando avaliados nas métricas de desempenho e uso dos recursos, requer uma analise experimental com foco nos recursos de destaque da linguagem.

Este trabalho apresenta um experimento que compara os resultados de performance entre os compiladores gc e gccgo nas métricas de corretude da solução, tempo total de execução, consumo de memória e tamanho do binário após compilação. Para entrada no benchmark, foram selecionados cinco problemas que foram implementados nas versões sequencial e concorrente. Para identificar um subconjunto ótimo de opções de otimizações para o gccgo foi implementado um algoritmo genético que auxiliou na busca por um subconjunto de opções quando comparado com os resultados de performance com opções de otimizações de uso geral. O experimento relaciona estatisticamente as aproximações dos resultados de cada métrica entre os compiladores, fornecendo informações para auxiliar na escolha do compilador mais adequado nos critérios de performance e tamanho de binário. Como principal contribuição e relevância científica deste trabalho devem ser destacados o estudo comparativo detalhado de performance entre os compiladores que pode ser úteis para outros experimentos comparativos.

1.2 Objetivos

Nesta seção são apresentados os objetivos deste trabalho.

1.2.1 Objetivo Geral

A presente pesquisa tem o objetivo de realizar um estudo de eficiência dos compiladores da linguagem Go, de forma a identificar, com a ajuda de um algoritmo genético, situações em que o gccgo possa ser mais performático que o Go Compiler.

1.2.2 Objetivo Específico

Os objetivos específicos deste trabalho são:

- **Objetivo 1** - Selecionar na literatura cinco algoritmos de programação paralela, visando servir como base para o experimento;
- **Objetivo 2** - Implementar os algoritmos selecionados utilizando recursos sequenciais e concorrentes da linguagem;
- **Objetivo 3** - Implementar um algoritmo genético para encontrar as soluções aproximadas de otimizações para o gccgo;
- **Objetivo 4** - Comparar resultados de benchmark entre os compiladores;
- **Objetivo 5** - Investigar as causas que justifiquem as diferenças de performance entre os compiladores;

1.3 Estrutura do Trabalho

O trabalho se encontra organizado em seis capítulos. Nesse capítulo foi apresentado a motivação, os objetivos e a estrutura do trabalho.

O capítulo 2, da fundamentação teórica, esclarece os fundamentos e conceitos teóricos necessários para a melhor compreensão do trabalho, situando-o no seu escopo de conhecimento.

O capítulo 3 contém a revisão de literatura referente a performance da linguagem Go e busca de subconjunto de opções de otimização do compilador GCC.

No capítulo 4 apresenta a metodologia do experimento, a qual foi utilizada para a realização desta pesquisa.

No capítulo 5 apresenta os detalhes do experimento e os resultados obtidos com as implementações dos objetivos propostos.

Por último, no Capítulo 6, serão apresentadas as conclusões e os trabalhos futuro.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo apresentar uma revisão dos principais conceitos necessários para o desenvolvimento desse trabalho, sendo esses Compilador, Algoritmos Genéticos e a Linguagem de Programação Go.

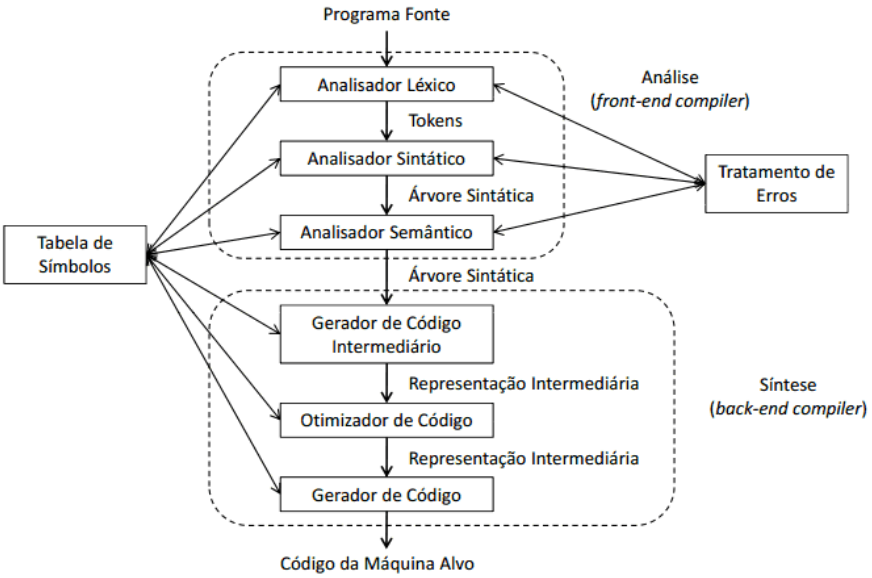
2.1 Compilador

Compilador é o responsável em transformar as notações computacionais em instruções que possam ser executadas por uma máquina (AHO et al., 2007). Essas notações são escritas geralmente através de uma linguagem de programação de alto nível, chamados de códigos fonte, que após passar como entrada para o compilador serão mapeados semanticamente a uma linguagem objeto capaz de ser executado pelo usuário.

Para realizar o mapeamento, o compilador divide esse procedimento em duas etapas macros denominadas de análise (front-end compiler) e síntese (back-end compiler) conforme ilustrado na figura 2.1. Na etapa de análise, o código fonte é subdividido em partes constituintes obedecendo uma estrutura gramatical que produzirá uma representação intermediária do código fonte. Em seguida, coleta informações do código fonte e os armazena em uma estrutura de dados chamada tabela de símbolos que é repassada junto com a representação intermediária para a fase de síntese onde é gerado o programa objeto.

Essas duas etapas podem ser subdivididas em seis fases que são: 1) análise léxica, 2) análise sintática, 3) analise semântica, 4) geração de código intermediário, 5) otimização de código e 6) a geração de código final. Para o estudo de performance dos compiladores, a principal fase responsável pela melhoria de performance é a fase de otimização de código (EIGENMANN, 2017).

Figure 2.1: Estrutura de um compilador.



Fonte: EIGENMANN (2017)

2.1.1 Otimização de Código

A fase de otimização de código é responsável por examinar o código intermediário e gerar um outro ainda melhor. Um código melhor geralmente significa mais rápido, porém também pode significar código com menos instruções, ou que use menos acessos à memória.

Nesta fase, é possível melhorar o código intermediário se for possível eliminar ou transformar alguma instrução, por exemplo, eliminar subexpressões comuns, eliminar código que não pode ser alcançado durante a execução do programa, alocar variáveis em áreas com menor custo, transformações baseadas em propriedades algébricas, como a comutatividade e a associatividade, entre outras (COOPER; TORCZON, 2011).

Segundo Aho et al. (2007), varia muito o número de otimizações de código realizadas por diferentes compiladores e quanto maior o número de operações de otimizações mais tempo é gasto nessa fase. O autor destaca também que não existe garantia que o código resultante é ótimo sob qualquer medida matemática, mas que muitas transformações simples podem melhorar consideravelmente o tempo de execução ou o espaço demandado pelo programa objeto.

Uma forma fácil de entender o tamanho do espaço de pesquisa de otimização de código é quando consideramos o exemplo de ordenação de fases, no qual os projetistas de compiladores definem um limite no número de otimizações durante uma compilação.

Table 2.1: Cálculos aproximados para seleção de opção de otimização

Número de otimizações disponíveis para ordenação de fases = ops	$= 40$
Sequência total da fase de ordenação de sequência de otimizações = len	$= 25$
Tempo de compilação e execução de um trecho de código = 1 n seg.	$= 10^{-9} seg$
Número de sequências únicas = ops len	$= 1.125 * 10^{34}$
Tempo gasto para avaliar uma sequência	$= 1 * 10^{-9} seg$
Tempo total para todas as avaliações	$= 1.125 * 10^{25} seg$
Idade do universo	$= 4.354 * 10^{17} seg$

Utilizando o compilador JVM RVM como o compilador alvo do exemplo, ao selecionar a opção de otimização O3 o compilador aplicará 67 opções e ao considerarmos que existem apenas 40 otimizações que podem ser aplicadas e nossa ordenação de fase é limitada em comprimento a um máximo de 25 otimizações então o número de pedidos possíveis de otimização seria 40^{25}. Se levássemos o tempo total necessário para compilar e executar um trecho de código para um nano de segundo, o tempo total necessário para avaliar toda a sequência possível ainda seria mais do que milhões de vezes a idade do universo (KULKARNI, 2014).

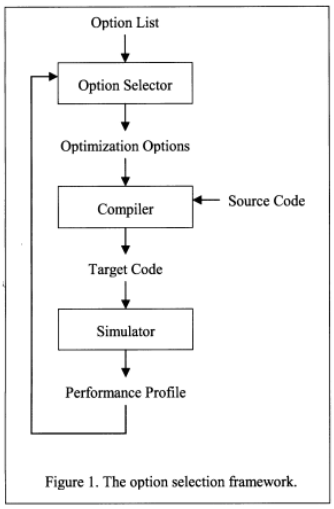
Podemos ver na tabela 2.1 que a quantidade de tempo necessária está mais próxima de milhões de vezes a idade do universo em que foi utilizado uma sequência fixa em todos os métodos de um benchmark, e que segundos os autores (LI; PADUA, 2005) mostraram que cada método funcionaria melhor com sequências de otimizações individualmente personalizadas. Isso significa que o número de combinações possíveis seria ainda maior no mundo real. Assim, a força bruta não pode e nunca será uma opção para iterar o espaço de busca de todas as ordenações de fases disponíveis (KULKARNI, 2014).

2.1.2 Seletor de Opção de Otimização de Código

O número de configurações de otimização que geram código preciso é extremamente grande. Não é possível para um compilador ou o engenheiro de compilador testar todas as configurações possíveis. Segundo Kulkarni (2014), seletor de opções de otimização de código é principalmente um algoritmo de otimização que tenta pesquisar um conjunto de opções de otimização para um programa de origem de entrada com o desempenho ideal conforme ilustrado na figura 2.2.

A princípio existem três questões que tornam esse problema de otimização um problema muito difícil e demorado. Primeiro, o grande número de opções de otimização torna o espaço de busca extraordinariamente grande. Segundo, o desempenho de um conjunto

Figure 2.2: Representação de um seletor de opção de otimização de código



Fonte: KULKARNI (2014)

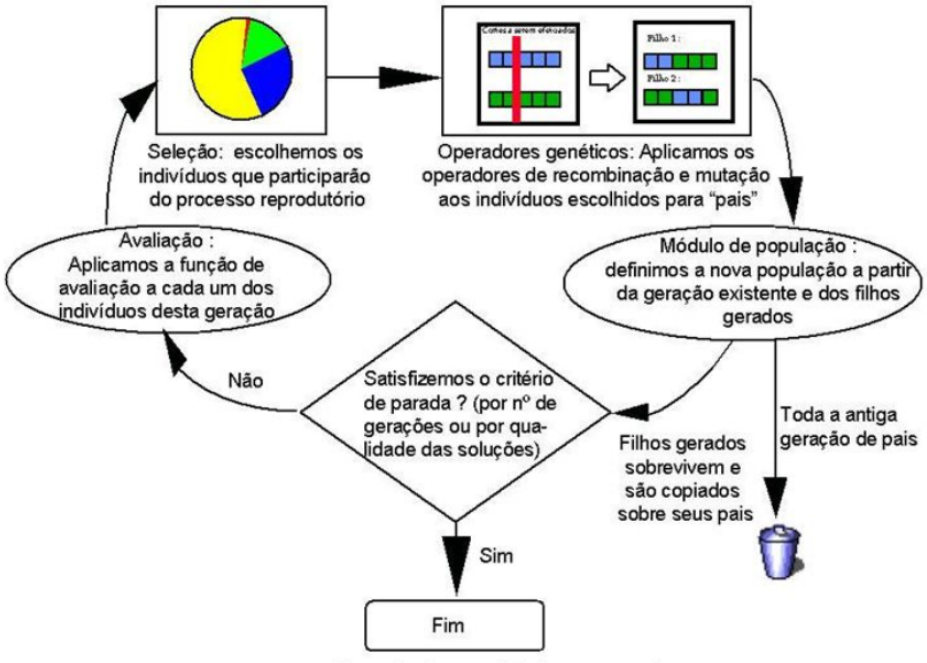
de opções de otimização para esse programa de entrada só pode ser conhecido após a compilação do mesmo com esse conjunto de opções de otimização e em seguida, criar o perfil do desempenho do código de destino gerado por meio de execução direcionada ou simulada. Isso torna a determinação do custo de um conjunto de opções de otimização muito demorada. Em terceiro lugar, as interações entre as opções de otimização dependem do código-fonte de entrada de forma tão complexa que é muito difícil encontrar heurísticas para acelerar o processo de busca (LIN CHI-KUANG CHANG, 2008).

2.2 Algoritmos Genéticos

Em 1975, o John Holland propôs um algoritmo genético para simular sistemas adaptativos (HOLLAND, 1975) como uma metáfora para os processos evolutivos com objetivo de estudar a adaptação e a evolução no mundo real dentro de computadores. David Goldberg aprofundou os estudos apresentandos por Holland ao publicar em 1989 o trabalho titulado *Genetic Algoritms in Search Optimization and Machine Learning*, utilizando como base o princípio apresentado por Charles Darwin de que indivíduos melhores adaptados tem mais chances de sobreviver e gerar descendentes (GOLDBERG, 1989) (GOLDBERG D., 1989).

Os algoritmos genéticos fazem parte de um ramo dos algoritmos evolucionários que partem de uma população de indivíduos que se desenvolvem através de operadores genéticos afim de aprimorar a adaptação média dos indivíduos no transcorrer das gerações. Ao inspirarem-se em mecanismos de evolução, os algoritmos genéticos têm como objetivo

Figure 2.3: Exemplo de Algoritmo Genético.



Fonte: LINDEN (2008)

encontrar eficientemente resultados ótimos ou sub-ótimos, ou seja, encontrar soluções mais próximos da melhor solução (LINDEN, 2012). No pseudo-código AlgGenBásico é apresentado um algoritmo genético básico que implementa a Figura 2.3 do algoritmo básico proposto por Linden (2008).

```
1:  procedure AlgGenBasico
2:      Inicialize a população P aleatoriamente
3:      Avalie os indivíduos na população P
4:      for Até o critério de parada ser satisfeito do
5:          for Até população P' completa do
6:              Selecione 2 indivíduos em P
7:              Aplique operadores de recombinação com probabilidade pr
8:              Aplique operadores de mutação com probabilidade pm
9:              Insira novos indivíduos em P'
10:         end for
11:         Avalie os indivíduos na população P'
12:         P ← P'
13:     end for
14: end procedure
```

Figure 2.4: Representação do cromossomo em cadeia binário, inteiro e real.

