Tópicos em Paralelização de Compiladores: Um Estudo de Caso no GCC

Giuliano Augusto Faulin Belinassi

Dissertação apresentada ao
Instituto de Matemática e Estatística
da Universidade de São Paulo
para obtenção do título de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Alfredo Goldman

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo
?? de ????? de 2019

Tópicos em Paralelização de Compiladores: Um Estudo de Caso no GCC

Giuliano Augusto Faulin Belinassi

Esta é a versão original da dissertação elaborada pelo candidato Giuliano Augusto Faulin Belinassi, tal como submetida à Comissão Julgadora.

Resumo

Giuliano Augusto Faulin Belinassi. **Tópicos em Paralelização de Compiladores: Um Estudo de Caso no GCC**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

Compiladores são grandes programas destinados a tradução de códigos fonte entre linguagens de programação distintas, e atualmente estão sendo paralelizados para melhor utilizar os recursos dos processadores *manycore* através de técnicas de compilação como o *Link Time Optimization* (LTO). Entretanto tal técnica costuma desacelerar o processo de desenvolvimento incremental de programas de computador, e pode gerar código menos eficiente que o processo clássico de compilação em alguns casos. Esse trabalho apresenta uma visão geral sobre o estado da arte de paralelismo em compiladores, e propõe uma alternativa ao LTO explorando paralelismo interno em um compilador, com *threads*. Para validar os resultados, algumas das técnicas discutidas são implementadas no GCC, e são realizadas análises no tempo total de compilação do projeto GCC e arquivos separados através de técnicas de inferência estatística.

Palavras-chave: Compiladores, Computação Paralela, Analisadores Léxicos, Analisadores Sintáticos, Análise de Fluxo de Dados, GCC, Toolchain.

Abstract

Giuliano Augusto Faulin Belinassi. **Topics in Compiler Parallelization: A Case Study in GCC**. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

Compilers are huge software destinated to translate the source code between distinct programming languages, and currently, they are being parallelized to better use multicore resources using compiling techniques such as Link Time Optimization (LTO). This technique usually slows down the process of interactively software development, and can sometimes generate less efficient code when compared to the classical compilation process. This thesis presents a general vision of the state of art about parallelism in compilers and proposes an alternative to LTO by better exploring parallelism in compiler internals, through threads. To validate the obtained results, some of the techniques discussed here are implemented in GCC, and analyses are performed in the total compilation time of the GCC project and separated files using statistical inference techniques.

Keywords: Compilers, Parallel Computing, Lexical Analysis, Parsers, Dataflow Analysis, GCC, Toolchain.

Lista de Abreviaturas

API	*Application Programming Interface*
AST	Árvore de Sintaxe Abstrata
DP	Programação Dinâmica
GCC	*GNU Compiler Collection*
GNU	*GNU is Not Unix*
IPA	*Inter Process Analysis*
LLVM	*Low Level Virtual Machine*
LGEN	*Local Generation*
LTRANS	*Local Transformations*
LTO	*Link Time Optimization*
OS	Sistema Operacional
RTL	*Register Transfer Language*
SSA	*Static Single Assignment*
WHOPR	*Whole Program Assumptions*
WPA	*Whole Program Analysis*

Lista de Figuras

Sumário

Apêndices

Anexos

Capítulo 1

Introdução

1.1 Motivação

Os avanços nos campos tecnológicos e computacionais levaram uma enorme expansão dos ecossistemas de *software*. Novos programas são criados para suprir as necessidades dos mais diversos domínios, seja através de sistemas web codificados em linguagens de *script*; ou por componentes para sistemas operacionais destinados a embarcados, com a finalidade de controlar algum recurso do *hardware*. Independente da razão por trás do desenvolvimento do *software*, é certo que, em alguma etapa, seu código será transformado em linguagem de máquina por um compilador, mesmo que ele seja executado por um interpretador.

Um compilador nada mais é que um *software* que traduz um código em uma linguagem de programação *A* para outra linguagem *B* (Jeffrey e Ullman, 2007). Compiladores são programas enormes, largamente adotados pela indústria e academia, onde muito esforço e pesquisa foi e ainda é empregado para que eles produzam código eficiente, com destaque para a corretude. Existem projetos enormes destinados a desenvolvê-los e aprimorá-los, como o Gnu Compiler Collection (GCC[1]), capaz de traduzir diversas linguagens como C, C++ e Fortran, para linguagem de máquina. Há também outros projetos menores como o F2C[2], um compilador de Fortran para C, utilizado em ambientes onde não há um compilador Fortran disponível.

Embora seja possível escrever código em linguagem de máquina, o que tornaria um compilador desnecessário, isto é extremamente caro e sujeito a erros, além de ser uma prática extremamente incomum nos projetos contemporâneos a este trabalho (GitHub, 2017) (vide Figura 1.1). Escrever programas em código de máquina tende a ser trabalhoso e financeiramente custoso, com difícil manutenção e quase impossível portabilidade. Um exemplo disso é o *Internal Revenue Service* (IRS) dos Estados Unidos, que ainda mantém máquinas compatíveis com o IBM System/360 devido a existência no sistema de várias linhas de código escritas em linguagem de montagem para essa arquitetura (GAO, 2017). Esse

[1] https://gcc.gnu.org/
[2] https://www.netlib.org/f2c/

problema ganhou visibilidade na imprensa após a falha no *Tax Day* de 2018 (McKenney, 2018).

Compiladores são usados em projetos dos mais variados tamanhos. Grandes projetos podem conter milhões de linhas de código, e até mesmo construir novas linguagens para facilitar o desenvolvimento de novas funcionalidades, e portanto o processo de compilação precisa ser rápido para viabilizar o seu desenvolvimento. Logo, muito estudo é aplicado para desenvolver algoritmos mais eficientes para eles, além de técnicas de computação paralela para aproveitar os recursos dos processadores *multicore*, cada vez mais comuns.

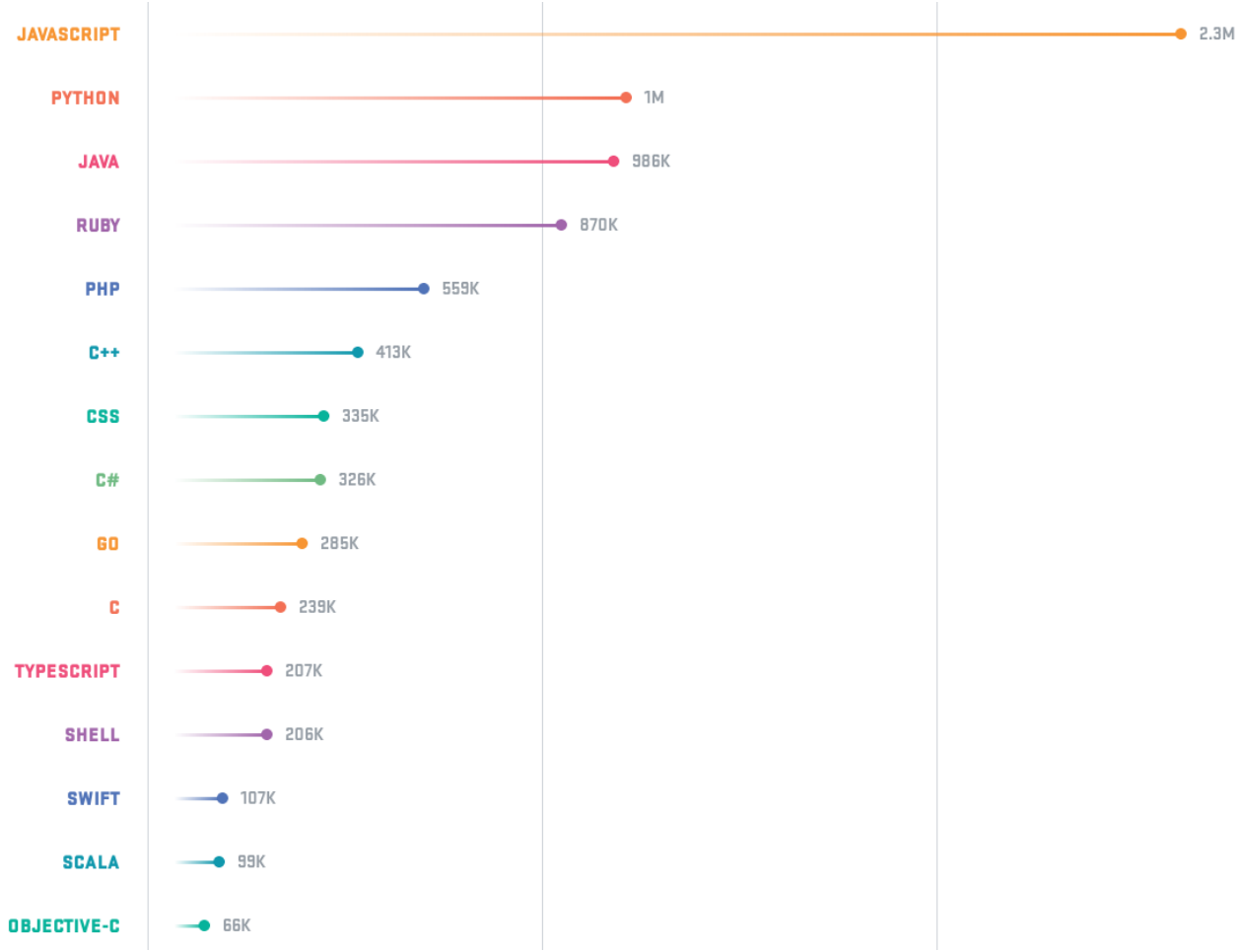


Figura 1.1: *As 15 linguagens mais usadas no GitHub em 2017. Fonte: GitHub, 2017*

1.2 O GCC

O GCC é um compilador de código aberto amplamente usado tanto no meio acadêmico quanto na indústria. Por ser um grande projeto, o GCC contém uma linguagem própria para facilitar a adição de otimizações na geração de código, que em seguida é compilado em um único arquivo C, para (1) aproveitar as otimizações já implementadas no próprio compilador, e (2) evitar reescrever várias funcionalidades já implementadas. Infelizmente, este processo gera um gargalo na compilação do projeto em máquinas *manycore*, pois o GCC não é capaz de compilar um arquivo em paralelo de maneira incremental.