Representação Binária

G0	G1	G2	G3	G4	G5
1	1	0	1	0	1

Representação Inteira

G0	G1	G2	G3	G4	G5
10	13	400	12	4	18

Representação Real

G0	G1	G2	G3	G4	G5
10,5	1114,1	3,01	17,4	41,2	50,1

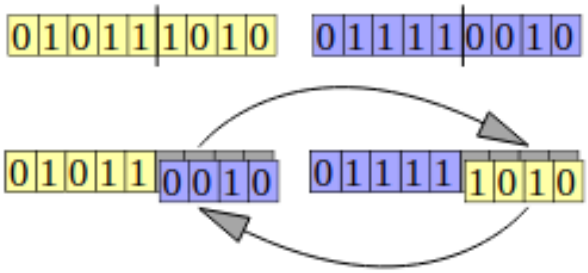
Fonte: Elaborado pelo autor.

2.2.1 Operadores Genéticos

A eficiência do funcionamento do algoritmo genético depende de parâmetros de controle da sua estrutura como o tamanho da população e os operadores genéticos. O tamanho da população é a representação da quantidade de indivíduos que compõem a população. O tamanho dessa representação influência no tempo total na avaliação da população necessitando de um equilíbrio. Quando o número de indivíduos é pequeno existem possibilidades de perda da diversidade para a busca da melhor solução e quando o número de indivíduos é grande existem possibilidades de utilizar mais recursos computacionais para avaliar a população (COSTA, 2009).

Os operadores genéticos são transformações da população por variações do material genético em sucessivas gerações com objetivo de obter indivíduos mais aptos. O material genético é a representação da informação do problema do mundo real capaz de conectar com as propriedades relevantes do algoritmo genético afim de obter o melhor resultado possível (LINDEN, 2012). Tradicionalmente na literatura, o material genético é denominado de cromossomo e representado como uma cadeia binária. Alguns algoritmos genéticos utilizam uma cadeia de reais e são geralmente utilizados para otimizações com restrição (LEMONGE, 1999). Na figura 2.4 é ilustrado uma representação binária, inteira e real.

Figure 2.5: Representação da recombinação.



Fonte: SOUZA (2014)

2.2.2 Recombinação

Esse método é baseado pela troca de material genético entre os pais durante o processo de reprodução humana, tendo como consequência a combinação das características para os filhos. Existem vários modos para realizar a troca de material genético e a eficiência do mesmo depende da representação do cromossomo (SOUZA, 2014). Na figura 2.5 é possível visualizar uma recombinação de um único ponto onde todos os dados além do ponto selecionados são trocados entre os progenitores resultando o material genético para os filhos.

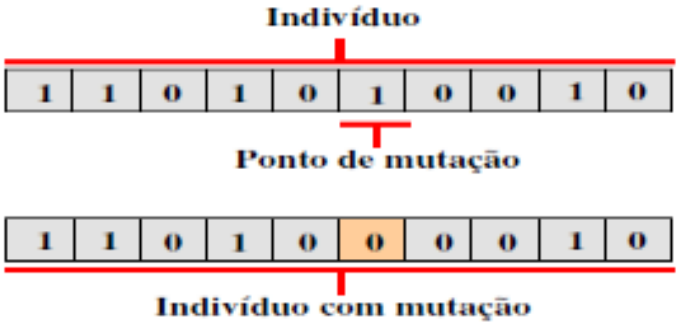
2.2.3 Mutação

Esse método é baseado na introdução de material genético na população, ou seja, injeta novos cromossomos na população (COSTA, 2009). Essa operação permite que o algoritmo genético explore novas soluções fora das limitações definidos nos indivíduos. A aplicação da mutação é feita em uma parcela pequena, entretanto a melhor taxa depende do problema, do tamanho da população e outros fatores (VOSE, 2004). Na figura 2.6 é possível visualizar uma mutação de um gene de um cromossomo de cadeia binária.

2.2.4 Método de Avaliação

Esse método tem a funcionalidade de avaliar a qualidade de um indivíduo para a solução do problema proposto no Algoritmo genético. Basicamente o método de avaliação é uma função que retorna uma nota do indivíduo no qual será utilizado por operadores de seleção do algoritmo genético.

Figure 2.6: Representação da mutação de um gene.



Fonte: COSTA (2009)

2.3 Go

Go é uma linguagem de programação de proposito geral, estaticamente tipada, compilada e com gerenciamento automático de memória (coletor automático de lixo) (PETTERSSON; WESTRUP, 2014). Foi criada pelo Google em setembro de 2007 e liberado ao público como código livre em novembro de 2009 na licença BSD (GOOGLE, 2009). Segundo os autores (GRIESEMER et al., 2017), a principal motivação para a criação da linguagem foi a frustração dos desenvolvedores ao tentar utilizar alguma linguagem da época que tivesse uma compilação eficiente, uma execução eficiente e com facilidades na programação.

Go combina a facilidade de programação de uma linguagem interpretada, dinamicamente tipada com a eficiência e a segurança de uma linguagem compilada com tipos estáticos. Suas características mais importantes são a facilidade em criar concorrência, compilação multiplataforma e a simplicidade na escrita (PIKE, 2009). Essas características são essenciais para as aplicações nos tempos atuais onde a infraestrutura converge para a nuvem com objetivo de facilitar a utilização eficiente dos recursos (OSTERMANN et al., 2009), possibilitar o uso de processadores com uma enorme quantidade de unidades de processamento e executar em diversas opções de plataformas (ASANOVIC et al., 2008). Por esse motivo, é comum encontrar a definição da linguagem Go como a linguagem C do século 21 (SUMMERFIELD, 2012).

A instalação padrão do Go acompanha o compilador, um conjunto completo de bibliotecas e ferramentas que auxiliam no desenvolvimento de aplicações de proposta geral. Entre as ferramentas podemos destacar o build, responsável em auxiliar no uso do compilador; o test, responsável em executar os suites de testes; e o gofmt, responsável em

automatizar o estilo padrão do código fonte.

Também é importante enfatizar que a linguagem é definida por uma especificação e não por uma implementação de um compilador. Por esse motivo, a equipe da linguagem escreve e mantém dois compiladores modernos, o gc (Go Compiler) e o gccgo (frontend Go para GCC), que implementam a especificação da linguagem de forma diferente e com focos distintos com a intenção de assegurar que a especificação esteja correta e completa (DONOVAN; KERNIGHAN, 2015).

2.3.1 Goroutines e Channels

Go implementa o Processo de Comunicação Sequencial (CSP) como o paradigma de programação concorrente no qual estabelece que a composição paralela de um número fixo de processos sequenciais se comuniquem de forma síncrona por mensagens de padrão fixo (HOARE, 1978).

Como mostrado na lista 2.1, a linguagem utiliza a palavra reservada go para indicar que uma expressão de chamada de função será executada através de um processo concorrente com uma estrutura leve e operado no mesmo espaço de endereço do processo pai (PUMPUTIS, 2015). Essa estrutura é denominada de goroutine e sua principal característica é inicializar uma estrutura de dados pequena, por exemplo, uma estrutura de dados inicial de uma goroutine utiliza inicialmente 8 kilobytes enquanto que uma estrutura de dados de uma thread do sistema operacional Linux x86-32 necessita inicialmente de 2 megabytes (BUTTERFIELD, 2016).

Listing 2.1: Iniciando uma goroutine

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func f(from string) {
8      for i := 0; i < 3; i++ {
9          fmt.Println(from, ":", i)
10     }
11 }
12
13 func main() {
14     go f("goroutine")
```

```
15 }
```

A linguagem fornece canais de comunicação para permitir a comunicação entre goroutines e para sincronização de estados. Basicamente um segmento aguardará a gravação de dados em um determinado canal e executará sua parte de lógica com base nesses dados, enquanto outros N goroutines podem estar escrevendo dados no canal. As operações do canal são unidirecionais, impedindo qualquer condição de corrida de dados entre eles (**??**). A sintaxe para enviar e receber dados é mostrada na lista 2.2.

Listing 2.2: Utilizando canais

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      syncChan := make (chan int)
9      done := make(chan bool, 2)
10     go consumer(syncChan, done)
11     go producer(syncChan, done)
12     <- done
13     <- done
14 }
15
16 func consumer (input <- chan int, done chan bool) {
17     in := <- input
18     fmt.Printf("%d \n" , in)
19     done <- true
20 }
21
22 func producer(output chan <- int, done chan bool) {
23     output <- (1 << 3)
24     done <- true
25 }
```

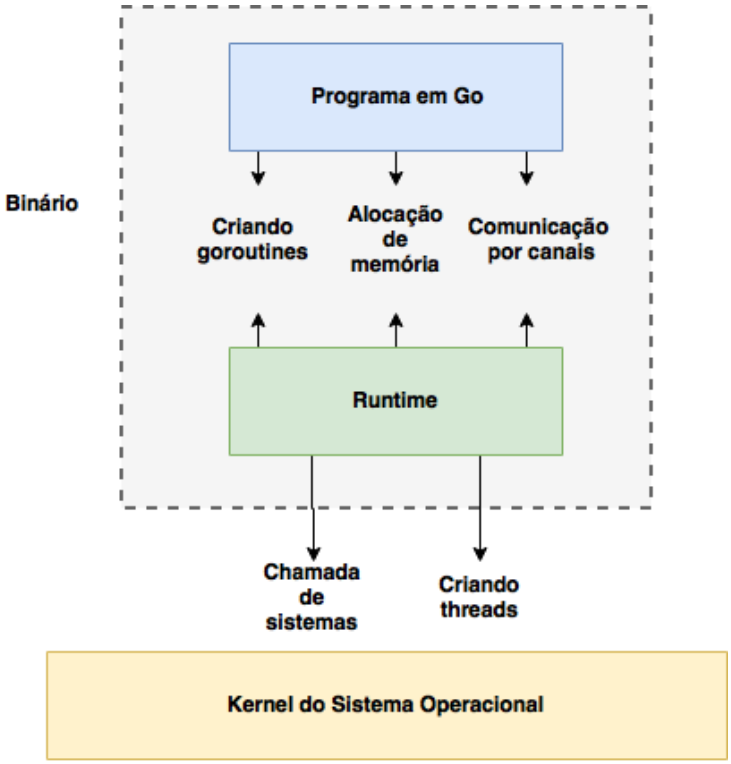
2.3.2 Go Runtime

Como a linguagem Go fornece construções de alto nível, como goroutines, canais e gerenciador automático de memória, é necessária um suporte em tempo de execução

(runtime) para garantir esses recursos. A Go Runtime foi escrita inicialmente em C e é ligada estaticamente ao código do usuário toda às vezes na compilação durante a fase de ligação (linker). Por estar ligado ao código do usuário, um programa Go é um executável autônomo no espaço do usuário para o sistema operacional, então o seu trabalho consiste em interagir com o sistema operacional e com o código Go escrito pelo programador.

Podemos imaginar um programa Go em execução como uma única aplicação composta por duas camadas bem delimitadas: o código do usuário e a runtime. Eles se comunicam através da chamada de funções para gerenciar goroutines, canais e outras construções de alto nível. Por exemplo, qualquer chamada que o código de usuário faça para as APIs do sistema operacional é interceptada pela camada da runtime (GOOGLE, 2017), como é ilustrado na Figura 2.7.

Figure 2.7: Diagrama da relação da runtime, Sistema Operacional e código do usuário.



Fonte: Elaborado pelo autor.

Uma das responsabilidades mais importantes da runtime de Go é o agendador de goroutine, pois esse gerenciamento é crucial para o desempenho eficiente dos programas. Para isso, a runtime acompanha o estado de cada goroutine com o intuito de agendar a execução em um conjunto de threads pertencentes ao processo. Em outras palavras, as goroutines são separadas das threads e a runtime gerencia o momento de disponibilizar uma thread para uma goroutines.

Embora possam existir várias threads para uma aplicação Go em execução, a proporção de goroutines para threads deve ser superior a 1-para-1, pois muitas das vezes são necessárias várias threads com objetivo de garantir que as goroutines não sejam bloqueadas desnecessariamente. Por exemplo, quando um goroutine faz uma chamada de recurso bloqueante, é alterado o estado do segmento executado para bloqueado e pelo menos uma nova thread é criada pela runtime para continuar a execução de outros goroutines que não estão bloqueadas no momento. Existe um limite máximo de threads executados em paralelo e esse valor é definido na variável de ambiente chamada GOMAXPROCS (VARGHESE, 2015).

No nível de implementação da linguagem, a runtime abstrai o gerenciamento de goroutines através de três principais estruturas que foram inicialmente escritas na linguagem C (GOOGLE, 2017). Essas estruturas são: a estrutura G, estrutura M e a estrutura Sched. Como ilustrado no código 2.3, uma instância da estrutura G representa uma única goroutines com a referência ao código responsável pela execução como também as informações necessárias para acompanhar o estado atual e a pilha. Já uma instância da estrutura M representa uma única thread do Sistema Operacional com a referência da instância G que está em execução no momento, como também a fila global de instâncias de goroutines e outras informações, como ilustrado no código 2.4.

Listing 2.3: Campos relevantes da estrutura G

```
1  struct G {
2      byte*    stackguard;      // stack guard information
3      byte*    stackbase;       // base of stack
4      byte*    stack0;          // current stack pointer
5      byte*    entry;           // initial function
6      void*    param;           // passed parameter on wakeup
7      int16    status;          // status
8      int32    goid;            // unique id
9      M*       lockedm;         // used for locking M's and G's
10     ...
11 };
```

Listing 2.4: Campos relevantes da estrutura M

```
1  struct M {
2      G*       curg;              // current running goroutine
3      int32    id;                // unique id
4      int32    locks;             // locks held by this M
5      MCache   *mcache;           // cache for this thread
6      G*       lockedg;           // used for locking M's and G's
```

```
7     uintptr createstack[32];     // Stack that created this thread
8     M*      nextwaitm;           // next M waiting for lock
9     ...
10 };
```

Tal qual ilustrado no código 2.5, a estrutura Sched é uma única estrutura global com referências as diferentes instâncias do tipo G e do tipo M e outras informações necessárias para o agendamento. Entre essas informações, existe a referência de uma fila de instâncias de G disponíveis para execução como também outra referência de uma fila de instâncias de G finalizados. Também existe uma referência de uma fila de instâncias de M que indica a quantidade de threads ociosos aguardando executar instâncias da estrutura G.

Listing 2.5: Campos relevantes da estrutura Sched

```
1  struct Sched {
2      Lock;                // global sched lock .
3                           // must be held to edit G or M queues
4      G       *gfree;      // available g's ( status == Gdead)
5      G       *ghead;      // g's waiting to run queue
6      G       *gtail;      // tail of g's waiting to run queue
7      int32   gwait;       // number of g's waiting to run
8      int32   gcount;      // number of g's that are alive
9      int32   grunning;    // number of g's running on cpu
10                          // or in syscall
11     M       *mhead;      // m's waiting for work
12     int32   mwait;       // number of m's waiting for work
13     int32   mcount;      // number of m's that have been created
14     ...
15 };
```

2.3.3 Gerenciamento Automático de Memória

Diferentemente de algumas linguagens, o programador da linguagem Go não precisa se preocupar com a destruição de um objeto ou o uso de um objeto já destruído em memória pelo fato da linguagem implementar o paradigma do Garbage Collection (DONOVAN; KERNIGHAN, 2015).