Atualmente, todo o paralelismo na fase de compilação é provido pelo GCC de duas maneiras distintas: ou pelo GNU Make[3], que apenas fornece uma granularidade a nível dos arquivos; ou através da estrutura do LTO, que será abordada na Seção 3. Esse gargalo pode parecer uma característica peculiar do projeto GCC, mas discussões com a comunidade levaram usuários a relatar o mesmo problema em seus projetos. Tal problema pode ser mitigado por um compilador que seja capaz de compilar um único arquivo em paralelo (Cheng, 2018) (atomsymbol, 2019). Outra possível solução seria partir o arquivo em arquivos menores e utilizar o esquema de paralelismo já existente do GNU Make, mas isto implica em uma modificação na estrutura do projeto, o que pode não ser o ideal.

Considerando que atualmente há uma tendência para que os processadores sejam cada vez mais paralelos, compiladores que usufruam deste recurso poderiam reduzir o tempo de compilação de um projeto, ou de uma suíte de testes que necessita recompilação antes de sua execução, economizando recursos no caso da computação ser cobrada por hora. Até então, o paralelismo empregado internamente em compiladores otimizadores foi consequência de uma estrutura criada para permitir otimizações mais agressivas e custosas, como o LTO (Glek e Hubicka, 2010).

1.3 Questões de Pesquisa

Deve ser observado que nem todos os processos de um compilador podem ser paralelizados. Compiladores operam em etapas que podem ser fortemente dependentes do resultado da etapa anterior. Por exemplo, ao compilar uma função em C, pode ser necessário saber se alguma outra foi declarada anteriormente; ou então o analisador sintático pode alterar um estado do analisador léxico para detectar variáveis de um novo tipo. Além disto, compiladores também se apoiam em algoritmos em grafos, cujo paralelismo é um desafio (Lumsdaine *et al.*, 2007).

Sendo assim, esse trabalho concentra-se em duas questões de pesquisas:

QP1 - Em que pontos um compilador pode usufruir de paralelismo?

QP2 - Qual é o ganho de desempenho ao paralelizar internamente um compilador?

essas questões de pesquisa visam propor uma alternativa ao LTO com foco em desenvolvimento incremental, e também cobrindo os casos onde o LTO produz binários menos eficientes através de melhorias no paralelismo do processo clássico de compilação. Para validar os resultados, este trabalho inclui a implementação no GCC de algumas das técnicas discutidas, além da utilização de técnicas de inferência estatística para análises no tempo total de compilação no projeto GCC e arquivos separados.

Este trabalho está dividido nos seguintes capítulos: no Capítulo 2, são apresentados conceitos teóricos como parte da Teoria de Compiladores e Computação Paralela para que seja possível compreender as questões acerca do trabalho. Em sequência, o Capítulo 3 contém uma apresentação dos trabalhos relacionados a paralelismo de compiladores até o estado da arte nesse tópico, além de uma breve discussão sobre seus pontos positivos e negativos. Por fim, o Capítulo 4 apresenta uma visão geral do trabalho de fato, com

[3] https://www.gnu.org/software/make/

uma análise de tempo consumido que permitiu identificar quais pontos necessitam de melhorias, além da proposta do trabalho, que contempla a arquitetura de paralelismo e o planejamento de sua implementação.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta uma breve introdução sobre os conceitos utilizados neste trabalho. Primeiro, são abordadas as partes do *pipeline* de um compilador de maneira a entender os passos de compilação. Em seguida, discute-se como o compilador GCC apresenta esses tópicos implementados, além do seu uso no processo de compilação de um programa. Por fim, são apresentados alguns conceitos e algoritmos básicos sobre Computação Paralela.

2.1 Compiladores

Para evitar confusões a respeito das diferentes linguagens que um compilador trabalha em seus muitos estágios, é adotado a seguinte nomenclatura neste trabalho: Denote *Linguagem Fonte* como sendo a linguagem na qual o código fonte do programa a ser compilado foi originalmente escrito; *Linguagem Intermediária* uma das linguagens internas do compilador; e *Linguagem Alvo* a linguagem na qual o compilador deverá gerar código.

Compiladores são grandes projetos destinados a tradução de códigos fonte entre distintas linguagens de programação. Normalmente, compiladores traduzem linguagens de alto nível, como C, para linguagens mais próximas da máquina, como a linguagem de montagem - embora isso não seja uma regra pois existem compiladores cuja única tarefa é realizar uma tradução entre duas linguagens de alto nível.

Compiladores também costumam otimizar o código que passam por eles, aplicando diversas heurísticas de maneira a acelerar o código a ser produzido, substituindo trechos de código por outros mais eficientes, reordenando as instruções do programa, etc.

Por serem programas extremamente grandes, existe um grande interesse em evitar reescrever um novo compilador sempre que uma nova linguagem é proposta, ou um novo *hardware* é criado. Para isso, a seguinte modularização é largamente adotada por projetistas de compiladores (Novillo, 2016) (LLVM, 2019a), conforme ilustrado na Figura 2.1:

- Um *Front End* para cada Linguagem Fonte. Neste módulo são executadas as análises léxica e sintática, com a finalidade de construir uma Árvore de Sintaxe Abstrata

(AST), para que em seguida, ela seja convertida para uma Linguagem Intermediária do compilador.

- Um *Middle End*, responsável por aplicar diversas otimizações independentes de arquitetura no código já convertido para a Linguagem Intermediária.
- Um *Back End* para cada linguagem alvo, responsável por converter a linguagem intermediária na linguagem alvo. Aqui outras otimizações específicas da linguagem alvo são realizadas, como alocação de registradores e seleção de instruções.

Porém, essa modularização é opcional: compiladores mais antigos destinados a fazer uma conversão direta entre duas linguagens, como o UNIX C Compiler, não costumam usar uma linguagem intermediária (Ritchie, 1979), além de diversos livros-texto não mencionarem tal módulo, fundindo suas funcionalidades com o *Back End* (Jeffrey e Ullman, 2007). Não adotar tal metodologia simplifica o projeto, mas dificulta a inserção de suporte a novas linguagens de programação.

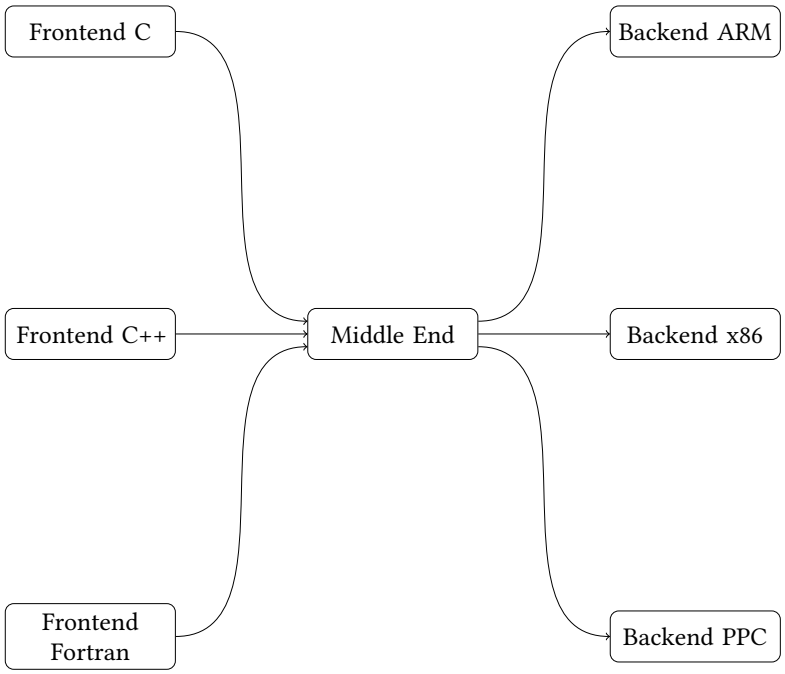


Figura 2.1: *Arquitetura de um compilador.*

2.1.1 *Front End*

Um *Front End* é um módulo responsável por interagir diretamente com o código escrito na Linguagem Fonte a ser compilado. Ele costuma ser único para cada Linguagem Fonte, ou seja, para inserir suporte a uma nova Linguagem Fonte, basta implementar um novo *Front End*, mas é possível que algumas classes sejam compartilhadas, como no caso de um compilador que dê tanto para C quanto para e C++.

Como primeiro passo, *Front End* realiza uma análise léxica e sintática, com a finalidade de gerar uma AST e verificar se o texto dado como entrada realmente pertence à linguagem. Estes dois analisadores se comunicam constantemente, conforme ilustrado na Figura 2.2.

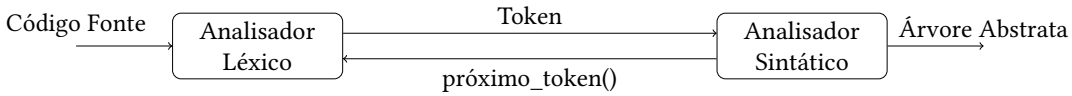


Figura 2.2: *Interações entre um analisador léxico e sintático.*

Primeiro, a análise léxica é responsável por ler o arquivo de texto contendo o código fonte, e quebrar as palavras em *tokens*, que serão alimentados para o analisador sintático. Por consequência, ele é capaz de realizar alguns filtros, *e.g.* ignorar os comentários, identificar constantes numéricas, eliminar espaços desnecessários, e substituir um *token* por outro. Analisadores léxicos também podem ser usados para substituição de macros, como implementado pelo preprocessador do C (CPP)[1].

Geralmente, os Analisadores Léxicos são implementados utilizando autômatos por diversos motivos, entre eles sua atrativa complexidade computacional $O(n)$, onde n é o tamanho da entrada, pela existência de algoritmos para converter expressões regulares em autômatos (Thompson, 1968), e pela existência de algoritmos para minimização de estados de um autômato (Hopcroft, 1971). Isto possibilita que geradores de analisadores léxicos como o `Flex`[2] sejam bastante eficientes.

Em conjunto com o Analisador Léxico, o analisador sintático é responsável por gerar a Árvore de Sintaxe Abstrata (AST), inspecionando a sequência de *tokens* fornecida pelo Analisador Léxico. Neste processo, ele certifica-se de que o código fonte respeita a gramática da linguagem, apontando erros caso contrário.

Analisadores Sintáticos se apoiam nas gramáticas não ambíguas que geram uma linguagem livre de contexto determinística, isto porque gramáticas mais poderosas requerem algoritmos computacionalmente mais custosos (Sipser, 2012). Existem vários algoritmos para realizar a análise sintática, mas destacam-se:

- Recursivo Descendente: Um algoritmo de caráter preditivo, que tenta construir a AST da raiz para as folhas. Uma característica desse algoritmo é a sua facilidade de codificação, que utiliza chamadas de funções para armazenar o contexto atual e progredir na análise conforme as regras da gramática. Seu uso é devido a facilidade de codificação e mensagens de erro mais informativas para o usuário.
- LL(k): Outro algoritmo de caráter preditivo, que tenta construir a AST da raiz para as folhas. Aqui o algoritmo usa uma máquina de estados em conjunto com uma pilha para armazenar o contexto atual. Sua codificação é mais complexa que o anterior, e as mensagens de erro costumam ser mais precárias.
- LR(k): Um algoritmo que tenta construir a AST das folhas para a raiz. Assim como o LL(k), este algoritmo também utiliza uma máquina de estados em conjunto com uma pilha. Apesar de sua codificação ser mais complicada, esse algoritmo é capaz de processar gramáticas mais complexas que o LL(k), conforme proposto por Knuth, 1965. Outra vantagem interessante é a existência de *softwares* como o `Bison`[3], capaz

[1] https://gcc.gnu.org/onlinedocs/cppinternals/Lexer.html
[2] https://www.gnu.org/software/flex/
[3] https://www.gnu.org/software/bison/

de gerar um analisador LR(1) a partir da especificação da gramática.

A complexidade dos analisadores LL(k) e LR(k) é $O(n)$, onde n é o tamanho da entrada. Já o Recursivo Descendente a complexidade pode variar dependendo da implementação. Por fim, uma descrição mais detalhada do funcionamento destes algoritmos pode ser encontrada em (Appel, 2004).

Uma vez gerada a AST, é possível fazer verificações extras de semântica relacionada a linguagem de programação. Em seguida, essa AST normalmente é traduzida para uma Linguagem Intermediária, onde serão feitas otimizações independentes da linguagem alvo no código. Por fim, o controle é passado para o *Middle End* do compilador.

2.1.2 *Middle End*

O *Middle End* é responsável por trabalhar em uma ou mais Linguagens Intermediárias do compilador, com a finalidade de efetuar otimizações no código e possíveis verificações de erro que possam ser postergadas até essa fase. Essas linguagens devem ser projetadas de maneira a capturar a semântica da Linguagem Fonte, na qual o programa original foi escrito. Existem várias representações convenientes para a Linguagem Intermediária, mas destaca-se para fins de otimização a *Three-Address Code*, que normalmente é usada em conjunto com a *Static Single Assignment* (SSA).

Three-Address Code

Nesta representação, expressões como $a * x + b$ são representadas como:

$$t_1 = a * x$$
$$t_2 = t_1 + b$$

ou seja, todas as expressões são transformadas em uma sequência de expressões contendo apenas dois operandos e uma operação de atribuição. Para isso, novas variáveis são declaradas para conter os valores intermediários possibilitando também a remoção de sub-expressões em comum. Também nesta linguagem, fluxo de código e laços são transformados em uma sequência de `ifs` e `gotos` para facilitar a construção de um grafo de controle de fluxo, possibilitando análises nessa estrutura.

Esta representação é demasiada útil por facilitar o processo de alocação de registradores, uma vez que nela, a maneira de como os processadores operam instruções aritméticas é representada com fidelidade(Lattner e Adve, 2002).

Static Single Assignment

O *Static Single Assignment* (SSA) é uma linguagem intermediária onde uma variável é atribuída uma única vez. Sendo assim, toda vez que uma variável no código original é modificada, é necessário crar na representação SSA uma nova variável para registrar essa atribuição. Isto facilita diversas otimizações no controle de fluxo do programa, assim como possibilita a remoção de variáveis não utilizadas ou que não serão mais utilizadas em um trecho de código, processo conhecido como *Liveness Analysis*.

Entretanto, quando essa representação é usada em conjunto com um fluxo de execução, como `if-else`, isso pode gerar um problema. Considere o código na Figura 2.3a. Neste código há dois valores possíveis para a quando a expressão $u = a * v$ for executada, sendo necessário anotar qual versão de a utilizar. A solução é introduzir uma função ϕ, que seleciona a variável corretamente utilizando a informação do arco usado para se chegar no bloco contendo a sentença com a função ϕ, conforme ilustrado na Figura 2.3b.

```
1  if (condition) then
2      a = −1;
3  else
4      a = 1;
5  end
6  u = a∗v;
```

(a)

```
1  if (condition) then
2      a1 = −1;
3  else
4      a2 = 1;
5  end
6  a3 = φ(a1, a2)
7  u = a3∗v;
```

(b)

Figura 2.3: *Um programa (a) e sua representação em SSA em (b)*

Quanto a implementação de SSA em compiladores, Cytron *et al.*, 1991 discute com detalhes os algoritmos para construir tal representação. Já Appel, 2004 discute tal representação em alto nível, evidenciando possíveis otimizações nesta representação.

Otimizações Independente de Arquitetura

Uma das funcionalidades mais atrativas de um compilador é sua capacidade de fazer otimizações no código enquanto mantém a corretude do mesmo, principalmente quando o alvo é uma linguagem de montagem. Um compilador com essa funcionalidade é capaz de gerar código que economiza energia gasta em processamento, poupa esforços de otimização, desenvolvimento, entre outros.

Considerando o contexto de um compilador capaz de compilar diversas linguagens para os mais variados alvos, o *Middle End* é exatamente onde maior parte do tempo gasto na implementação de otimizações é alocado, pois isto implica gerar um código mais eficiente para todas as Linguagens Alvo cujo o compilador dá suporte.

Como o *Front End* já traduziu todo o código para a linguagem intermediária, então é perfeitamente possível marcar o início de todas as funções do código. Com isto, é possível particionar o conjunto das possíveis otimizações em dois conjuntos disjuntos:

- Otimizações *Intra-Procedurais*: Otimizam cada função sem interagir com as demais funções do programa.
- Otimizações *Inter-Procedurais*: Procuram observar o programa como um todo, observando as interações de cada função com as demais funções do programa. Um exemplo clássico é remover funções não utilizadas, e decidir se uma determinada função será compilada com o atributo `inline`.

Essa distinção é útil pois não há dependência entre as funções ao aplicar uma otimização *Intra-Procedural*, e portanto as funções podem ser processadas em paralelo. Diversas

otimizações clássicas como Invariante de Laços, Propagação de Constantes, Eliminação de Redundância, Eliminação de Subexpressão Comum, Propagação de Constantes e Eliminação de Código Morto são classificadas como tal, reforçando o argumento de paralelização.