O Garbage Collection lida principalmente com duas preocupações centrais no uso de recursos. A primeira é encontrar e recuperar a memória não utilizada pela aplicação e a segunda é lidar com a fragmentação da memória estática (heap) onde é assim chamada porque seus valores podem permanecer até o fim do programa (JONES; MOSS, 2012).

Para esse fim, o garbage collection da linguagem Go utiliza algumas técnicas onde destacamos a técnica do Stop-the-world que primeiramente paralisa completamente a aplicação e, a partir de um conjunto de ponteiros raiz (registros, pilhas, variáveis globais, etc), rastreia e marcar o conjunto de objetos que estão vivos na aplicação. Em seguida, o aplicativo é retomado e quaisquer objetos não marcados serão adicionados à lista para serem liberados da memória. Sua simplicidade e alto rendimento tornam comuns o uso desta técnica, porém a sua principal desvantagem é o período de pausa que pode chegar de 10-40ms por GB de pilha impossibilitando muitas vezes aplicações de tempo real (GIDRA et al., 2013).

2.3.4 Compiladores

A linguagem é definida por uma especificação e não por uma implementação de um compilador (DONOVAN; KERNIGHAN, 2015). Por esse motivo, a equipe da linguagem escreve e mantém dois compiladores modernos, o gc (Go Compiler) e o gccgo (frontend Go para GCC), que implementam a especificação da linguagem de forma diferente e com foco distintos com a intenção de assegurar que a especificação esteja correta e completa.

2.3.4.1 Go Compiler (gc)

O Go Compiler (GC) é denominado como o compilador principal da linguagem por acompanhar a instalação padrão. Foi originalmente escrito em C e a partir da versão 1.5 foi quase inteiramente escrito em Go (GOOGLE, 2015).

O processo para a transição do gc de C para Go foi em sua grande parte automatizado e, como resultado, a base de código não usou verdadeiramente os recursos ou vantagens que o Go tem sobre C. Uma consequência disso é que o gc usa análises feitas à mão e o código de análise de tipo derivado da versão C, apesar da existência de pacotes de verificação de análise de tipo na biblioteca padrão do Go. O código para executar essas tarefas é bastante acoplado a outras partes do sistema. Por esse motivo, a adição de novos recursos ao sistema gc é difícil, e uma quantidade significativa de trabalho teria que ser dedicada a trabalhar em torno das complexidades do design do software C-like.

Por ser originalmente baseado no compilador do Plan9 C, sua principal característica é a compilação de forma extremamente rápida mesmo para uma grande estrutura de código fonte. Entretanto, essa característica dificulta intrinsecamente a inclusão de otimizações mais complexas no processo de compilação e que poderiam gerar códigos binários per-

formáticos e/ou de menor tamanho permitindo assim a possibilidade de um melhor desempenho da aplicação (DONOVAN; KERNIGHAN, 2015).

2.3.4.2 Go Frontend para o GCC (gccgo)

O gccgo é um frontend típico do gcc e escrito em C++ com mais de 50.000 linhas de código, incluindo linhas em branco e comentários com objetivo principal de gerar código assembly utilizado por assemblers e linkers do backend do GCC (TAYLOR, 2010).

Diferentemente da implementação do Go Compiler, gccgo possui a característica de não se preocupar com o tempo total de compilação, mas com a possibilidade de utilizar opções de otimizações para produzir binários menores e altamente otimizados.

O GCC possui um histórico de incentivo a construção de algoritmos de otimizações e que, disponíveis nos dias atuais, ultrapassa a marca de cem opções de otimizações que podem ser informados e/ou relacionados no momento da compilação na busca de mais performance (LI et al., 2014).

Por fornecer muitas possibilidades de otimização (PALLISTER et al., 2013), o GCC disponibiliza níveis de otimizações com objetivo de agrupar opções por tipos de otimizações, ou seja, os níveis de otimização combinam várias opções de compilação usando alguma heurística. Por exemplo, a opção de nível de otimização O1 representa para o compilador um conjunto de trinta opções de otimização que reduzirá o tamanho do código e o tempo de execução sem executar otimizações que levam uma grande quantidade de tempo de compilação.

Esse conjunto de opções (HAGEN, 2011) poderia ser informado manualmente pelo usuário no ato da compilação, por exemplo, ao invés de informar a opção de nível O1, o usuário informaria as opções fauto-inc-dec, fcompare-elim, fcprop-registers, fdce, fdefer-pop, fdelayed-branch, fdse, fguess-branch-probability, fif-conversion2, fif-conversion, fipa-pure-const, fipa-profile, fipa-reference, fmerge-constants, fsplit-wide-types, ftree-bit-ccp, ftree-builtin-call-dce, ftree-ccp, ftree-ch, ftree-copyrename, ftree-dce, ftree-dominator-opts, ftree-dse, ftree-forwprop, ftree-fre, ftree-phiprop, ftree-sra, ftree-pta, ftree-ter, funit-at-a-time.

Esses níveis de otimização pré-estabelecidos não utilizam todo o potencial por não explorar o restante das opções de otimização disponíveis no compilador. Entretanto, a escolha adequada das opções de otimização não é uma tarefa fácil para usuários comuns, pois para obter uma solução ideal é necessária uma busca exaustiva (MALIK, 2010).

Capítulo 3

Revisão da Literatura

Neste capítulo apresentamos uma revisão dos principais trabalhos relacionados ao tema de "performance dos compiladores" aplicados a linguagem Go, como também trabalhos que auxiliam na seleção de opções de otimização do compilador GCC.

3.1 Trabalhos Relacionados

Até o presente momento não foi encontrado na literatura trabalhos que fizeram um estudo de avaliação comparativa de desempenho exclusivamente entre os compiladores GC e GCCGo. Porém foram encontrados experimentos que analisaram a performance da linguagem Go em relação a outras linguagens e, por esse motivo, realizaram um estudo indireto na performance do compilador.

Em (NANZ et al., 2013) os autores realizaram um estudo comparativo com linguagens de programação com suporte a computação de multicore. Entre as linguagens estudadas no experimento estavam a Chapel, Cilk, Threading Building Blocks (TBB) e a Go. No experimento, foi implementado 6 problemas da literatura por desenvolvedores experientes de cada linguagem com intuito de comparar entre as linguagens o tamanho do código fonte, o tempo para codificar a solução, o tempo total da execução e o tempo de aceleração. Segundo os autores, a linguagem Go realizou um trabalho aceitável embora alguns problemas de performance tenham sido detectados no experimento. Na época da pesquisa, a linguagem Go era a mais nova dentre as linguagens escolhidas para o estudo e provavelmente o compilador não estava amadurecido nesses aspectos.

Um estudo entre as linguagens Go e Java foi realizado por Togashi e Klyuev (2014) ao realizar uma comparação nos quesitos de tempo de compilação e concorrência. Para o

experimento, foi implementado em cada linguagem um programa simples de multiplicação de matrizes na versão sequencial e na versão concorrente. Em todas as implementações foram utilizados recursos exclusivamente nativo das linguagens. Na aplicação concorrente escrita com a linguagem Java foi utilizado o recurso oriundo da class Thread e na aplicação escrita com a linguagem Go foi utilizado os recursos Goroutine e Channel. Os resultados dos experimentos mostraram que Go obteve a melhor performance no tempo de processamento tanto na versão sequencial quanto na versão concorrente. No quesito de tempo total de compilação, o compilador Go também foi melhor com um tempo médio três vezes superior que o da média do Java. O compilador utilizado no experimento foi o Go Compiler na versão 1.2.

Outro experimento com Go e uma outra linguagem com mecanismo de programação paralela de alto nível foi realizado por Serfass e Tang (2012) ao realizar uma comparação com a linguagem C++ TBB. Os autores Implementaram uma versão paralela do algoritmo de programação dinâmica da árvore de pesquisa binária em ambas as linguagens, baseadas em um gráfico de tarefas acíclicas diretas. Após medirem o tempo de execução de ambas as implementações, os resultados mostraram que o agendador de tarefa e sincronização no TBB eram menores com um desempenho geral de 1,6 à 3,6 vezes mais rápido do que os resultados encontrados na linguagem Go. Não foi informado a versão do compilador da linguagem Go utilizado no experimento, porém é possível deduzir que foi a versão 1.0 pela época da publicação.

No estudo (JOHNELL, 2015) o autor realizou um comparativo performático de Go com a linguagem Scala pelo fato da linguagem possuir coleções paralelas, futures e atores que podem ser utilizado para concorrência e programação paralela. Para o experimento, foram implementado as versões paralelas da multiplicação da matriz e da multiplicação da cadeia matricial e utilizado exclusivamente os recursos nativos de paralelismo das linguagens. Os resultados do estudo mostraram que Scala teve um melhor desempenho em comparação com Go na multiplicação da matriz paralela. Entretanto o autor ressalta que Go é mais eficiente na multiplicação de cadeia matricial quando ocorre um aumento significativo de uso de goroutines e atores. Não foi informado a versão do compilador da linguagem Go utilizado no experimento.

Hundt (2011) comparou o desempenho sequencial entre as linguagens C++, Java, Go e Scala. O autor utilizou a abordagem que permitisse uma comparação fiel dos recursos de linguagem como a complexidade do código, o tempo de compilação, tamanhos de binários, tempos de execução e uso de memória. Com base no tempo de execução, a linguagem

C++ foi declarada como a mais rápida no benchmark. Para o autor, os compiladores da linguagem Go ainda são imaturos, por isso refletiu nos resultados do desempenho quanto no tamanho gerado do binário. Não foi informada a versão do compilador da linguagem Go utilizado no experimento.

Uma comparação entre os compiladores GCC e o LLVM foi demostrado em (PARK et al., 2014) utilizando os benchmarks da EEMBC no simulador AE32000 onde mediu a contagem de instruções dinâmicas. Os códigos fontes foram compilados com a opção de otimização "O2" nos dois compiladores. Como resultado do experimento, o compilador GCC mostrou melhor desempenho na maioria das comparações. O LLVM foi bom no deslocamento do loop e na função aritmética inlet, mas na alocação de registro e na otimização de salto não obteve um bom desempenho. No quesito tamanho do binário, o GCC gerou 4% em média um binário menor que o do compilador LLVM. No experimento os autores utilizaram as versões dos compiladores GCC 4.7.1 e o LLVM 3.1 para o microprocessador embarcado AE32000.

Alguns trabalhos foram desenvolvidos para extração de características de aplicação com intuito de auxiliar na identificação de opções de otimizações para o compilador GCC. Entretanto não foi encontrado trabalhos que utilizassem, em conjunto com outras informações, a especificação ou características dinâmicas e estáticas da linguagem Go como entrada nos algoritmos de identificação de opções de otimização.

Malik (2010) utiliza o Data Flow Graph (DFG) resultante da compilação de uma aplicação para extrair informações estáticas do código fonte com objetivo de auxiliar na seleção de opções de otimizações do GCC através de técnicas de aprendizado de máquina. Os autores apresentam uma técnica utilizando informações baseadas na distribuição espacial das instruções no grafo ao transformar em características para um algoritmo de aprendizado de máquina. Segundo os autores, com essa técnica foi possível obter um ganho de 70% na velocidade de selecionar opções de otimizações ao comparar com uma pesquisa interativa usando 1000 interações. Foi utilizado o GCC 4.4.1 com IBM Milepost Framework e a benchmark escrita na linguagem C com as suites MiBench e SPEC2006.

Em (LI et al., 2014) é proposto um método de geração de características estáticas de um código fonte para cada fase de otimização realizado pelo compilador. Para isso, os autores utilizam um algoritmo genético para extrair essas informações após cada ciclo de otimização. São reunidos como conjuntos de dados para treinar o modelo de aprendizagem os melhores registros de desempenho de diferentes alvos, com os planos de otimização e vetores de características. Os resultados da avaliação mostraram que o método foi mais

performático que o nível de otimização O3 do GCC nos benchmarks. No experimento, os autores utilizaram o GCC 4.6.0 compilado com a opção de plugins habilitado e, para o benchmark, foi utilizado o MiBench com códigos do CBench com KDataSets.

Os autores Lin Chi-Kuang Chang (2008) propõem um algoritmo genético com peso nos genes para buscar opções de otimização do GCC. O algoritmo proposto associa um peso a cada gene para indicar uma adequação estimada da opção de otimização correspondente ao código fonte de entrada. Durante cada geração da evolução, os pesos dos genes são modificados de acordo com a aptidão atual. No experimento, os autores utilizam o GCC 4.1.2 que disponbilizava 128 opções de otimização e foi utilizado com códigos escritos em C no simulador ADS 1.2

Capítulo 4

Métodos Propostos

Neste capítulo são apresentados design do experimento, os problemas selecionados para o experimento como também os pseudocódigos dos problemas, do algoritmo genético e do benchmark. A seção 4.1 apresenta o projeto de execução do experimento, como ela foi conduzida e porque essa foi a abordagem escolhida. A abordagem adotada na seleção dos problemas da literatura é apresentada na seção 4.2. As seções 4.2.1 à 4.2.5 apresentam os algoritmos utilizados no experimento com mais detalhes e a seção 4.3 apresenta a estratégia adotado para coletar as métricas dos binários. Por fim, a seção 4.4 apresenta os detalhes do algoritmo genético utilizado no experimento.

4.1 Design do Experimento

Este experimento foi dividido em três etapas com objetivo de facilitar o desenvolvimento e a execução. A primeira etapa foi a implementação de cinco problemas da literatura tanto nas versões utilizando os recursos sequenciais quanto utilizando os recursos concorrentes da linguagem e da ferramenta para auxiliar na extração das métricas em tempo de execução dos binários.

A segunda etapa foi a execução do experimento utilizando como entrada as implementações citadas anteriormente, as quais foram compiladas usando o gc e o gccgo. Na compilação com o compilador gccgo, foi utilizado como entrada as opções de otimização de uso padrão como por exemplo as opções "-O2" e "-fgo-optimize-allocs". Esse experimento teve como objetivo validar a hipótese de que o gccgo poderia ser mais performático do que o gc utilizando opções de uso geral, porém os resultados mostraram que os binários compilados no gc obteve um melhor desempenho do que os binários compilados com o gccgo.

Como a primeira hipótese alternativa foi refutada, foi implementado um algoritmo genético para auxiliar na seleção de um subconjunto ótimo de opções de otimização que fossem mais performático do que a métrica de tempo de execução encontrado no experimento anterior do gccgo.