Entretanto, esse argumento não é válido para as otimizações *Inter-Procedurais*, pela própria definição. Estas otimizações são normalmente aplicadas utilizando algoritmos em grafos: Cada rotina é representado como um vértice, e existe um arco de f à g se f chama g em seu corpo, conforme ilustrado na Figura 2.4. Construir tal grafo pode ser um desafio dependendo da linguagem de programação, principalmente quando se utiliza orientação a objetos devido a possibilidade de sobrescrita de métodos. Por fim, mais detalhes a respeito dos algoritmos referentes a essas otimizações são discutidos por Khedker *et al.*, 2009.

```
1   extern function g,h
2
3   function f()
4       call g
5       call h
6   end
7
8   function main()
9       call f
10      call h
11  end
```

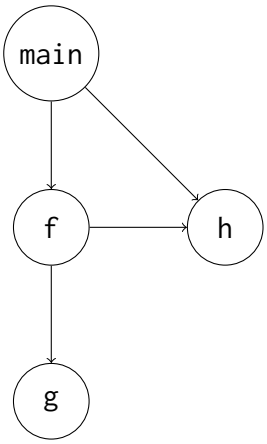


Figura 2.4: *Um programa e seu respectivo grafo de chamada de funções*

Após todo esse processo de otimização, o código nesta linguagem ainda pode sofrer transformações para outras Linguagens Intermediárias para algo que seja mais próximo do *hardware*-alvo (nos casos em que a compilação tem como alvo código de máquina). Finalmente, após essa sequência de transformações, o controle é passado ao *Back End* do compilador, onde ele será transformado na Linguagem Alvo.

2.1.3 *Back End*

O *Back End* é responsável pela geração do código final na Linguagem Alvo, além otimizações dependentes da arquitetura. Sendo assim, ele também deve efetuar otimizações referentes a seleção de instruções mais eficazes e alocação registradores da melhor maneira possível para as variáveis do programa, caso a linguagem alvo seja a Linguagem de Montagem.

Existem diversas técnicas para selecionar instruções. Uma estratégia possível é substituição de macros, onde as instruções são substituídas observando localmente cada linha de código da Linguagem Intermediária. Esta estratégia tem a vantagem de realizar uma tradução rápida e de fácil implementação, mas que gera código ineficiente, conforme ilustrado na Figura 2.5. Existem otimizações que podem ser aplicadas para melhorar o resultado das instruções geradas, sendo a mais famosa delas a *Peephole Optimization*, que tenta remover leituras e escritas desnecessárias a memória buscando instruções que se anulam.

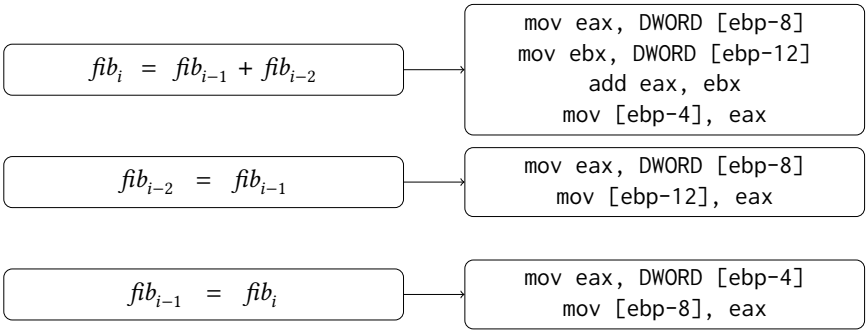


Figura 2.5: *Seleção de instruções usando expansão de macros para i386. Sintaxe da Intel.*

Uma outra maneira é realizar a seleção de instruções através de uma representação em árvore da Linguagem Intermediária. Inicialmente desenvolvida como um aperfeiçoamento da técnica de substituição de macros, essa técnica busca por padrões na árvore e as substitui pelas instruções de acordo com uma regra especificada. As primeiras implementações deste método eram heurísticas simples, substituindo os padrões conforme eles eram encontrados, sem se preocupar em escolher o melhor conjunto de padrões.

Glanville e Graham, 1978 evoluiu tal método utilizando as mesmas técnicas de análise léxica e sintática normalmente empregadas no *Front End*. A partir do algoritmo LR(1), Graham conseguiu postergar a emissão de código utilizando a máquina de estados desse algoritmo para armazenar um contexto limitado das instruções já vistas. Quando o algoritmo decide emitir código, é selecionado o conjunto de instruções que minimiza o custo total do contexto atual. Isso permite selecionar instruções em tempo linearmente proporcional ao tamanho da expressão e considerar o custo das instruções no momento de emitir o código.

Embora a técnica de Graham-Glanville tenha uma complexidade computacional bem atraente, esse método se mostrou difícil de implementar na prática devido a quantidade de instruções disponíveis nos processadores, resultando em uma máquina de estados com milhares de estados, e vários conflitos de ambiguidade. Também é importante notar que esse método não é capaz de gerar código com custo ótimo no caso geral, pois uma vez que o algoritmo decide emitir as instruções, elas não serão mais modificadas independentemente das instruções lidas posteriormente (Blindell, 2016).

Ainda considerando a seleção de instruções em uma representação em árvore, é possível selecionar as instruções de maneira ótima através de Programação Dinâmica, conforme proposto por Ripken, 1977. A ideia consiste em observar que para minimizar o custo de selecionar os padrões na árvore inteira, basta considerar o melhor custo dos filhos e tentar selecionar um padrão que minimiza o custo da raiz. Matematicamente, seja P_n o conjunto de padrões que são possíveis de aplicar em n, o nó atual da árvore. Então o custo mínimo $DP[n]$ é:

$$DP[n] = \begin{cases} \min_{p \in P_n} CUSTO[p] & \text{se } n \text{ é nó folha.} \\ \min_{p \in P_n} CUSTO[p] + \sum_{f \in FILHO[p]} DP[f] & \text{caso contrário.} \end{cases}$$

Outra maneira de selecionar instruções é através de uma representação em Grafos Ací-

clicos Direcionados (DAGs). Sua vantagem é a possibilidade de representar subexpressões em comum em uma expressão, mas infelizmente selecionar instruções de maneira ótima em um DAG é NP-completo (Koes e Goldstein, 2008). A maioria das técnicas empregadas em seleção de instruções utilizando DAGs são gulosas, como o caso do LLVM, 2019b.

Por fim, uma discussão mais detalhada sobre as técnicas de geração de código de máquina é discutida por Blindell, 2016. Além disso, para adicionar suporte a uma nova arquitetura ou linguagem alvo, basta implementar um novo *Back End*, aproveitando todo o código já implementado nas etapas anteriores.

2.2 O Compilador GCC

A coleção de compiladores da GNU, também conhecido como GCC, é um projeto iniciado por Richard Stallman com a primeira versão disponibilizada ao público em Março de 1987. Tal versão apenas fornecia suporte a linguagem C, mas já permitia gerar código para diversas arquiteturas (GNU, 2019a). Hoje, o GCC fornece suporte a diversas Linguagens Fonte como C, C++, Fortran, Go, Ada, e várias arquiteturas, como i386, amd64, riscv, arm, ppc, pdp-11. O GCC também é um compilador otimizador, ou seja, ele é capaz de alterar o código fornecido pelo usuário com a finalidade de gerar código mais eficiente, sem alterar a semântica do programa. Isso é possível por conta das várias Linguagens Intermediárias implementadas no GCC: GENERIC, GIMPLE, e RTL. O compilador executa vários passos de otimização em cada uma dessas linguagens intermediárias.

2.2.1 GENERIC

O GENERIC é uma linguagem intermediária que possibilita representar quaisquer funções do programa através de árvores. Tal linguagem é um superconjunto das funcionalidades e atributos que cada Linguagem Fonte contém, por exemplo, atributos como `volatile` de C e C++ contém uma representação nessa linguagem. Outras funcionalidades são: chamadas de função, declaração de variáveis, classes, funções, expressões, entre outros (GNU, 2019c).

Essa linguagem contém uma representação mais próxima de como o programa foi escrito pelo programador, ou seja, não são feitas modificações nas expressões. Sendo assim, essa linguagem intermediária não se enquadra nas categorias de *Three Address Code*, ou *Static Single Assignment* pois isso demanda modificações no código. O objetivo principal dessa linguagem é facilitar a tradução para a próxima linguagem intermediária, a GIMPLE, e assim facilitar o desenvolvimento dos *Front End* do compilador, mas isso não significa que otimizações não possam ser aplicadas aqui.

O GCC possui um mecanismo de substituição de nós em sua representação interna, dando suporte a árvore construída em GENERIC. Esse mecanismo é simples: o compilador procura por uma cadeia de nós na árvore, buscando padrões, como especificado no arquivo `gcc/match.pd`, e os substitui pelo o que foi instruído na regra de substituição. Esse mecanismo, embora simples, permite com que diversas simplificações matemáticas sejam implementadas no código, o que pode significar uma melhora de até 10× em casos particulares (Belinassi, 2018).

Por fim, cada *Front End* do GCC é responsável por traduzir o código fonte para esta linguagem intermediária. Para visualizar um programa nessa representação, basta compilá-lo utilizando o atributo `-fdump-tree-original`.

2.2.2 GIMPLE

O GIMPLE é outra linguagem intermediária do GCC, basicamente consistindo na conversão de GENERIC para *Three Address Code*, havendo ainda um passo posterior adicionando uma representação SSA a esta linguagem. Assim como o GENERIC, a GIMPLE também é geral o suficiente para representar os atributos de todas as Linguagens Fonte de cada *Front End*. Essa linguagem intermediaria é baseada na linguagem SIMPLE, do compilador McCAT (GNU, 2019d).

Por ser uma linguagem do tipo *Three Address Code*, é aqui onde grande parte das otimizações livre de arquitetura são aplicadas, e portanto o GIMPLE fornece uma *Application Programming Interface* (API) detalhada para efetuar alterações no código, incluindo a representação em forma SSA para análise de fluxo que os passos de otimização deverão utilizar.

Por fim, o compilador pode ser instruído a escrever o código em GIMPLE através da opção `-fdump-tree-gimple`, ou então usando a opção `-fdump-tree-ssa` para escrever a representação SSA em disco.

2.2.3 *Register Transfer Language*

Por fim, a última linguagem intermediária do compilador GCC é a *Register Transfer Language* (RTL). A sintaxe dessa linguagem tem inspiração no Lisp, onde as expressões são representadas por uma lista de comandos (GNU, 2019b).

A RTL é uma linguagem que visa ser extremamente próxima da linguagem de montagem, mas ainda sendo genérica o suficiente a ponto de ser compatível com todas as Linguagens Alvo do GCC. Para isso, ela representa uma máquina com infinitos registradores, que serão alocados em registradores reais no passo correspondente. Caso a arquitetura alvo não tenha o número necessário de registradores, as variáveis deverão ser guardadas em memória. Por exemplo, no caso do i386, elas deverão ser salvas na pilha. Esse processo é conhecido como *Register Spilling*.

Essa é a última Linguagem Indermediária antes da geração do código final. O *Back end* transformará o código nessa linguagem para a Linguagem Alvo substituindo uma sequência de instruções em RTL por uma sequência de instruções na Linguagem Alvo. Essa escolha segue um sistema de custos no qual o compilador procura escolher uma sequência de instruções que minimiza a soma do custo total da tradução.

Na arquitetura i386, as regras de substituição estão implementadas em `gcc/config/i386/i386.md`, com o sistema de custos definido na `struct processor_costs`, no arquivo `gcc/config/i386/i386.h`. Cada processador da arquitetura i386 deve definir o custo de cada instrução, como definido em `gcc/config/i386/x86-tune-costs.h`.

2.2.4 Passos de Otimização

O GCC possui vários passos de otimização que são aplicados em cada linguagem intermediária. Esses passos são divididos em três etapas:

1. **IPA** - Após a tradução do programa para GENERIC, o compilador constrói um grafo de chamadas de função. Para cada chamada de g a partir de f, é inserido um arco de f à g. Com isso o compilador pode decidir eliminar funções que não são utilizadas no programa simplesmente descartando a função do código, ou então decidir que uma função precisa ser convertida para `inline`, como programado em `pass_ipa_inline`.

2. **GIMPLE** - Após a etapa IPA, são executados os passos de otimização do GIMPLE. Tais passos são executados em cada função de maneira independente, ou seja, são otimizações Intra Procedurais. Existem diversas otimizações que são aplicadas nesse passo. Por exemplo, aqui é aplicado a `pass_vectorize`, que tenta vetorizar um laço, a `pass_parallelize_loops`, que tenta fazer paralelização automática de laços, a `pass_sincos`, que tenta aplicar otimizações algébricas em funções trigonométricas, entre outras.

3. **RTL** - Após a etapa GIMPLE e o código RTL ter sido gerado, são aplicadas otimizações no código RTL. Essas otimizações também são de caráter Intra Procedural, e portanto podem ser aplicadas de maneira independente em cada função.

Cada otimização está implementada em algum arquivo específico do GCC, agrupada por similaridade. Por exemplo, o passo de vetorização está implementado em `gcc/tree-ssa-loop.c`. Cada otimização também deve ser implementada herdando e implementando as funções de uma das seguintes classes: `gimple_opt_pass`, `ipa_opt_pass_d`, `rtl_opt_pass`, `simple_ipa_opt_pass`, como ilustrado na Figura 2.6. Por fim, a ordem que as otimizações são feitas estão especificadas no arquivo `gcc/passes.def`.

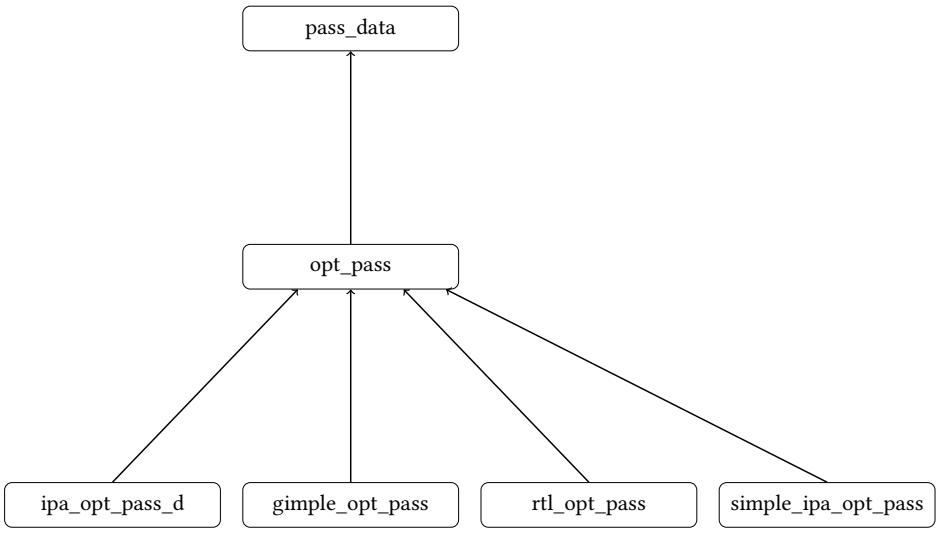


Figura 2.6: *Hierarquia dos passos de otimização do GCC.*

2.2.5 GNU Toolchain e o Processo Clássico de Compilação

Um *toolchain* é um conjunto de ferramentas que são ligadas em cascata para o desenvolvimento de *software*. Normalmente o *toolchain* consiste em um compilador, um montador e um linkeditor, mas também pode conter mais ferramentas, como um *debugger*.

A GNU providencia um conjunto de ferramentas para desenvolvimento de software conhecido como GNU *toolchain*. São parte desse toolchain:

- *Binutils*, um conjunto de ferramentas contendo o linkeditor LD, o montador AS, e outras ferramentas[4].
- O driver `gcc`, responsável por chamar os compiladores `cc1`, `cc1plus`, `f951` para C, C++ e Fortran, respectivamente, e também responsável por interagir com as demais ferramentas do *Toolchain*.
- A biblioteca `glibc`.
- O *debugger* GDB.

As ferramentas acima interagem umas com as outras da seguinte forma: primeiro, o código na Linguagem Fonte é compilado usando o GCC para a linguagem de máquina, em seguida, o código é montado utilizando o AS (que constrói o arquivo objeto) e por fim o linkeditor LD coleta todos os arquivos objetos gerados, construindo o binário especificado. A Figura 2.7 retrata esse processo.

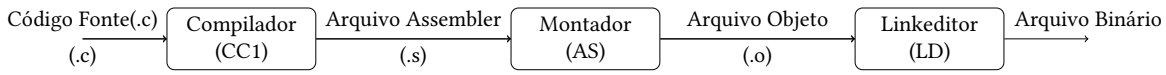


Figura 2.7: *Etapas de compilação.*

Esse processo é tradicionalmente usado para compilar *softwares* complexos, em conjunto com uma ferramenta que controla a geração de arquivos objetos e executáveis como o GNU Make, como ilustrado na Figura 2.8. Aqui cada arquivo é compilado individualmente, passando por todos os processos da compilação (tradução para a linguagem intermediária, otimização, tradução para a Linguagem Alvo, e encapsulamento no arquivo objeto), para que em seguida eles sejam ligados por um linkeditor como o LD. Através do Make, onde é possível explicitar as dependências de um arquivo, a compilação pode ser efeutada em paralelo entre arquivos que não dependem uns dos outros.

Neste processo, não é possível aplicar otimizações observando o programa como um todo, pois o compilador não tem informações sobre as funções que estão em outros arquivos. Sendo assim, todas as otimizações são aplicadas localmente usando apenas o contexto do arquivo.

[4] https://www.gnu.org/software/binutils/

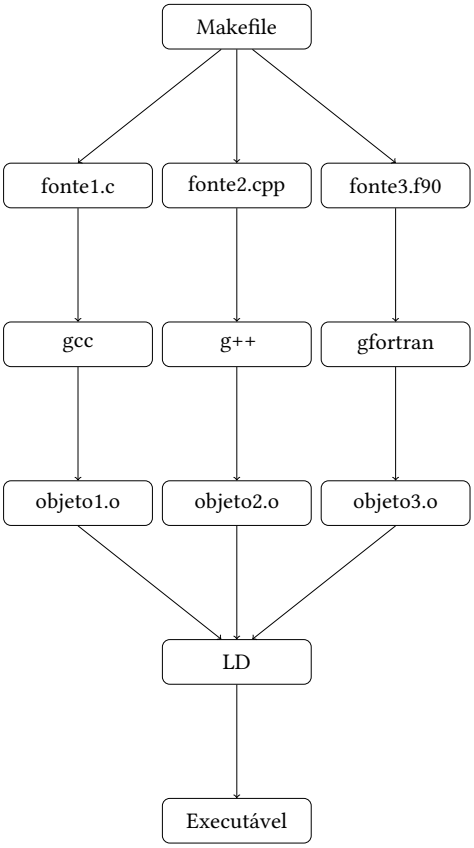


Figura 2.8: *Processo clássico de compilação de um programa.*

2.3 Computação Paralela

A Computação Paralela é o ato de coordenar vários fluxos de execução operando simultaneamente em um ou mais computadores para atingir um objetivo em comum. Ela se tornou uma tendência na computação contemporânea por conta da facilidade de aquisição e custo de produção reduzido dos processadores *multicore* e *manycore*, em relação a um processador de igual desempenho com apenas um núcleo.