A terceira etapa foi a execução do experimento utilizando como entrada as implementações dos problemas compiladas no gc e no gccgo com o subconjunto ótimo de opções de otimização encontrados com o auxílio do algoritmo genético. Esse experimento teve como objetivo validar a hipótese de que o gccgo poderia ser mais performático do que o gc utilizando o subconjunto ótimo de opções de otimização, porém os resultados mostraram que os binários compilados no gc foram mais performático do que os binários compilados com o gccgo. Por fim, partimos para uma analise das causas com base nos resultados dos experimentos para identificar o que motivou o ganho de performace do compilador gc.

Em resumo a execução do experimento utilizou os compiladores gc e gccgo, as implementações sequenciais e concorrentes do problemas de programação paralela, uma ferramenta de benchmark para auxiliar na coleta das métricas em tempo de execução, implementação de um algoritmo genético para auxiliar na seleção de um subconjunto de opções de otimização e por fim, resultando no estudo dos compiladores para identificar as causas dos resultados encontrados.

4.2 Seleção dos Problemas de Programação Paralela

Existem na literatura uma infinidade de conjunto de problemas de programação paralela com complexidade de implementação variada (WILSON et al., 1993; WILSON; BAL, 1996; FEO, 2016; ASANOVIC et al., 2006; ASANOVIC et al., 2009). Entretanto, o peso principal para seleção dos problemas está na facilidade de implementação de forma sequencial quanto concorrente.

Para esse experimento, foram selecionados cinco problemas de um conjunto maior descritos por Wilson e Irvin (1995). Esses problemas também foram utilizados por Nanz et al. (2013) para auxiliar no benchmark na comparação de linguagens de programação com suporte a concorrência. As descrições dos problemas serão detalhados nas próximas seções para um melhor entendimento.

4.2.1 Random Number Generation (randmat)

Este algoritmo tem como objetivo preencher uma matriz com números inteiros aleatórios. Para isso, recebe como entrada o número de linhas e colunas da matriz e a semente de geração de números aleatórios.

A saída do algoritmo é apresentado por uma matriz com delimitações informados na entrada preenchidas com números aleatórios conforme o Pseudocódigo 1.

Algorithm 1 Pseudocódigo do Random Number Generation

```
1: procedure RANDMAT(numLinha, numColuna, semente)
2:     matriz ← NovaMatrix(numLinha, numColuna)
3:     l ← 0
4:     for l ≤numLinha do
5:         c ← 0
6:         for c ≤numColuna do
7:             numAleatorio ← RandNumber(semente)
8:             matriz[l][c] ← numAleatorio
9:         end for
10:    end for
11:    return matriz
12: end procedure
```

4.2.2 Outer Product (outer)

Este algoritmo tem como objetivo transformar um vetor contendo posições dos pontos em uma matriz simétrica, densa, calculando as distâncias entre cada par de pontos. É recebido como entrada um vetor com posições dos pontos representado por x e y, o número de pontos no vetor e o tamanho da matriz.

Como saída, uma matriz com valores das distâncias entre os pontos e um vetor com valores das distâncias do ponto inicial conforme o Pseudocódigo 2.

4.2.3 Matrix-Vector Product (product)

Este algoritmo tem como objetivo calcular o produto entre uma matriz A e um vetor V. A entrada são uma matriz real, um vetor real e o número de valores no vetor e o tamanho da matriz ao longo de caixa eixo. A saída é um vetor real com os resultados do produto. No Pseudocódigo 3, é possível observar os passos realizados pelo algoritmo.

Algorithm 2 Pseudocódigo do Outer Product

```
1: procedure OUTER(posPontos, numPontos)
2:     matriz ← NovaMatrix(numLinha, numColuna)
3:     distancias ← NovoVetor(numPontos)
4:     i ← 0
5:     for i ≤posPontos do
6:         numMax ← 0
7:         j ← 0
8:         for j ≤posPontos do
9:             if i ≠j then
10:                d ← CalcDist(posPontos[i].x, posPontos[i].y, posPontos[j].x, posPontos[j].y)
11:                if d > numMax then
12:                    numMax ← d
13:                end if
14:                m[i ∗ (numPontos + j)] ← d
15:            end if
16:        end for
17:        m[i ∗ (numPontos + 1)] ← numPontos ∗ numMax
18:        distancias[i] ← CalDist(0, 0, posPontos[i].x, posPontos[i].y)
19:    end for
20:    return matriz, distancias
21: end procedure
```

Algorithm 3 Pseudocódigo do Matrix-Vector Product

```
1: procedure PRODUCT(matriz, vetor, tamanhoVetor)
2:     produto ← NovoVetor(tamanhoVetor)
3:     i ← 0
4:     for i < tamanhoVetor do
5:         soma ← 0.0
6:         j ← 0
7:         for j < tamanhoVetor do
8:             soma ← soma + matriz[i ∗ tamanhoVetor + j] ∗ vetor[j]
9:         end for
10:        produto[i] ← soma
11:    end for
12:    return produto
13: end procedure
```

4.2.4 Histogram Thresholding (thresh)

Este algoritmo tem como objetivo executar um limiar de histograma em uma matriz. Em outras palavras, dado uma matriz inteira I e um percentual P, o algoritmo constrói uma matriz booleana onde Bij é verdade se, e somente se, não mais de P por cento dos valores em I são maiores que Iij. Para isso é informado como entrada para o algoritmo uma matriz, o número de linhas e colunas e a porcentagem mínima. A saída é uma matriz booleana resultante do limiar conforme o Pseudocódigo 4.

Algorithm 4 Pseudocódigo do Histogram Thresholding

```
1: procedure THRESH(matriz, numLinha, numColuna, percentualMin)
2:     resultado ← NovaMatrix(numLinha, numColuna)
3:     limiar ← (numLinha * numColuna * percentualMin)/100
4:     i ← 0
5:     for i < numLinha do
6:         j ← 0
7:         for j < numColuna do
8:             resultado[i][j] ← matriz[i][j] >= calculoLimite(limiar)
9:         end for
10:    end for
11:    return resultado
12: end procedure
```

4.2.5 Weighted Point Selection (winnow)

Este algoritmo tem como objetivo converter uma matriz de inteiros em vetor de pontos representados por x e y. É informado ao algoritmo uma matriz de inteiros, uma matriz booleana, o número de linhas e colunas na matriz e o número de pontos selecionáveis. A saída será um vetor de pontos com uma representação de x e y conforme observado no pseudocódigo 5.

4.3 Seleção das Métricas

Foram selecionados quatro métricas de avaliação para cada experimento independente da versão implementada do problema.

A primeira métrica corresponde a corretude da solução. Foi considerada uma execução bem sucedida, ou resolvida, aquela que teve os mesmos valores de saída ao ser comparada com o resultado da solução correta.

Algorithm 5 Pseudocódigo do Weighted Point Selection

```
1: procedure WINNOW(matrizInt, matrizBool, numLinha, numColuna, numPontos)
2:     vetor ← NovoVetor(len(matrizInt))
3:     pontos ← NovoVetor(len(matrizInt))
4:     i ← 0
5:     for i < numLinha do
6:         j ← 0
7:         for j < numColuna do
8:             if matrizBool[i][j] then
9:                 vetor = i * ncols + j
10:            end if
11:        end for
12:    end for
13:    limite ← len(matrizInt)/numPontos
14:    c ← 0
15:    for c < numPontos do points[i] = values.e[i*chunk]
16:        pontos[c] ← vetor[c * limite]
17:    end for
18:    return pontos
19: end procedure
```

A segunda métrica de avaliação refere-se ao tempo de execução, o que corresponde ao tempo decorrido após o pré-processamento até a parada por conclusão do processamento.

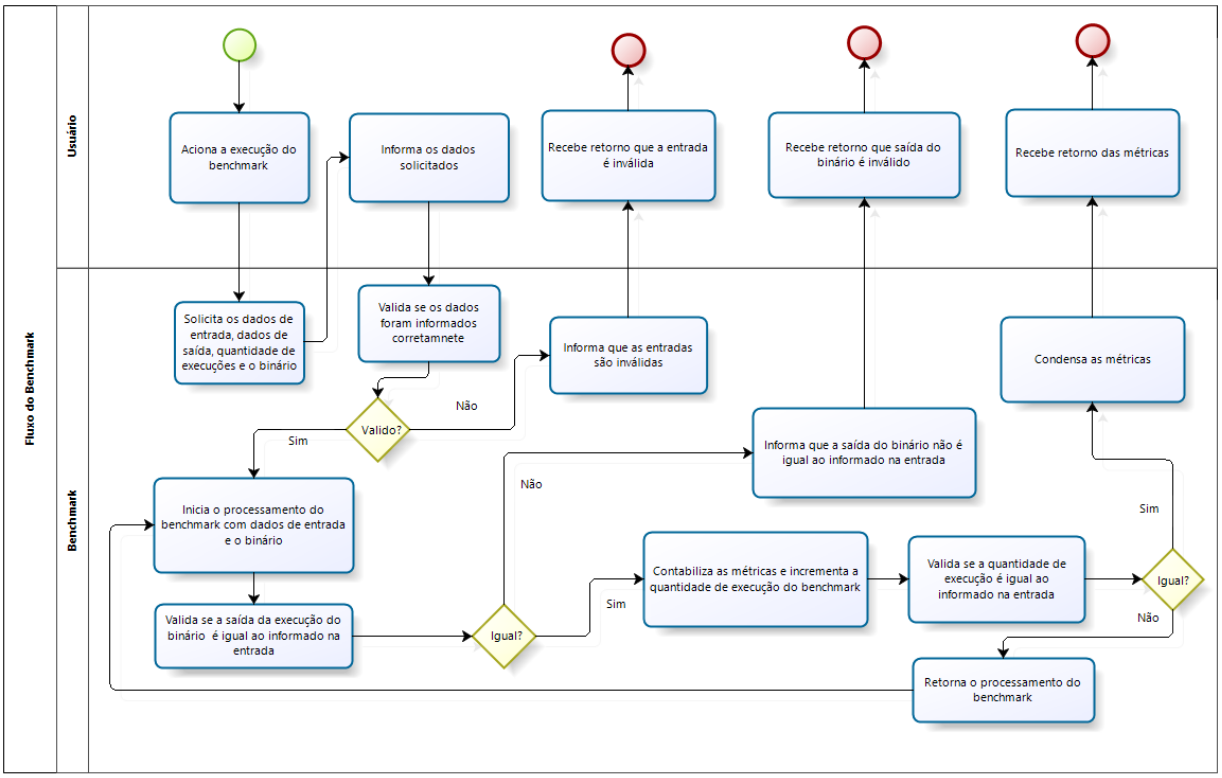
A terceira métrica usada foi o tamanho final do binário gerado após a compilação e, por fim, a quarta métrica foi o pico do uso de memória da aplicação desde o início da execução até o fim do processamento.

4.3.1 Coleta das Métricas

A coleta das métricas do experimento é uma das parte mais sensíveis do trabalho por estarem diretamente relacionado com os resultados e posterior discussão do mesmo. Como os binários gerados pelo o compilador Go não executam em maquinas virtuais que poderiam auxiliar na extração das informações como as aplicações Java (DUFOUR KAREL DRIESEN; VERBRUGGE, 2003), mas como um processo no sistema operacional, foi adotado a estratégia de acessar diretamente os dados catalogados na runtime do processo de cada binário executado no experimento. Como foi comentado na seção 2.3.2, a runtime gerencia as solicitações de recursos do binário para o sistema operacional e, portanto, seria a forma mais fidedigna de extrair essas informações.

Para auxiliar na coleta das métricas foi implementado uma solução denominada de benchmark que iria interagir com a runtime dos binários para catalogar o uso de memória

Figure 4.1: Fluxo do software de Benchmark.



Fonte: Elaborado pelo autor.

como também calcular o tempo total de execução, o tamanho final do binário e a corretude.

Como mostra na figura 4.1, o benchmark é responsável por validar as entradas do experimento como também se a saída da execução corresponde com a informada para garantir a integridade do experimento. A primeira validação é se o resultado da execução corresponde com o esperado afim de garantir a corretude da execução. No segundo momento são registradas e informadas as métricas e se a quantidade de execução extrapola ao que foi informado pelo usuário. Por fim, ao chegar no final do processamento do experimento, é condensado os resultados das métricas e retornado para o usuário como registros categorizado por métrica.

4.4 Seleção das Opções de Otimização

Como o problema de seleção de opções de otimização é um problema difícil e que é necessário um algoritmo que utilize técnica de busca (KULKARNI, 2014), foi implementado um algoritmo genético para auxiliar na solução do problema de identificar um subconjunto de opções de otimização para o compilador gccgo. Para isso, foram definidos a representação cromossomial e a função de avaliação baseado nas características desse experimento conforme será explicado a seguir.

4.4.1 Representação Cromossomial

A representação cromossomial utilizada nesse trabalho foi a cadeia binária por estar diretamente ligada ao conceito do uso de um opção de otimização (LIN CHI-KUANG CHANG, 2008). Em outras palavras, cada gene corresponde a uma opção dos 141 que estão disponíveis por padrão na versão 7.2 do backend do compilador GCC e mais uma que foi acrescido da opção experimental para a linguagem Go denominada de fgo-optimize-allocs. Portanto, cada cromossomo corresponde a uma cadeia binária de tamanho 142 com valores entre 0 e 1 com objetivo de representar se a opção será informado ou não como entrada na compilação do GCC. Na figura 4.2 é possível observar a representação do cromossomo do experimento.

A posição da representação cromossomial são faggressiveloopoptimizations, falignfunctions, falignjumps, falignlabels, falignloops, fassociativemath, fasynchronousunwindtables, fautoincdec, fbranchcountreg, fbranchprobabilities, fbranchtargetloadoptimize, fbranchtargetloadoptimize2, fbtrbbexclusive, fcallersaves, fcombinestackadjustments, fcompareelim, fconservestack, fcpropregisters, fcrossjumping, fcsefollowjumps, fcxfortranrules, fcxlimitedrange, fdce, fdeferpop, fdelayedbranch, fdeletedeadexceptions, fdeletenullpointerchecks, fdevirtualize, fdevirtualizespeculatively, fdse, fearlyinlining, fexceptions, fexpensiveoptimizations, ffinitemathonly, ffloatstore, fforwardpropagate, ffpcontract=off, ffpcontract=on, ffpcontract=fast, ffunctioncse, fgcse, fgcseafterreload, fgcselas, fgcselm, fgcsesm, fgraphite, fgraphiteidentity, fguessbranchprobability, fhandleexceptions, fhoistadjacentloads, fifconversion, fifconversion2, findirectinlining, finline, finlinefunctionscalledonce, finlinesmallfunctions, fipacp, fipacpalignment, fipacpclone, fipaicf, fipaicffunctions, fipaprofile, fipapta, fipapureconst, fipara, fipareference, fipasra, firaalgorithm=priority, firaalgorithm=CB, firahoistpressure, firalooppressure, firaregion=one, firasharesaveslots, firasharespillslots, fisolateerroneous-

Figure 4.2: Representação cromossomial.