Para ilustrar a evolução dos processadores *multicore*, Rupp, 2018 publicou um gráfico mostrando o número de transistores, núcleos e desempenho de cada núcleo desde 1970, apresentado na Figura 2.9. Neste gráfico é possível notar que a Lei de Moore ainda se aplica, mas a desempenho sequencial dos processadores está em desaceleração, enquanto o número de núcleos cresce exponencialmente desde 2008. Isto justifica o uso de computação paralela sempre que possível para que seja utilizado o máximo dos recursos computacionais disponíveis.

Nessa seção são discutidas alguns conceitos básicos de computação paralela e alguns algoritmos úteis na paralelização de programas.

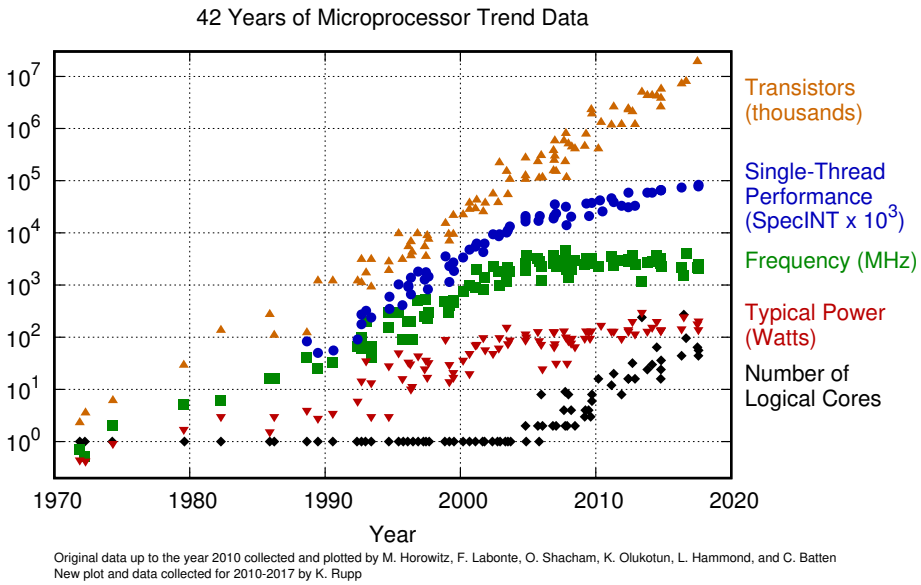


Figura 2.9: *42 anos de evolução dos processadores. Fonte: Rupp, 2018*

2.3.1 *Speedup*

Uma maneira de medir o ganho de tempo em uma implementação paralela de um algoritmo é através no conceito de *speedup*. Sejam T_1 o tempo do algoritmo sequencial e T_n o tempo desse algoritmo paralelizado com n núcleos de processamento. Então o *speedup* S_n é dado por:

$$S_n = \frac{T_1}{T_n}$$

2.3.2 A Taxonomia de Flynn

Mesmo com a evolução dos processadores *multicore* e *manycore*, estas não são as únicas arquiteturas disponíveis para realizar computação paralela. Michael J. Flynn (Pacheco, 2011) definiu uma taxonomia das arquiteturas, mas vale destaque para duas delas:

1. **SIMD** - *Single Instruction, Multiple Data*. Isso se refere a processadores vetoriais que permitem que uma operação seja executada simultâneamente em todos os elementos de um vetor simultâneamente. Alguns exemplos são as instruções SSE da Intel, e também as *Graphics Processing Units* (GPU), embora essa última não seja considerada uma arquitetura puramente SIMD.

2. **MIMD** - *Multiple Instruction, Multiple Data*. Isso se refere aos sistemas *multicore* independentes, capazes de executar tarefas de maneira assíncrona. Aqui estão localizados ambas as arquiteturas de memória compartilhada e memória distribuída.

2.3.3 Programação Paralela em Memória Compartilhada

Uma maneira de permitir Computação Paralela é através de uma memória compartilhada. Aqui vários processadores ou núcleos são ligados através de um barramento a uma memória compartilhada entre todos os processadores, que permite leitura e escrita

de dados. Para evitar problemas de concorrência, são necessários mecanismos de travas e sincronização.

Aqui, muitos dos recursos para sincronização são fornecidos pelo Sistema Operacional (OS), e portanto é necessário explicar um pouco das abstrações que ele fornece.

Processos e *Threads*

Antigamente, os processadores eram capazes de executar apenas um programa por vez. Com isto em mente, os desenvolvedores de Sistemas Operacionais criaram uma série de abstrações para que fosse possível dar a ilusão ao usuário que vários programas estavam executando ao mesmo tempo. Embora isso não seja mais um fato, já que hoje os processadores são capazes de executar vários fluxos de execução simultaneamente, essa é a principal motivação do conceito de Processos e *Threads*.

Um *Processo* é um modelo de abstração do Sistema Operacional que dá a um programa a ilusão de que ele tem o monopólio do processador (Love, 2005). Cada processo tem a sua região de memória privada por padrão, e para que processos possam compartilhar memória, isso deve ser explicitado no programa. Um exemplo onde isso pode ser feito é através das chamadas `shmget` do extinto `System V`, mas cujo o Linux ainda oferece suporte (*shmget Man Page* 2019).

Cada processo contém pelo menos um fluxo de execução, e cada fluxo de execução é chamado de *thread*. Cada *thread* dentro de um processo compartilha memória com as demais *threads* do processo, facilitando assim o desenvolvimento de mecanismos de comunicação através de uma memória compartilhada.

Mecanismos de Sincronização

Seja através de auxílio do *hardware*, ou com algoritmos puramente implementados em *software*, o Sistema Operacional costuma fornecer mecanismos de sincronização para coordenar as várias *threads* de um processo, ou até mesmo vários processos.

Um dos mecanismos mais famosos são os *Semáforos* (Dijkstra, 1965). Um semáforo nada mais é que um contador que assume valores não negativos com duas operações atômicas: `P` para decrementá-lo, e `V` para incrementá-lo. Quando o semáforo atinge o valor `0`, a *thread* que em sequência tentar decrementá-lo será bloqueada. A sua execução apenas será retomada quando o semáforo for novamente incrementado por outra *thread* (*sem_overview Man Page* 2019).

Semáforos também podem ser utilizados para resolver o problema da seção crítica, que é uma seção de código que apenas uma *thread* deve poder executar por vez, embora seja mais comum a utilização de *Mutexes*.

Mutexes nada mais são que uma estrutura que garante a exclusão mútua de uma região, conforme ilustrado na Figura 2.10. Nessa figura, três *threads* são iniciadas executando a função f, mas apenas uma pode executar a seção crítica. Quando o *mutex* `lock` é travado, todas as outras *threads* que tentarem acessar a seção crítica serão bloqueadas pelo Sistema Operacional, garantindo assim que apenas uma *thread* esteja executando a seção crítica.

As *threads* bloqueadas somente voltarão a executar quando o *mutex* for desbloqueado pela *thread* que o travou.

```
1   extern function mutex_lock, mutex_unlock;
2   mutex_t lock;
3
4   async function f()
5       mutex_lock(lock);
6     ▹ Região crítica
7       mutex_unlock(lock);
8   end
9
10  function main()
11      for i=1, 3 do
12          call f;
13      done
14  end
```

Figura 2.10: *Ilustração de uso de um mutex,*

Outro mecanismo de sincronização muito útil são as Barreiras. Uma barreira é um ponto no programa onde, assim que a *thread* o executar, ela será bloqueada até que todas as *threads* cheguem ao mesmo ponto, como ilustrado na Figura 2.11. Isso é bastante útil para evitar que uma *thread* avance na execução do código enquanto ela deveria estar aguardando o resultado da computação de outras *threads*. As barrerias também costumam receber um parâmetro de inicialização indicando quantas *threads* precisam atingir a barreira para que todas as *threads* que a atingiu sejam desbloqueadas.

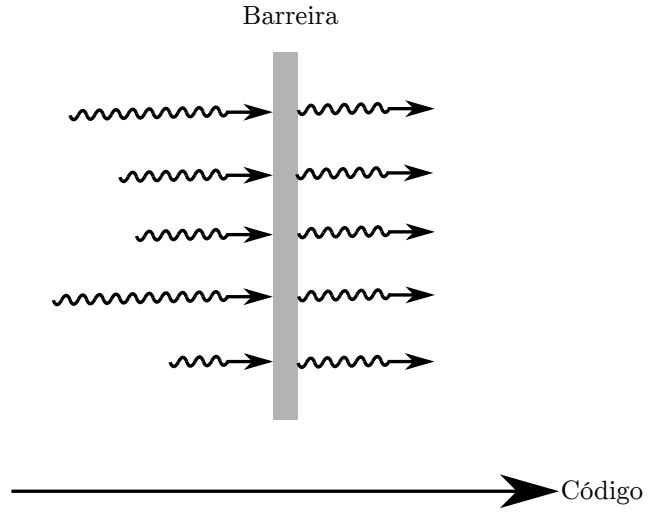


Figura 2.11: *Ilustração do funcionamento de uma barreira.*

Por fim, exceto os semáforos, todos esses mecanismos mencionados acima são implementados na biblioteca `Pthread`. Já os semáforos estão disponíveis no Linux através da interface `semaphore.h`.