0	1	2	...	140	141
1	1	0	...	0	1

Fonte: Elaborado pelo autor.

pathsattribute, fisolateerroneouspathsdereference, fivopts, fjumptables, fkeepgcrootslive, flifetimedse, flifetimedse=1, fliverangeshrinkage, floopnestoptimize, floopparallelizeall, flraremat, fmatherrno, fmodulosched, fmoduloschedallowregmoves, fmoveloopinvariants, fnoncallexceptions, fnothrowopt, fomitframepointer, foptinfo, foptimizesiblingcalls, foptimizestrlen, fpackstruct, fpartialinlining, fpeelloops, fpeephole, fpeephole2, fplt, fpredictivecommoning, fprefetchlooparrays, freciprocalmath, fregstructreturn, frenameregisters, freorderblocks, freorderblocksalgorithm=simple, freorderblocksalgorithm=stc, freorderblocksandpartition, freorderfunctions, freruncseafterloop, freschedulemoduloscheduledloops, froundingmath, frtti, fschedcriticalpathheuristic, fscheddepcountheuristic, fschedgroupheuristic, fschedinterblock, fschedlastinsnheuristic, fschedpressure, fschedrankheuristic, fschedspec, fschedspecinsnheuristic, fschedspecload, fschedspecloaddangerous, fschedstalledinsns, fschedstalledinsnsdep, fschedstalledinsnsdep=1, fschedstalledinsns=1, fsched2usesuperblocks, fschedulefusion, fscheduleinsns, fscheduleinsns2, fsectionanchors, fselschedpipelining, fselschedpipeliningouterloops, fselschedreschedulepipelined, fselectivescheduling, fselectivescheduling2, fshortenums e fgooptimizeallocs.

4.4.2 Função de Avaliação

A avaliação de um cromossomo corresponde ao valor de desempenho encontrado pelo binário após o código fonte ser compilado com as opções ativas sinalizadas nos genes do cromossomo. Esse desempenho é comparado com valores de desempenho encontrado no experimento onde foi utilizado as opções 02 e fgo-optimize-allocs.

4.4.3 Algoritmo

Na construção do algoritmo genético foi utilizada a biblioteca GAGO escrita por Halford (2017) com objetivo de auxiliar na implementação. Para tal, a biblioteca obriga a representação de um objeto e a implementação de métodos que serão utilizados na a execução do algoritmo genético. Esse métodos são chamados de Evaluate, Mutate, e o Crossover que serão explicados a seguir.

O método Evaluate corresponde a função de avaliação da litetura onde é retornado o desempenho do cromossomo. No método é verificado a métrica de corretude e o tempo total de execução do binário para definir o desempenho.

O método Mutate corresponde a ação de mutação do cromossomo. Para o método, ficou estabelecido a mutação de 3 genes e com uma taxa de mutação de 2%.

O método Crossover representa a recombinação dos genes conforme explicado no 2.2.2. Foi utilizado a técnica denominada de crossover de ponto N generalizado (AHMED, 2010) onde um ponto idêntico é escolhido no genoma de cada pai e esses segmentos são trocados.

Com a representação cromossomial e os métodos definidos, pode-se iniciar a implementação do Algoritmo Genético com a biblioteca GAGO. No Pseudocódigo 6, é possível observar os passos realizados pelo algoritmo.

Algorithm 6 Pseudocódigo do Algoritmo Genético

```
1: procedure AlgGen(geracaoInt, matrizPopulacao, desempenho)
2:     modelo ← iniciarModeloGeracional(matrizPopulacao)
3:     modelo.Inicializar()
4:     i ← 0
5:     for (i < geracaoInt)OR(melhorDesempenho > desempenho) do
6:         i = i + 1
7:         modelo.Evoluir()
8:         melhorDesempenho = modelo.Fitness()
9:     end for
10:    return modelo
11: end procedure
```

Capítulo 5

Experimentos Computacionais e Resultados

Esta seção descreve os experimentos computacionais envolvendo os compiladores da linguagem Go, bem como os resultados obtidos.

5.1 Experimento

Esta seção apresenta e discute o experimento, conforme definido na seção 4.1. Para facilitar a replicação dos resultados, um repositório on-line[1] fornece todo o código e dados do experimento.

5.1.1 Preliminar

Cada problema foi implementado em duas versões, uma versão utilizando os recursos sequenciais da linguagem, como por exemplo, utilizando a palavra reservada *for* e uma segunda versão de implementação utilizando a palavra reservada *go* com objetivo de obter métricas com recursos concorrente da linguagem. Pode-se notar na tabela 5.1 que a versão concorrente possuem mais linhas de código que a versão sequencial, entretanto retornando a mesma saída.

Os resultados dos experimentos foram separadas por tabelas e categorizados por compilador e versões dos problemas para uma melhor visualização. As tabelas 5.2, 5.3 e 5.4 fornecem os números absolutos dos experimentos com as versões do código sequencial e as tabelas 5.5, 5.6 e 5.7 fornecem os números absolutos dos experimentos com as versões

[1] https://github.com/ohninar/goexperiment

Table 5.1: Totais de linhas de código dos problemas da literatura por versão de implementação utilizados nos experimentos.

Codenome	**Nome**	**Sequencial - LoC**	**Concorrente - LoC**
randmat	Random number generation	140	159
outer	Outer product	168	192
product	Matrix-vector product	154	175
thresh	Histogram thresholding	170	217
winnow	Weighted point selection	218	306

concorrente. As colunas das tabelas de resultados serão explicadas a seguir para facilitar no entendimento dos resultados. A coluna Tempo total de execução corresponde o tempo total gasto em nanosegundos pelo binário na execução do problema, a coluna Tamanho do binário corresponde ao tamanho do binário em kilobytes gerado pelo compilador, a coluna Pico de uso da memória corresponde ao uso máximo em bytes alocados para objetos na memória heap, a coluna Total de memória disp. S.O. corresponde ao total em bytes disponibilizados previamente pelo Sistema Operacional, a coluna Qtd de obj. alocado na memória é a contagem cumulativa de objetos alocados na memória heap e a coluna Total de uso na stack corresponde ao total em bytes alocados para objetos na memória stack.

Todas as implementações foram compiladas por versões dos compiladores que implementavam a especificação 1.8 da linguagem Go. Na compilação dos problemas com compilador gccgo foi utilizado versão 7.2 e com o gc foi utilizado a versão 1.8.1.

Os experimentos foram realizados utilizando uma CPU AMD PRO de 4 núcleos com 3.0 GHz, com 16 Gb de memória RAM e o sistema operacional GNU/Linux Ubuntu 16.04 LTS.

5.1.2 Configurações dos compiladores

Para o experimento com o compilador Go Compiler, as implementações dos códigos da literatura foram compilados utilizando o comando build com a opção -o, correspondente a saída do arquivo binário gerado pelo compilador. Não foi possível introduzir opção de otimização como entrada para o compilador pelo motivo do mesmo não possuir entrada manuais de otimização.

No experimento do compilador gccgo com grupo de opções padrões, as implementações dos códigos da literatura foram compilados utilizando a opção -o, referente a saída do arquivo binário gerado pelo compilador e também foi informado manualmente as opções de otimizações -O2, -O3 e -fgo-optimize-allocs. Já no segundo experimento com o compilador

gccgo, as implementações dos algoritmos da literatura foram compilados utilizando as opção -o, referente a saída do arquivo binário gerado pelo compilador e o subconjunto ótimo de opções de otimizações que foram selecionados após a execução do algoritmo genético.

5.1.3 Execução do Experimento

Na primeira parte do experimento, os problemas foram implementados utilizando o recurso de concorrência existente na linguagem Go com o propósito de extrair dados de performance na programação paralela. Em outro momento, os problemas foram implementados utilizando os recursos sequenciais da linguagem com o objetivo principal de investigar se haveria alguma diferença nas métricas de performance entre os compiladores ao comparar com os dados colhidos do experimento anterior.

Para colher os dados das métricas, foi implementando uma aplicação para realizar o benchmark com propósito de obter os dados de desempenho de um binário. Para isso, a aplicação de benchmark executava uma quantidade pré-estabelecido de vezes com um conjunto de entrada e, no fim do processamento, era retornado como saída um conjunto de dados sobre o desempenho do binário conforme a lista 5.1.

Listing 5.1: Código do benchmark

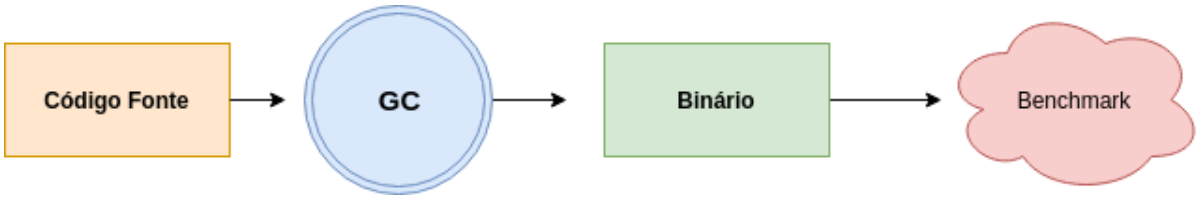
```
1  package main
2
3  import (
4      "exec"
5      "fmt"
6      "time"
7      "os"
8  )
9
10 const GENERATIONS = 1000
11
12 func main() {
13     var resultado float64
14
15     for i := 0; i < GENERATIONS; i++ {
16         resultado += runner()
17     }
18
19     fmt.Println(resultado / GENERATIONS)
```

```
20 }
21
22 func runner() float64 {
23     var elapsed time.Duration
24     start := time.Now()
25
26     out, err := exec.Command("bash", "-c", "main-gccgo < main.in").Output()
27     if err != nil {
28         os.Exit(1)
29     }
30     elapsed = time.Since(start)
31
32     if string(out) != corretudeOut {
33         os.Exit(1)
34     }
35
36     return elapsed.Seconds()
37 }
```

Para cada instância de execução do binário, foram utilizados uma mesma entrada para que fosse possível validar a saída com valores já validados. Foram executados exaustivamente por 10 mil vezes cada arquivo binário gerado pelo compilador e extraído uma média no tempo do processamento, o tamanho do binário e o uso de memória.

Os experimentos que tinham como alvo o compilador Go Compiler seguiu os seguintes passos: a implementação do problema era compilado e, como entrada para o benchmark, o binário era executado por 1000 vezes em 10 momentos distintos e suas métricas eram registrados para análise no momento posterior como ilustrado na Figura 5.1.

Figure 5.1: Fluxo do experimento para o GC.



Fonte: Elaborado pelo autor.

Para o gccgo foi necessário implementar um software com técnicas de algoritmo genético com objetivo de encontrar um conjunto de opções mais performático do que utilizado por padrão na compilação para cada problema. Isso foi necessário pelo fato do GCC permitir como entrada as opções de otimizações para a compilação e a forma

de selecionar essas opções podem influenciar na performance do problema. Para tal, foi utilizado a biblioteca GAGO escrita por Halford (2017) com objetivo de auxiliar no desenvolvimento do algoritmo genético. O método evolução ficou responsável em retornar o calculo do tempo de execução do binário que foi compilado com o subconjunto de opções conforme ilustrado na lista 4.1.

Listing 5.2: Método Evolução

```
1  package algGen
2
3  import (
4      "log"
5      "math/rand"
6      "os"
7      "os/exec"
8      "strconv"
9      "strings"
10     "time"
11
12     "github.com/MaxHalford/gago"
13     uuid "github.com/nu7hatch/gouuid"
14 )
15
16 //NumFlags ...
17 const (
18     NumFlags     = 141
19     NoCorretude = 100000000
20 )
21
22 //Flags ...
23 type Flags []int
24
25 //Evaluate ...
26 func (f Flags) Evaluate() float64 {
27     u, _ := uuid.NewV4()
28     filename := "output/g" + u.String()
29
30     _ = getTimeCompilation(f, filename)
31
32     _ := getLengthCompilation(f, filename)
33
34     timeExec := getTimeExec(filename)
35     if timeExec == NoCorretude {
```

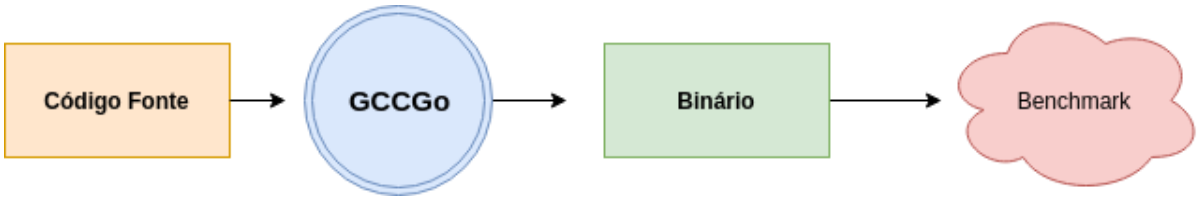
```
36          return NoCorretude
37      }
38
39      return lenComp
40  }
41
42  func getTimeCompilation(f Flags, filename string) float64 {
43      compileTarget := "/usr/bin/gccgo-6"
44      codeTarget := "alg/outer/expertseq/main.go"
45
46      cmd := exec.Command(compileTarget, codeTarget)
47      cmd.Env = os.Environ()
48
49      for c, o := range f {
50          if o == 1 {
51              cmd.Args = append(cmd.Args, Opt[c])
52          }
53      }
54      cmd.Args = append(cmd.Args, "-o")
55      cmd.Args = append(cmd.Args, filename)
56
57      start := time.Now()
58
59      err := cmd.Start()
60      if err != nil {
61          return 0
62      }
63
64      if err := cmd.Wait(); err != nil {
65          return 0
66      }
67
68      elapsed := time.Since(start)
69
70      return float64(elapsed.Seconds())
71  }
72
73  func getLengthCompilation(f Flags, filename string) float64 {
74      out, err := exec.Command("wc", "-c", filename).Output()
75      if err != nil {
76          log.Fatal(err)
77      }
78
```

```
79      lengthString := strings.Split(string(out), " ")[0]
80
81      lengthInt, _ := strconv.Atoi(lengthString)
82
83      return float64(lengthInt)
84  }
85
86  func getTimeExec(filename string) float64 {
87      var elapsed time.Duration
88      start := time.Now()
89
90      cmd := exec.Command("bash", "-c", filename+" <
    alg/outer/expertseq/main.in")
91
92      out, err := cmd.Output()
93      if err != nil {
94          log.Println(err.Error(), filename, out)
95          return NoCorretude
96      }
97
98      elapsed = time.Since(start)
99
100     if string(out) == OutOuter {
101         log.Println("eh igual", filename)
102         return float64(elapsed.Seconds())
103     }
104
105     return NoCorretude
106 }
107
108 func (f Flags) Mutate(rng *rand.Rand) {
109     gago.MutPermuteInt(f, 3, rng)
110 }
111
112 func (f Flags) Crossover(Y gago.Genome, rng *rand.Rand) {
113     gago.CrossGNXInt(f, Y.(Flags), 2, rng)
114 }
115
116 func (f Flags) Clone() gago.Genome {
117     var ff = make(Flags, len(f))
118     copy(ff, f)
119     return ff
120 }
```

```
121
122 func MakeBoard(rng *rand.Rand) gago.Genome {
123     var flags = make(Flags, NumFlags)
124     for i, flag := range rng.Perm(NumFlags) {
125         if flag%2 == 0 {
126             flags[i] = 0
127         } else {
128             flags[i] = 1
129         }
130     }
131     return gago.Genome(flags)
132 }
```

Os experimentos que tinham como alvo o compilador gccgo seguiram dois procedimentos, no primeiro procedimento que foi denominado de gccgo, o problema era compilado informando opções de uso padrão de otimização e, como entrada para o sistema de apoio, o binário era executado por 1000 vezes em 10 momentos distintos onde suas métricas eram registradas para serem analisadas em um momento posterior como ilustrado na figura 5.2.

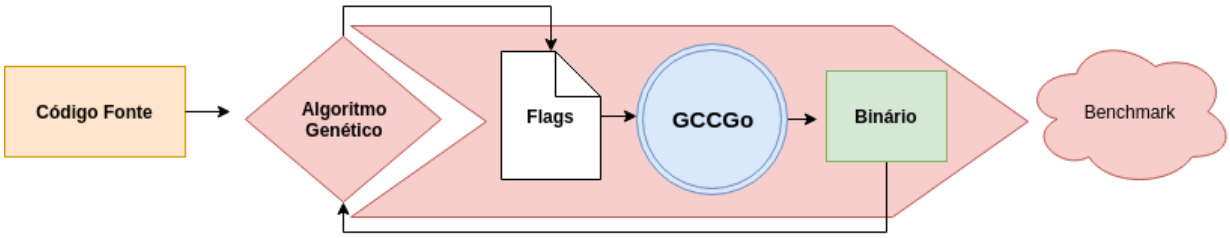
Figure 5.2: Fluxo do experimento para o gccgo.



Fonte: Elaborado pelo autor.

No segundo procedimento que foi denominado de gccgo opt, o problema era informado para o software de algoritmo genético com um conjunto de opções de otimizações disponíveis e, como saída, era retornado um subconjunto de opções que eram mais performático na métrica tempo total de execução de que no primeiro procedimento. Por fim, o problema era compilado com o subconjunto de opções de otimização e, como entrada para o benchmark, o binário era executado por 1000 vezes em 10 momentos distintos onde suas métricas eram registrados para serem analisados em um momento posterior como ilustrado na Figura 5.3.

Figure 5.3: Fluxo do experimento para o gccgo com opções de otimização.



Fonte: Elaborado pelo autor.

5.2 Análise dos Resultados

Os resultados foram categorizados e analisados separadamente pela forma de implementação dos algoritmos e comparado entre os compiladores. A primeira análise foram os resultados obtidos no experimento que utilizou a implementação sequencial dos problemas e a segunda análise foram os resultados no experimento com as implementações na versão concorrente.

5.2.1 Corretude

A corretude do programa é o fator mais importante para garantir igualitariedade no experimento entre os compiladores. Ele foi tratado como um pré-requisito para a avaliação de desempenho e, para isso, catalogamos os dados apenas após validar se o programa solucionou o problema corretamente.

O método utilizado para validação da corretude foi os testes unitários que foram executados após a execução dos programas. Tanto o gc quanto o gccgo compilaram programas que executaram corretamente todos os problemas em ambas versões. Portanto, essa métrica serviu como controle para evitar a utilização de dados de programas inválidos.

5.2.2 Resultados na Implementação Sequencial

Nessa seção será descrito os resultados encontrados nos experimentos que utilizaram as implementações dos problemas na versão sequencial. Os resultados foram divididos por compiladores nas tabelas 5.2, 5.3 e 5.4.

Table 5.2: gc 1.8 - Experimento Sequencial

Experimento Sequencial	gc 1.8					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.0047399310	1.745.918	98.136	2.134.016	316	294.912
Outer	0.0061860420	1.754.320	100.728	1.806.336	459	294.912
Product	0.0059100990	1.754.393	98.984	2.134.016	380	294.912
Thresh	0.0058217605	1.754.188	98.072	1.871.872	328	294.912
Winnow	0.0058583125	1.754.597	98.824	2.134.016	373	294.912

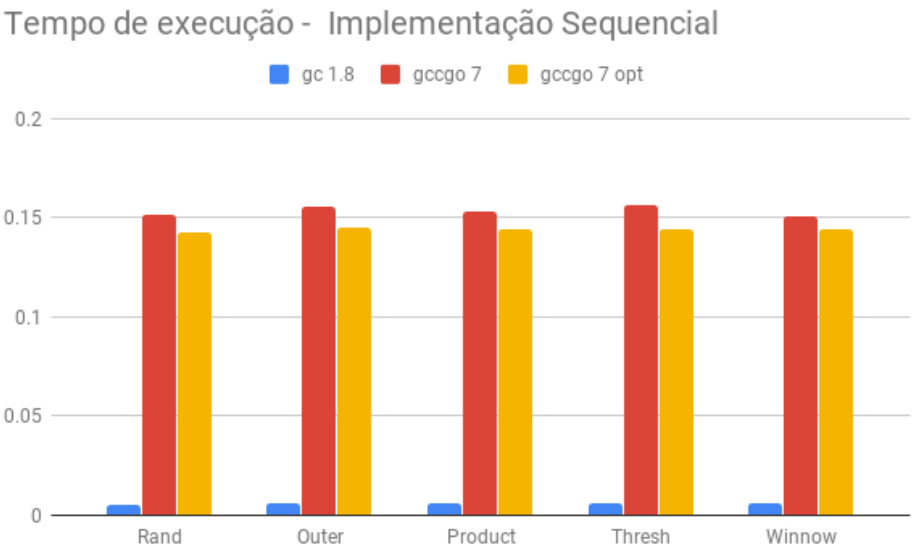
Table 5.3: gccgo 7 - Experimento Sequencial

Experimento Sequencial	gccgo 7					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.1511502237	49.616	351.239	3.938.552	6049	0
Outer	0.1553079722	50.083	356.283	3.938.552	6631	0
Product	0.1527603806	40.277	360.481	3.938.552	5859	0
Thresh	0.1559751145	55.130	358.450	3.938.552	5649	0
Winnow	0.1510133295	70.134	376.184	3.938.552	5749	0

Table 5.4: gccgo 7 opt - Experimento Sequencial

Experimento Sequencial	gccgo 7 opt					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.1423260110	45.856	349.144	3.938.552	6043	0
Outer	0.1446070505	45.864	353.144	3.938.552	6625	0
Product	0.1441271635	39.360	353.392	3.938.552	5846	0
Thresh	0.1440213430	51.960	342.032	3.938.552	5633	0
Winnow	0.1440966885	65.144	352.992	3.938.552	5741	0

Figure 5.4: Comparativo do tempo de execução no experimento sequencial.



Fonte: Elaborado pelo autor.

5.2.2.1 Tempo de Execução

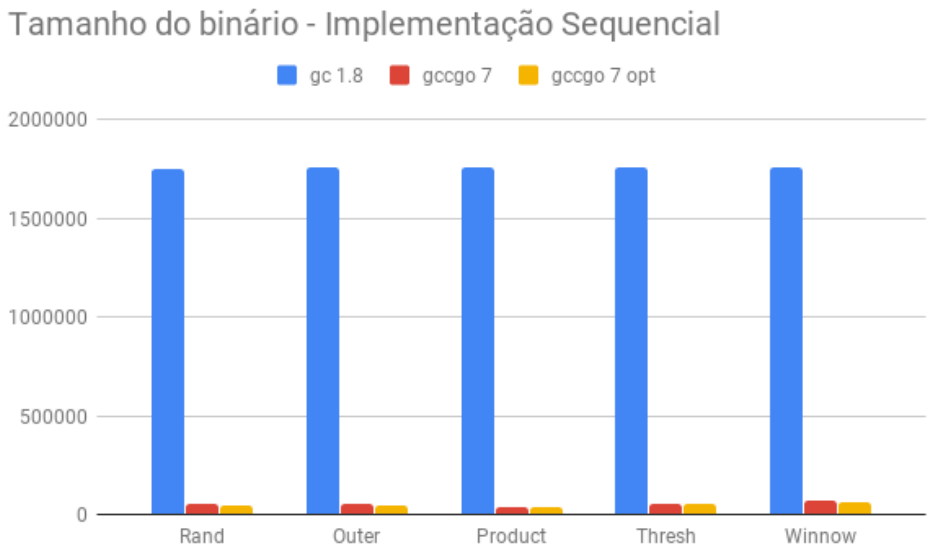
Seguindo o princípio de utilizar os mesmos códigos fonte como entrada em ambos os compiladores e que os dados dos resultados utilizados no comparativo foram validados com sucesso a sua corretude e por consequência a métrica do tempo total de execução usou dados válidos. Em outras palavras, foi apenas contabilizado o tempo total de execução de binários válidos enquanto que as métricas de binários inválidos foram descartados independente dos valores dos resultados.

A figura 5.4 mostra o comparativo do tempo total de execução de ambos os compiladores por implementações dos problemas na versão sequencial. Logo, ao visualizar os gráficos, é perceptível a diferença de performance entre os compiladores.

Os programas compilados com gccgo levaram mais tempo para executar os problemas quando comparado com os resultados obtidos dos programas compilados com gc. No problema Rand, o gc foi em média 30 vezes mais performático enquanto que no problema Outer foi em média 23 vezes. Nos problemas Product, Thresh e Winnow, o gc foi em média 24 vezes mais performático que o gccgo.

No total, a diferença de performance do compilador gc foi em média 25 vezes mais performático do que o gccgo na métrica de tempo total de execução para os problemas implementados na versão sequencial.

Figure 5.5: Comparativo do tamanho do binário no experimento sequencial.



Fonte: Elaborado pelo autor.

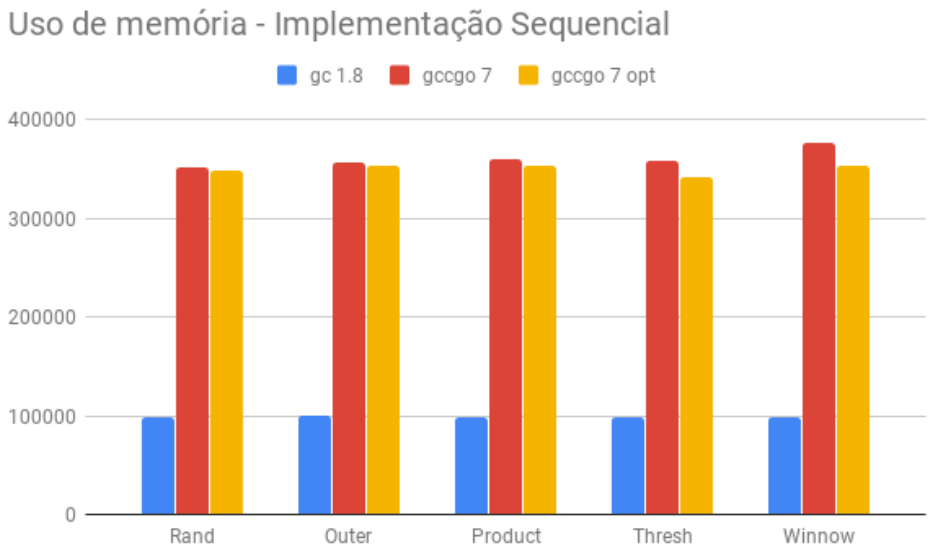
5.2.2.2 Tamanho do Binário

Após a compilação e execução, foram medidos o tamanho dos binários gerados em cada compilação nos problemas sequencias. A principal motivação para coleta desta informação é que, dependendo das limitações de recursos da arquitetura alvo em que será executado o binário, o desenvolvedor precisará selecionar um compilador que retorne como saída um menor binário possível.

Nesse quesito o compilador gccgo apresentou o menor tamanho de binário em comparação com o compilador gc nas implementações dos problemas na versão sequencial. No problema Rand o gccgo gerou em média 38 vezes um binário de tamanho menor quando comparado ao binário gerado pelo gc do mesmo código fonte do problema. Esse mesmo resultado foi encontrado para o problema Outer, entretanto no problema Product foi encontrado a maior diferença entre os binários com uma média de 44,57 vezes. Nos problemas Thresh e Winnow, o gccgo obteve um binário menor 33,76 e 26,93 em média respectivamente.

Na figura 5.5 mostra o comparativo do tamanho do binário gerado após a compilação nas implementações dos problemas nas versões sequencial. A diferença média dos tamanhos dos binários são 36,30 vezes quando comparado com o tamanho dos binários compilados no gc.

Figure 5.6: Comparativo do uso de memória no experimento sequencial.



Fonte: Elaborado pelo autor.

5.2.2.3 Uso de Memória

O uso de memória de uma aplicação é um ponto fundamental principalmente quando a arquitetura alvo possui limitações como por exemplo os sistemas embarcados. Por isso é de extrema importância para o desenvolvedor a informação do comportamento da aplicação poderá ser diferente dependendo de qual compilador foi utilizado.

Nesse experimento, o compilador gccgo apresentou o maior uso de memória Heap em comparação com o compilador gc, totalizando o uso em média de 3,5 vezes. A figura 5.6 mostra o comparativo do uso de memória após a execução dos programas até o fim do processamento nas implementações dos problemas na versão sequencial.

O gccgo utilizou a memória heap 3,53 vezes a mais no problema Rand e no problema Outer, foi encontrado um uso maior de 3,50 quando comparado com os resultados gc. Nos problemas Product, Thresh e Winnow, o gccgo também utilizou memória a mais nas execuções do experimento com 3.57, 3.48 e 3,57 em média respectivamente.

Além disso, é importante destacar que o gccgo alocou mais objetos na memória no experimento, totalizando em média 16,29 mais objetos do que o gc. No problema Rand, foi encontrado a maior diferença entre os compiladores com uma média de 19,12 vezes e no problema Outer foi encontrado a menor diferença com uma média de 14,43 vezes. Nos demais problemas, foram encontrados uma média de 15,38 para o problema Product, de 17,17 para o problema Thresh e de 15,39 para o problema Winnow.

Table 5.5: gc 1.8 - Experimento Concorrente

Experimento concorrente	gc 1.8					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.0048455930	1.754.238	92.728	2.134.016	329	294.912
Outer	0.0063504000	1.754.444	94.536	2.134.016	486	327.680
Product	0.0060089165	1.754.521	92.856	2.396.160	397	294.912
Thresh	0.0059790705	1.758.494	95.720	3.574.008	414	327.680
Winnow	0.0061508630	1.767.178	94.312	3.149.824	403	327.680

Table 5.6: gccgo 7 - Experimento Concorrente

Experimento Concorrente	gccgo 7					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.154466469	58.328	360.380	3.938.552	5886	0
Outer	0.156785041	61.301	374.880	3.938.552	6772	0
Product	0.150728096	53.830	375.984	3.938.552	5869	0
Thresh	0.150557916	72.718	365.164	3.938.552	5668	0
Winnow	0.150194440	89.052	388.158	3.938.552	5779	0

5.2.3 Resultados na implementação concorrente

Nessa seção será descrito os resultados encontrados nos experimentos que utilizaram as implementações dos problemas na versão concorrente. Os resultados foram divididos por compiladores nas tabelas 5.5, 5.6 e 5.7.