Capítulo 3

Trabalhos Relacionados

Boa parte das pesquisas dedicada ao paralelismo em compiladores é destinada ao desenvolvimento de compiladores que paralelizam o código fonte fornecido automaticamente. Há basicamente duas áreas de estudo relacionadas a paralelização de compiladores, são elas: análise de texto em paralelo e análise de fluxo de dados em paralelo. Trabalhos dessas áreas também são apresentados em ordem cronológica nessa seção.

Dos algoritmos apresentados na Seção 2, o algoritmo Recursivo Descendente é o mais simples deles. Lincoln, 1970 propôs a paralelização da análise léxica de um compilador Fortran por meio da quebra do código fonte em partes menores. Para isso, o algoritmo proposto mapeia informações de final de linha e de posicionamento de cada caractere e realiza um processamento através de operações em APL. Posteriormente, Krohn, 1975 estendeu o trabalho anterior, propondo uma maneira de realizar a análise sintática, a tradução para a AST, e a geração de código utilizando os registradores vetoriais do supercomputador STAR-100. Ambos os artigos não apresentam nenhuma análise assintótica para os algoritmos propostos, e tampouco apresentam experimentos.

Posteriormente, Mickunas e Schell, 1978 propôs uma paralelização do algoritmo LR(0). Sua ideia consiste em quebrar arbitrariamente a entrada em vários segmentos e executar um analisador em paralelo em cada um destes segmentos. Estes analisadores (exceto o primeiro, que executa no início da entrada) são modificados na tentativa de evitar conflitos, uma vez que modificar a ordem de início da análise pode impossibilitar a distinção entre o prefixo e o sufixo de uma fase. Quando um conflito é detectado, o analisador envia a sequência de *tokens* lida até então ao analisador à sua esquerda, retorna ao estado inicial, e reinicia a sua análise. No termino da análise, cada analisador envia o resultado para o analisador a sua esquerda, acumulando o resultando no primeiro. Os autores não mostraram experimentos ou análise assintótica do algoritmo proposto. Pennello e DeRemer, 1978 implementou essa estratégia em um compilador Pascal, mas seus experimentos foram realizados em um computador paralelo simulado.

Mark Thierry Vandevoorde, 1988 paralelizou o *Titan C compiler*, um compilador C escrito em Modula-2. Sua implementação consiste na execução de um analisador léxico em paralelo a um analisador sintático, sendo os *tokens* alimentados através de um *pipeline*. O autor também propôs a paralelização de um analisador sintático recursivo descendente

que, após a sequência de declarações do programa, executa novas *threads* para cada expressão da gramática e tem como premissa que qualquer declaração de funções do programa está no cabeçalho do arquivo. Essa estratégia de granularidade fina necessitou a implementação de uma maneira de controlar o paralelismo, onde foi utilizado o conceito de WorkCrews (Mark T Vandevoorde e Roberts, 1988) que limita o número máximo de *threads* em execução simultânea. O autor relata um ganho de 10% no uso do *pipeline* entre o analisador léxico e sintático, e um *speedup* de até 3.1× utilizando 5 processadores MicroVAX II em memória compartilhada. Entretanto, tal compilador não possui estágios de otimização, e assim, métodos de paralelização de um otimizador não foram discutidos neste trabalho.

De maneira similar, Wortman e Junkin, 1992 implementou um compilador de Modula-2+ paralelo. A estratégia consiste em utilizar um analisador léxico capaz de encontrar funções no código fonte e prosseguir com a compilação em paralelo nesse nível, gerando uma tarefa para cada função. Para solucionar problemas relacionados à chamada de funções e acesso a símbolos ainda não definidos, o autor propõe o uso de uma lógica ternária na tabela de símbolos, especificando um estado "*Doesn't know yet*" (Ainda não visto), e propõe três estratégias para implementar essa funcionalidade. Em seguida, o autor propõe o uso de um número fixo de *threads* trabalhadoras, e também o uso de uma fila de prioridade produtor consumidor, onde as *threads* deverão retirar o trabalho. O uso de uma fila de prioridade permite que as funções mais longas sejam processadas primeiro. Os experimentos conduzidos pelo autor mostraram *speedups* variando de 1.5× até 6× utilizando 8 processadores MicroVAX II em memória compartilhada.

Os artigos apresentados até então não abordam questões relacionadas a otimização de código. Isso porque os compiladores naquela época possuíam poucos passos de otimização, o que implicava em pouco tempo necessário para fazê-la. Mesmo assim, alguns pesquisadores dedicaram-se a estudar esses passos. Lee e Ryder, 1994 estudaram paralelismo na análise de controle de fluxo. Como justificativa para esse estudo, os autores afirmam que a maior parte do tempo gasto por compiladores otimizadores é com este tipo de análise, principalmente nas otimizações Inter Procedurais. A investigação se concentrou em algoritmos que usam o grafo de controle de fluxo como entrada. Os autores propõem uma heurística de aglutinação para particionar tal grafo em regiões conexas e adjacentes de tamanho não maior que um certo inteiro s. O uso da heurística é por razão do problema de de particionamento ser NP-difícil. Os experimentos da implementação utilizaram 8 processadores do computador iPSC/2 em memória distribuída, e obtiveram *speedups* variando de 2.8× até 6.5×. Os autores argumentaram que o *speedup* foi limitado devido às características dos programas no teste: os arquivos consistiam de várias funções pequenas, o que normalmente é considerado uma boa prática de programação.

Uma outra forma de explorar o paralelismo utilizando um Grafo Dirigido Acíclico foi proposta por Kramer *et al.*, 1994. Os autores mostram formas de propagar em paralelo as informações nesse grafo explorando seus caminhos independentes, e mostra uma maneira de transformar um Grafo de Controle de Fluxo com laços em um Grafo de Controle de Fluxos acíclico equivalente. Os experimentos realizados pelos autores apenas mostram oportunidades de paralelismo encontrada pelo algoritmo.

Após estes resultados, houve um aparente hiato nas pesquisas relacionadas a com-

pilação em paralelo. As possíveis razões são a complexidade de um compilador como programa, e o modelo dos processadores daquela época, que ainda eram majoritariamente sequenciais. Uma década depois, houve uma explosão de processadores paralelos e do poder computacional como um todo, como apresentado na Seção 2.3. Este ganho computacional levou às empresas a propor soluções ainda mais complexas para gerar código mais eficiente. Embora apresentadas como maneiras de otimizar o código dado como entrada, há relação nessas soluções com paralelismo em compiladores.

Briggs *et al.*, 2007 propuseram uma alteração no compilador GCC com a finalidade de possibilitar otimizações no programa como um todo. A alteração consiste em construir um grafo de chamadas de função em relação a todo o programa, ao contrário do modelo clássico, em que cada módulo é compilado independentemente. Esse processo é composto por três etapas:

1. *Local Generation* (LGEN). Cada função do programa é compilada na Linguagem Intermediária, em conjunto com o grafo de chamadas de função. Esse estágio pode ser executado em paralelo.

2. *Whole Program Analysis* (WPA). O grafo de chamadas de função global é construído e as otimizações a serem efetuadas são selecionadas. Esse estágio deve ser executado sequencialmente.

3. *Local Transformations* (LTRANS). Todas as decisões aplicadas na etapa 2 são implementadas localmente e o código objeto final é gerado. Esse estágio pode executar em paralelo.

A Figura 3.1 retrata este processo. Primeiro, cada arquivo é compilado para a Linguagem Intermediária do Compilador (no GCC, a GIMPLE), em conjunto com o seu Grafo de Chamadas de Função, gerando arquivos objeto `lto_obj*.o`. Esse passo ocorre em paralelo para cada arquivo. Em sequência, o gcc_wpa é chamado, unindo todos os grafos de chamada de função e decidindo como aplicar cada otimização Inter Procedural para cada função do programa. Em seguida, o gcc_ltrans é chamado para cada partição do Grafo de Chamadas de Função, gerando os arquivos objetos para que o LD por fim gere o binário final.

Essa proposta foi implementada no GCC (Glek e Hubicka, 2010) com o nome de *Link Time Optimization* (LTO). A granularidade escolhida na etapa LGEN foi paralelismo por arquivo. Já na LTRANS a granularidade escolhida foram partições do arquivo objeto, o que possibilitou, como consequência, a otimização de funções de um arquivo em paralelo. Os autores testaram essa implementação com o código fonte do Mozilla Firefox e do GCC, e relataram que o GCC construiu binários menores e mais rápidos. Os autores relataram que essa implementação tornou o processo de autocompilação do GCC duas vezes mais lenta, mas que também houve um ganho de 20 minutos na compilação do Firefox. Em todos os casos relatados, os passos de otimização Intra Procedural consumiu mais tempo. Liška, 2014 completa a análise dessa funcionalidade. Um dos problemas encontrados foi o uso elevado de memória para carregar o grafo de chamadas de função global necessário na fase WPA, que atingiu 12Gb na compilação do Chromium e 15Gb na compilação do Kernel Linux. A maior crítica desse processo foi o tempo necessário para recompilar todo o programa quando uma simples alteração era efetuada em um único arquivo, o que inviabilizava um desenvolvimento incremental.