5.2.3.1 Tempo de Execução

A figura 5.7 mostra o comparativo do tempo de execução de ambos os compiladores para as implementação dos problemas na versão concorrente.

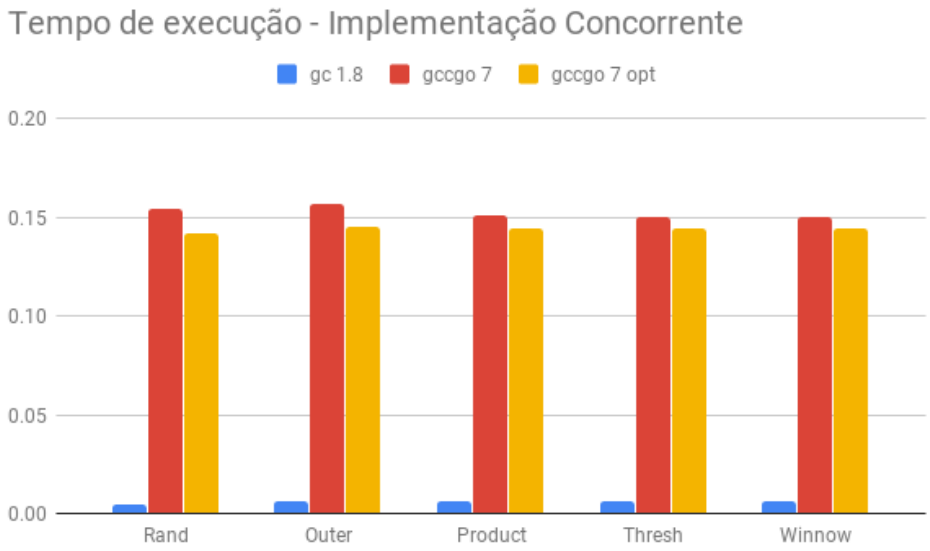
Nota-se que o mesmo desempenho encontrado nos resultados de performance para os problemas na versão sequencial foram semelhantes para os problemas implementados na versão concorrente. Os programas compilados com gccgo também levaram mais tempo para executar em todos os problemas concorrentes quando comparado com os resultados dos programas compilados com gc.

No problema Rand, o gc foi em média 29,35 vezes mais performático enquanto que no

Table 5.7: gccgo 7 opt - Experimento Concorrente

Experimento Sequencial	gccgo 7 opt					
	Tempo total de execução (ns)	Tamanho do binário (byte)	Pico de uso de memória (kbyte)	Total de memória disp. S.O. (kbyte)	Qtd de Obj. alocado na memória	Total de uso da Stack (kbyte)
Rand	0.1422343180	57.544	346.616	3.938.552	5877	0
Outer	0.1451699900	58.320	367.560	3.938.552	6768	0
Product	0.1445056810	51.816	358.416	3.938.552	5863	0
Thresh	0.1441937790	69.920	354.952	3.938.552	5664	0
Winnow	0.1446193650	84.000	370.800	3.938.552	5778	0

Figure 5.7: Comparativo do tempo de execução no experimento concorrente.

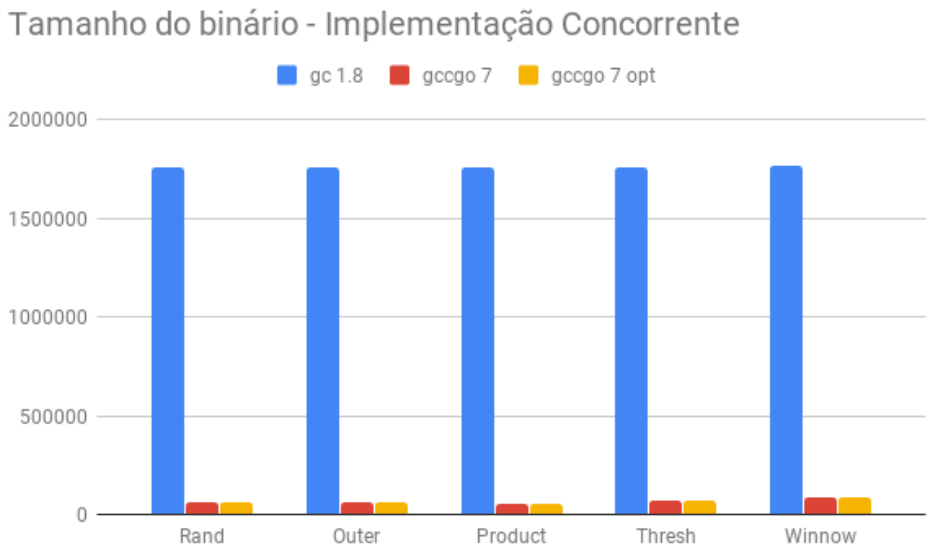


Fonte: Elaborado pelo autor.

problema Outer foi em média 22,85 vezes. Nos problemas Product, Thresh e Winnow, o gc foi respectivamente em média 24,04, 24,11 e 23,51 vezes mais performático que o gccgo.

No total, a diferença de performance do compilador gc foi em média 24,7 vezes mais performático do que o gccgo na métrica de tempo total de execução para os problemas implementados na versão concorrente. Ao comparar a média total da métrica entre as versões de implementação percebe-se uma melhora no desempenho do gccgo, entretanto continua uma diferença considerável de performance entre os dois compiladores.

Figure 5.8: Comparativo do tamanho do binário no experimento concorrente.



Fonte: Elaborado pelo autor.

5.2.3.2 Tamanho do Binário

Nesse quesito com implementações concorrente, o compilador gccgo também apresentou o menor tamanho de binário em comparação com o compilador gc. Como pode-se visualizar na figura 5.8, a média de diferença de tamanho foi de 28,12 vezes entre gc para o gccgo.

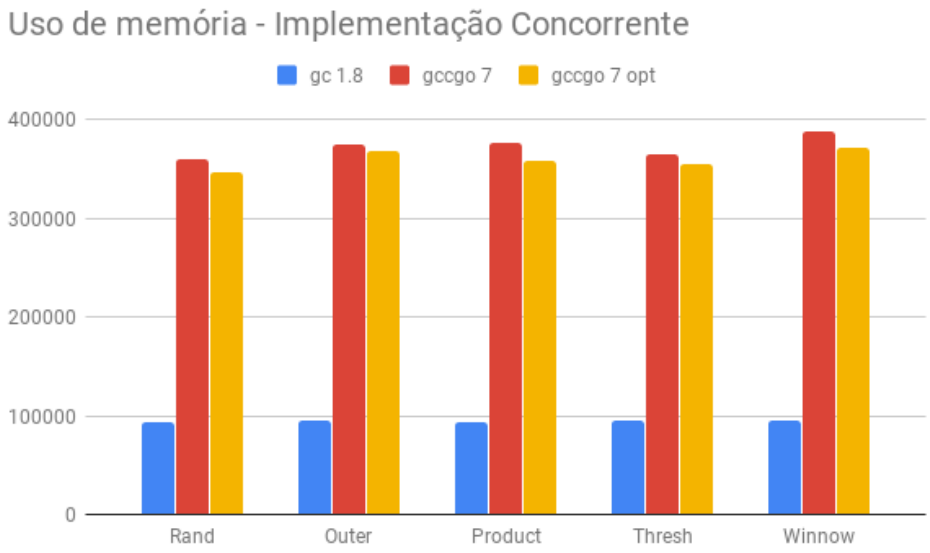
Nos problemas Rand e Outer, o gccgo gerou em média um binário 30 vezes menor que o binário compilado no gc. No problema Product, foi encontrado a maior diferença entre os binários com uma média de 33,86 vezes. Em contrapartida, os resultados encontrado para os problemas Thresh e Winnow foram os menores do experimento com 33,76 e 26,93 em média respectivamente.

5.2.3.3 Uso de Memória

A figura 5.9 exibe o comparativo do uso de memória após a execução dos programas até o fim do processamento nas implementações dos problemas na versão concorrente, onde é possível perceber que o gccgo também utilizou mais memória Heap como no experimento anterior.

O compilador gc apresentou o menor uso de memória Heap em todas as implementações em comparação com o compilador gccgo. A média de diferença do uso foi de 3,81 vezes entre os binários compilados por gccgo e o gc.

Figure 5.9: Comparativo do uso de memória no experimento concorrente.



Fonte: Elaborado pelo autor.

O gccgo utilizou a memória heap 3,73 vezes a mais no problema Rand enquanto que no problema Outer, foi encontrado um uso maior de 3,88 quando comparado com os resultados do gc. Nos problemas Product, Thresh e Winnow, o gccgo também utilizou memória a mais nas execuções do experimento com 3,85, 3,70 e 3,93 em média respectivamente.

Destacamos também que foi encontrado o mesmo comportamento já apresentado no experimento anterior, mas com uma média menor entre os experimentos. Nesse experimento, o gccgo alocou mais objetos na memória na média de 14,91 mais objetos do que o gc. No problema Rand, foi encontrado a maior diferença entre os compiladores com uma média de 17,83 vezes e no problema Outer foi encontrado a média de 13,92 vezes. Nos demais problemas, foram encontrados uma média de 14,76 para o problema Product, a menor média no experimento de 13,68 para o problema Thresh e de 15,39 para o problema Winnow.

5.2.4 Discussão

Os dados dos experimentos foram coletados para viabilizar a comparação de performance entre os binários resultantes dos compiladores gc, gccgo e o gccgo opt. Esses dados relatam um paradoxo na afirmação de Taylor (2012) ao afirmar que o gccgo possa ser 30% mais eficiente que gc em determinados casos por ter mais de uma centena de opções de otimizações. Desse modo, foi realizado um estudo sobre as otimizações realizadas pelo gc com o intuito de identificar o principal motivo do desempenho nos experimentos.

Pode-se perceber nos experimentos de tempo total de execução e no uso de memória que o compilador gc foi mais performático tanto nas versões sequenciais e nas concorrentes. O principal motivo para esses resultados estão na otimização de *escape analysis* e do *inlining* utilizado por padrão na compilação do binário no gc. Essas otimizações são responsáveis principalmente por decidir se o compilador irá alocar o recurso na área de memória denominada de Stack ao invés de alocar na área de memória denominada de Heap. É possível visualizar essas decisões de otimizações do compilador através do comando -gcflags "-m -m" no momento da compilação conforme mostrado na listagem 5.3 no problema rand sequencial.

Listing 5.3: Listagem das otimizações do gc no problema rand sequencial

```
1  go build -gcflags "-m -m" target/randmat/expertseq/main.go
2
3  # command-line-arguments
4  ./main.go:27:6: can inline NewByteMatrix as: func(int, int) *ByteMatrix {
5  return &ByteMatrix literal }
6  ./main.go:31:6: can inline (*ByteMatrix).Row as: method(*ByteMatrix)
7  func(int) []byte { return m.array[i * m.Cols:(i + 1) * m.Cols] }
8  ./main.go:44:6: cannot inline randmat: unhandled op FOR
9  ./main.go:45:25: inlining call to NewByteMatrix func(int, int) *ByteMatrix {
10 return &ByteMatrix literal }
11 ./main.go:50:20: inlining call to (*ByteMatrix).Row method(*ByteMatrix)
12 func(int) []byte { return m.array[i * m.Cols:(i + 1) * m.Cols] }
13 ./main.go:88:6: cannot inline SaveMemProfile: non-leaf function
14 ./main.go:60:6: cannot inline main: non-leaf function
15 ./main.go:74:21: inlining call to (*ByteMatrix).Row method(*ByteMatrix)
16 func(int) []byte { return m.array[i * m.Cols:(i + 1) * m.Cols] }
17 ./main.go:66:11: &nrows escapes to heap
18 ./main.go:66:11:     from ... argument (arg to ...) at ./main.go:66:10
19 ./main.go:66:11:     from *(... argument) (indirection) at ./main.go:66:10
20 ./main.go:66:11:     from ... argument (passed to call[argument content
21 escapes]) at ./main.go:66:10
22 ./main.go:66:11: &nrows escapes to heap
23 ./main.go:66:11:     from &nrows (interface-converted) at ./main.go:66:11
24 ./main.go:66:11:     from ... argument (arg to ...) at ./main.go:66:10
25 ./main.go:66:11:     from *(... argument) (indirection) at ./main.go:66:10
26 ./main.go:66:11:     from ... argument (passed to call[argument content
27 escapes]) at ./main.go:66:10
28 ./main.go:63:6: moved to heap: nrows
29 ./main.go:67:11: &ncols escapes to heap
30 ./main.go:67:11:     from ... argument (arg to ...) at ./main.go:67:10
```

```
./main.go:67:11:     from *(... argument) (indirection) at ./main.go:67:10
./main.go:67:11:     from ... argument (passed to call[argument content
escapes]) at ./main.go:67:10
./main.go:67:11: &ncols escapes to heap
./main.go:67:11:     from &ncols (interface-converted) at ./main.go:67:11
./main.go:67:11:     from ... argument (arg to ...) at ./main.go:67:10
./main.go:67:11:     from *(... argument) (indirection) at ./main.go:67:10
./main.go:67:11:     from ... argument (passed to call[argument content
escapes]) at ./main.go:67:10
./main.go:63:6: moved to heap: ncols
./main.go:68:11: &seed escapes to heap
./main.go:68:11:     from ... argument (arg to ...) at ./main.go:68:10
./main.go:68:11:     from *(... argument) (indirection) at ./main.go:68:10
./main.go:68:11:     from ... argument (passed to call[argument content
escapes]) at ./main.go:68:10
./main.go:68:11: &seed escapes to heap
./main.go:68:11:     from &seed (interface-converted) at ./main.go:68:11
./main.go:68:11:     from ... argument (arg to ...) at ./main.go:68:10
./main.go:68:11:     from *(... argument) (indirection) at ./main.go:68:10
./main.go:68:11:     from ... argument (passed to call[argument content
escapes]) at ./main.go:68:10
./main.go:64:6: moved to heap: seed
./main.go:76:26: row[j] escapes to heap
./main.go:76:26:     from ... argument (arg to ...) at ./main.go:76:15
./main.go:76:26:     from *(... argument) (indirection) at ./main.go:76:15
./main.go:76:26:     from ... argument (passed to call[argument content
escapes]) at ./main.go:76:15
./main.go:66:10: main ... argument does not escape
./main.go:67:10: main ... argument does not escape
./main.go:68:10: main ... argument does not escape
./main.go:76:15: main ... argument does not escape
```

Essa decisão é importante pelo motivo do custo de alocação e desalocação da Stack por ser menor quando comparado com a alocação e desalocação na memória da Heap. O uso da area Heap necessita da ação do Garbage Collector para realizar a limpeza de tempos em tempos e com isso acarretando um tempo extra de latência na execução do binário. Por essas otimizações na compilação, o gc é mais performático por acessar menos a heap ganhando assim um bom tempo no desempenho de acesso na memória ao contrário do que foi coletado no experimento com o gccgo. Além disso, é importante destacar que a decisão para o uso da memória por parte do escape analysis está relacionado a condição do compartilhamento do acesso do valor para outras funções e não por um tipo de valor

específico da linguagem ou por uma chamada especifica como por exemplo a palavra reservada new de outras linguagens.

O gccgo possui uma opção de otimização denominada de -fgo-optimize-allocs com o objetivo similar da otimização de escape analysis e inlining do gc, entretanto não foi coletado melhorias na alocação de memória e na performance do binário quando utilizado a opção nos experimentos.

Outra métrica abordada no experimento foi o tamanho do binário no qual o gccgo obteve o menor tamanho quando comparado com o binário resultante do gc. Essa diferença de tamanho é motivado pelo fato do gccgo utilizar no ato da compilação a ligação compartilhada das bibliotecas e também por não implementar toda runtime do Go como visualizado na listagem 5.4 no problema rand sequencial. O gc por usa vez liga estaticamente as bibliotecas e que, por consequência, torna-se um binário independente de bibliotecas pré-instaladas no sistema operacional alvo, como visualizado na listagem 5.5 no problema rand sequencial.

Listing 5.4: Listagem das bibliotecas compartilhadas na compilação gccgo

```
1  ldd -v maingccgo
2    linux-vdso.so.1 =>  (0x00007ffd8d5f0000)
3    libgo.so.9 => /usr/lib/x86_64-linux-gnu/libgo.so.9 (0x00007f680b466000)
4    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f680b24e000)
5    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f680ae7e000)
6    /lib64/ld-linux-x86-64.so.2 (0x000055b17e80b000)
7    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f680ac5e000)
8    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f680a94e000)
9
10   Version information:
11     ./maingccgo:
12       libgcc_s.so.1 (GCC_3.3.1) => /lib/x86_64-linux-gnu/libgcc_s.so.1
13       libgcc_s.so.1 (GCC_3.0) => /lib/x86_64-linux-gnu/libgcc_s.so.1
14       ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
15       libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
16     /usr/lib/x86_64-linux-gnu/libgo.so.9:
17       ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
18       libpthread.so.0 (GLIBC_2.3.2) => /lib/x86_64-linux-gnu/libpthread.so.0
19       libpthread.so.0 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libpthread.so.0
20       libgcc_s.so.1 (GCC_3.3.1) => /lib/x86_64-linux-gnu/libgcc_s.so.1
21       libgcc_s.so.1 (GCC_3.0) => /lib/x86_64-linux-gnu/libgcc_s.so.1
22       libgcc_s.so.1 (GCC_3.3) => /lib/x86_64-linux-gnu/libgcc_s.so.1
23       libgcc_s.so.1 (GCC_4.2.0) => /lib/x86_64-linux-gnu/libgcc_s.so.1
```

```
24        libm.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libm.so.6
25        libc.so.6 (GLIBC_2.7) => /lib/x86_64-linux-gnu/libc.so.6
26        libc.so.6 (GLIBC_2.6) => /lib/x86_64-linux-gnu/libc.so.6
27        libc.so.6 (GLIBC_2.9) => /lib/x86_64-linux-gnu/libc.so.6
28        libc.so.6 (GLIBC_2.10) => /lib/x86_64-linux-gnu/libc.so.6
29        libc.so.6 (GLIBC_2.17) => /lib/x86_64-linux-gnu/libc.so.6
30        libc.so.6 (GLIBC_2.3.4) => /lib/x86_64-linux-gnu/libc.so.6
31        libc.so.6 (GLIBC_2.5) => /lib/x86_64-linux-gnu/libc.so.6
32        libc.so.6 (GLIBC_2.3) => /lib/x86_64-linux-gnu/libc.so.6
33        libc.so.6 (GLIBC_2.4) => /lib/x86_64-linux-gnu/libc.so.6
34        libc.so.6 (GLIBC_2.14) => /lib/x86_64-linux-gnu/libc.so.6
35        libc.so.6 (GLIBC_2.3.2) => /lib/x86_64-linux-gnu/libc.so.6
36        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
37      /lib/x86_64-linux-gnu/libgcc_s.so.1:
38        libc.so.6 (GLIBC_2.14) => /lib/x86_64-linux-gnu/libc.so.6
39        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
40      /lib/x86_64-linux-gnu/libc.so.6:
41        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
42        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
43      /lib/x86_64-linux-gnu/libpthread.so.0:
44        ld-linux-x86-64.so.2 (GLIBC_2.2.5) => /lib64/ld-linux-x86-64.so.2
45        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
46        libc.so.6 (GLIBC_2.14) => /lib/x86_64-linux-gnu/libc.so.6
47        libc.so.6 (GLIBC_2.3.2) => /lib/x86_64-linux-gnu/libc.so.6
48        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
49        libc.so.6 (GLIBC_PRIVATE) => /lib/x86_64-linux-gnu/libc.so.6
50      /lib/x86_64-linux-gnu/libm.so.6:
51        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
52        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
53        libc.so.6 (GLIBC_PRIVATE) => /lib/x86_64-linux-gnu/libc.so.6
```

Os códigos compilados do gccgo poderiam ser úteis em dispositivos embarcados ou microcontroladores onde o hardware possui menos espaço em disco e que os sistemas operacionais possuem bibliotecas instaladas para facilitar o compartilhamento de dependências entre os binários.

Listing 5.5: Listagem das bibliotecas compartilhadas na compilação gc

```
1 ldd -v maingc
2     not a dynamic executable
```

Capítulo 6

Considerações Finais

Com a realização deste trabalho, foi possível apresentar um experimento com uso de técnicas de otimização capaz de auxiliar na analise de performance entre dois compiladores da linguagem Go. Com o fim da realização deste experimento, foi possível destacar quatro principais contribuições do trabalho.

Em primeiro lugar, comprovamos que o compilador gc é o responsável por compilar o binário de maneira mais eficiente do que o gccgo nos quesitos de tempo de processamento e no uso de memória. Essa performance não foi ultrapassada mesmo quando utilizado um subconjunto ótimo de opções de otimizações disponível no gccgo. Com relação a técnica de otimização, o algoritmo genético foi capaz de encontrar um subconjunto de opção de otimização mais eficiente quando comparado com as opções de otimizações de uso padrão do compilador gccgo, como por exemplo as opções -O2 -fgo-optimize-allocs.

Segundo, ao aplicar o experimento no gc e no gccgo, foi possível realizar um estudo comparativo detalhado de performance e desempenho entre os compiladores da linguagem Go que, não sendo encontrados na literatura pesquisada para este trabalho até a data atual, podem ser úteis para experimentos futuros.

Terceiro, para auxiliar nos experimentos deste trabalho foram construídos ferramentas e implementações dos problemas da literatura na linguagem Go que foram disponibilizados sem custo na internet com objetivo de permitir a sua utilização para trabalhos futuros. Por fim, nossos resultados podem servir como material de pesquisa para os desenvolvedores afim de minimizar as dúvidas na escolha do compilador ideal para compilar os códigos fonte escrito na linguagem Go.

Tendo em vista que foi exposto nesse estudo, podemos acrescentar duas propostas de trabalho futuro. A primeira diz a respeito a realização do experimento com outras

arquitetura, como por exemplo as arquiteturas x86 ou ARM. A segunda se refere ao uso de um conjunto de problemas com foco no uso exaustivo de memória que, após o fim dos experimentos deste trabalho, mostrou-se como sendo o diferencial nos resultados entre os compiladores estudados neste trabalho.

Referências Bibliográficas

AHMED, Z. H. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. 2010.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. [S.l.]: Addison-wesley Reading, 2007. v. 2.

ASANOVIC, K. et al. *The landscape of parallel computing research: A view from berkeley*. [S.l.], 2006.

ASANOVIC, K. et al. The parallel computing laboratory at uc berkeley: A research agenda based on the berkeley view. *EECS Department, University of California, Berkeley, Tech. Rep*, 2008.

ASANOVIC, K. et al. A view of the parallel computing landscape. *Communications of the ACM*, ACM, v. 52, n. 10, p. 56–67, 2009.

BUTTERFIELD, E. H. Fog computing with go: A comparative study. 2016.

COOPER, K.; TORCZON, L. *Engineering a compiler*. [S.l.]: Elsevier, 2011.

COSTA, R. D. O. *Algoritmo genético especializado na resolução de problemas com variáveis continuas e altamente restritos*. Tese (Doutorado), 2009.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015.

DUFOUR KAREL DRIESEN, L. H. B.; VERBRUGGE, C. Dynamic metrics for java. 2003.

EIGENMANN. *Optimizing Compilers*. 2017. ⟨https://engineering.purdue.edu/~eigenman/ECE663/⟩. Acessado: 2017-11-04.

FEO, J. T. *A comparative study of parallel programming languages: the Salishan problems*. [S.l.]: Elsevier, 2016.

GIDRA, L. et al. A study of the scalability of stop-the-world garbage collectors on multicores. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2013. v. 48, n. 4, p. 229–240.

GITHUB. *The State of the Octoverse 2017*. 2017. ⟨https://octoverse.github.com/⟩. Acessado: 2017-05-10.

GOLDBERG, D. Genetic algoritms in search optimization and machine learning. 1989.

GOLDBERG D., L. B. H. J. Classifer systems and genetic algorithms. 1989.

GOOGLE. *Go license*. 2009. ⟨http://golang.org/LICENSE⟩. Acessado: 2017-04-12.

GOOGLE. *Go 1.5 Release Notes*. 2015. ⟨http://tip.golang.org/doc/go1.5⟩. Acessado: 2017-02-02.

GOOGLE. *The Go Programming Language Specification*. 2016. ⟨https://golang.org/ref/spec⟩. Acessado: 2016-12-04.

GOOGLE. *Go source code*. 2017. ⟨https://github.com/golang/go⟩. Acessado: 2017-01-20.

GRIESEMER, R.; PIKE, R.; THOMPSON, K. *Effective Go The Go Programming Language*. 2017. ⟨https://golang.org/doc/effective⟩. Acessado: 2017-01-13.

HAGEN, W. V. *The definitive guide to GCC*. [S.l.]: Apress, 2011.

HALFORD, M. *Golang genetic algorithm library*. 2017. ⟨https://github.com/MaxHalford/gago⟩. Acessado: 2017-06-10.

HOARE, C. Communicating sequential processes. *Communications of the ACM*, v. 21, p. 666–677, ago. 1978.

HOLLAND, J. Adaptation in natural and artificial systems. 1975.

HUNDT, R. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days*, v. 2011, p. 38, 2011.

JOHNELL, C. Parallel programming in go and scala: A performance comparison. In: . [S.l.: s.n.], 2015.

JONES, A. H. R.; MOSS, E. *The Garbage Collection Handbook*. [S.l.]: CRC Press, 2012.

KULKARNI, S. Improving compiler optimizations using machine learning. 2014.

LEMONGE, A. *Application of Genetic Algorithms in Strucutural Optimization Problems*. Tese (Doutorado), 1999.

LI, F.; TANG, F.; SHEN, Y. Feature mining for machine learning based compilation optimization. In: IEEE. *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*. [S.l.], 2014. p. 207–214.

LI, M. J. G. X.; PADUA, D. Optimizing sorting with genetic algorithms. 2005.

LIN CHI-KUANG CHANG, N.-W. L. S.-C. Automatic selection of gcc optimization options using a gene weighted genetic algorithm. 2008.

LINDEN, R. Algoritmos genéticos. 2012.

MALIK, A. M. Spatial based feature generation for machine learning based optimization compilation. In: IEEE. *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*. [S.l.], 2010. p. 925–930.

NANZ, S. et al. Benchmarking Usability and Performance of Multicore Languages. *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, 2013. ISSN 1949-3770.

OSTERMANN, S. et al. A performance analysis of ec2 cloud computing services for scientific computing. In: SPRINGER. *International Conference on Cloud Computing*. [S.l.], 2009. p. 115–131.

PALLISTER, J.; HOLLIS, S. J.; BENNETT, J. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, Oxford University Press, v. 58, n. 1, p. 95–109, 2013.

PARK, C. et al. Performance comparison of gcc and llvm on the eisc processor. In: IEEE. *Electronics, Information and Communications (ICEIC), 2014 International Conference on*. [S.l.], 2014. p. 1–2.

PETTERSSON, F.; WESTRUP, E. Using the go programming language in practice. *Department of Computer Science, Faculty of Engineering LTH*, 2014.

PIKE, R. The go programming language. *Talk given at Google's Tech Talks*, 2009.

PUMPUTIS, M. *Message Passing for Programming Languages and Operating Systems*. Tese (Doutorado), 2015.

SERFASS, D.; TANG, P. Comparing parallel performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem. *ACM-SE '12 Proceedings of the 50th Annual Southeast Regional Conference*, 2012.

SOUZA, G. Otimização de funções reais multidimensionais utilizando algoritmo genético contínuo. 2014.

SUMMERFIELD, M. *Programming in Go: creating applications for the 21st century*. [S.l.]: Pearson Education, 2012.

TAYLOR, I. L. The go frontend for gcc. 2010.

TAYLOR, I. L. *GCCGo in GCC 4.7.1*. 2012. ⟨https://blog.golang.org/gccgo-in-gcc-471⟩. Acessado: 2017-01-20.

TOGASHI, N.; KLYUEV, V. Concurrency in Go and Java: Performance analysis. *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, 2014.

VARGHESE, S. User-defined types and concurrency. In: *Web Development with Go*. [S.l.]: Springer, 2015. p. 35–58.

VOSE, M. The simple genetic algoritm. 2004.

WILSON, G. V.; BAL, H. E. Using the cowichan problems to assess the usability of orca. *IEEE Parallel & Distributed Technology: Systems & Applications*, IEEE, v. 4, n. 3, p. 36–44, 1996.

WILSON, G. V.; IRVIN, R. B. *Assessing and comparing the usability of parallel programming systems*. [S.l.]: University of Toronto. Computer Systems Research Institute, 1995.

WILSON, G. V.; SCHAEFFER, J.; SZAFRON, D. Enterprise in context: assessing the usability of parallel programming environments. In: IBM PRESS. *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: distributed computing-Volume 2*. [S.l.], 1993. p. 999–1010.

ZDNET. *Programming language of the year 2016*. 2017. ⟨http://www.zdnet.com/article/googles-go-beats-java-c-python-to-programming-language-of-the-year-crown/⟩. Acessado: 2017-05-10.