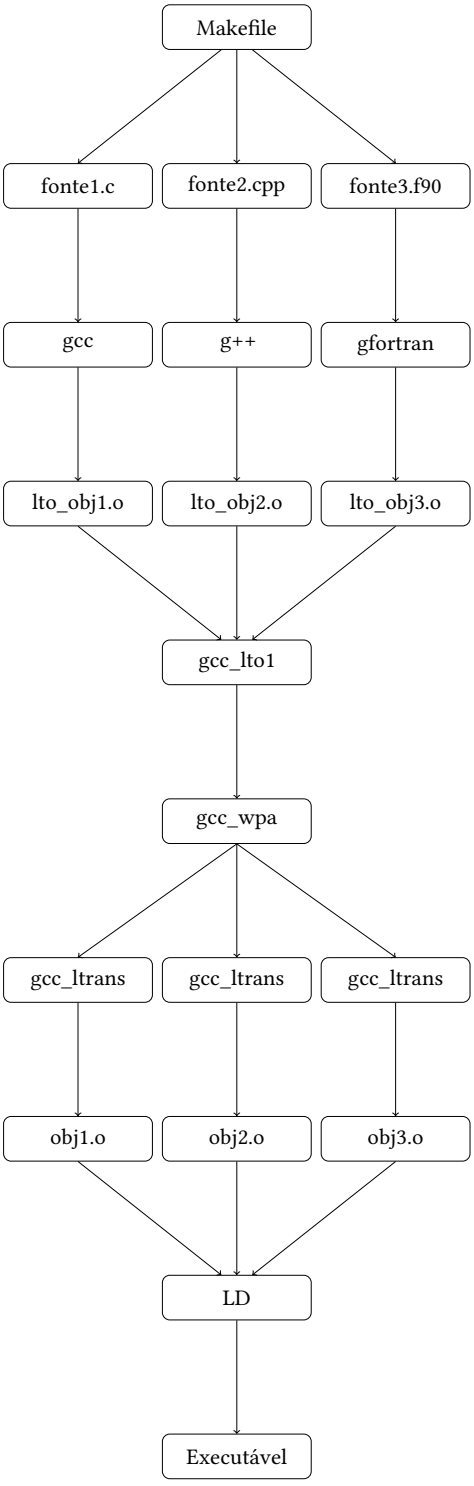


Figura 3.1: *Compilação de um programa utilizando WHOPR.*

Considerando o alto uso de memória e tempo de compilação nos casos relatados acima, Johnson *et al.*, 2017 propôs o ThinLTO. A ideia é considerar apenas parte do grafo global a cada vez, evitando carregá-lo completamente na memória; e postergar ao máximo aplicar as otimizações possíveis, realizando apenas análises de maneira sequencial e aplicando-as no *backend* em paralelo. Os autores relataram um *speedup* de até 15× quando comparado a implementação do LTO original do Clang e até 4× quando comparado a implementação do LTO do GCC. Os autores também relataram uma diminuição considerável no consumo de memória quando comparado ao LTO do GCC: Redução de 2.9Gb para 1Gb na compilação do Clang, 10.8Gb para 1.0Gb no Chromium, e permitiu melhores otimizações no *Ad Delivery* da Google, enquanto a implementação do LTO no GCC abortava a compilação após consumir mais de 25Gb de memória. O desempenho dos binários gerados também foi similar, inclusive onde o LTO original gerava binários mais lentos que o processo clássico de compilação. O tempo de compilação também melhorou significativamente quando comparado ao LTO original, mas permaneceu pior em todos os casos quando comparado ao processo clássico de compilação.

Houve também nas pesquisas um renascimento de análise sintática em paralelo. Barenghi *et al.*, 2015 encontrou uma maneira de explorar a propriedade de Análise Local em Gramáticas com Precedência de Operadores. Isso permitiu a construção de um gerador de analisadores sintáticos PAPAGENO (*Parallel Parser Generator*). Para isto, a gramática a ser utilizada nesse analisador precisa ser convertida para uma Gramática com Precedência de Operadores, conforme discutido e exemplificado pelos autores. Testes empíricos foram efetuados implementando um analisador JSON e outro analisador para a linguagem Lua. Em um Opteron 8378 (16 núcleos), os autores atingiram um *speedup* de até 5.3× no análisador JSON em arquivos grandes, e pouco mais de 4× no analisador Lua. O algoritmo paralelo foi comparado com os gerados pelo analisador sintático *Bison*, que apenas gera código sequencial.

Atualmente, os compiladores são *softwares* bem mais complexos que aqueles apresentados por Mark T Vandevoorde e Roberts, 1988 e Wortman e Junkin, 1992, havendo muito mais passos de otimização, enquanto houve pouquíssima ou nenhuma adição de complexidade nos tópicos de análise léxica, sintática, e geração de código o que esses trabalhos procuraram explorar. Já os trabalhos de Lee e Ryder, 1994 e Kramer *et al.*, 1994 são relevantes por mostrar estratégias de paralelização de otimizações Inter Procedurais, entretanto elas não são a maioria das otimizações implementadas nos compiladores. Por outro lado, o LTO e o ThinLTO não foram projetados para serem usadas em desenvolvimento incremental, além de gerar binários menos eficientes em alguns casos. Efetuando a paralelização do compilador internamente (usando *threads*, por exemplo) a estrutura original do compilador é preservada, evitando esse problema.

Capítulo 4

Proposta de Trabalho

Neste capítulo é apresentado um plano de trabalho que será executado após o período de qualificação, bem como resultados esperados. Este trabalho tem o objetivo de trazer duas contribuições:

1. Uma revisão sobre o que pode ser feito para acelerar o processo clássico de compilação em máquinas *manycore* através de paralelismo, e quais problemas devem ser atacados em trabalhos futuros.

2. Uma opção no compilador GCC para que ele seja capaz de compilar um único arquivo em paralelo sem utilizar a estrutura do LTO. Isso é útil para desenvolvedores que trabalham com grandes softwares iterativamente, pois deve minimizar gargalos gerados por arquivos grandes e cobrir os casos onde o LTO gera binários menos eficientes, fornecendo assim uma alternativa a essa tecnologia. Este trabalho se concentra em paralelizar os passos de otimização Intra Procedural a nível de funções, ou seja, duas análises executarão em paralelo em funções distintas, evitando adentrar em paralelizar a análise de fluxo de dados. Espera-se utilizar as informações do item 1 para realizar um estudo de caso.

O segundo item será enviado ao GCC como uma contribuição ao *software* livre. O projeto de paralelização foi submetido e aprovado para o GSoC 2019, conforme o Apêndice A.

4.1 Paralelização do GCC com *threads*

Nesta subseção é apresentado o estado atual do projeto de paralelização do GCC com *threads*. O GCC foi escolhido como candidato para paralelização pois a comunidade demonstrou grande interesse no projeto, conforme discutido na Seção 1, e também devido a familiaridade do autor com o projeto, já tendo previamente contribuído com este. Sendo assim, utilizar outro compilador como o Clang para implementar o projeto demandaria estudos sobre a estrutura do projeto e convencimento da comunidade das possíveis vantagens deste trabalho ao projeto. Por outro lado, implementar um novo compilador não é uma alternativa viável dado que um dos maiores gargalos na compilação é o otimizador, peça que não seria possível implementar em dois anos com o mesmo poder das já existentes

no GCC ou no Clang.

Como atestado por Liška, 2018, há um gargalo de paralelismo dentro do próprio GCC por conta de arquivos grandes. Outro gargalo também foi relatado por Cheng, 2018 em outro projeto interno. Uma das soluções para este problema é melhorar o paralelismo dentro do GCC, tornando possível fazer com que a compilação destes arquivos utilize mais núcleos de processamento. Nesta discussão, foi proposta uma maneira de visualizar o problema de paralelismo através de um gráfico gerado por dados de um GNU Make modificado. Como a alteração no Make é razoavelmente complicada e o *script* proposto possuía sérios problemas de estabilidade, foi desenvolvida uma outra maneira de replicar os resultados.

Como desenvolvido e publicado por Belinassi, 2019, a ferramenta aqui proposta é capaz de coletar e exibir dados referente ao tempo de compilação de cada arquivo no GCC, incluindo os testes gerados pelo GNU Autotools. A ferramenta funciona da seguinte forma: há um programa escrito em C chamado `cc_wrapper` que encapsula o compilador C e C++ do ambiente, no caso o GCC e o G++. O caminho para estes compiladores são passados como um parâmetro da compilação do `cc_wrapper` de maneira que os binários gerados os simulem. Em seguida o programa abre um novo processo através do `fork()`, chamando o GCC/G++ com os parâmetros passados a ele sem alterações. O processo inicia a coleta do tempo, busca pelo nome do arquivo objeto a ser gerado, e aguarda o GCC chamado terminar. Essa busca foi codificada de maneira a ser muito eficiente, tento um pior caso $O(n)$ com uma constante muito baixa, onde n é o número de parâmetros passados ao GCC. Em seguida, o programa escreve o tempo de início, tempo de fim, e o nome do arquivo em um arquivo de texto. Houve cautela para que não haja mistura de linhas no arquivo por razão de escrita simultânea no arquivo. Há também um *script* em *Bash* para reproduzir os resultados facilmente.

Também é disponibilizado pelo autor um programa em *Python* para análise dos resultados, responsável por gerar os gráficos conforme mostrado na Figura 4.3. Em um dos eixos há o tempo de execução, no outro há o trabalho do Makefile. Para construir o gráfico, é utilizado a técnica de coloração de grafos de intervalos: cada nó representa um intervalo de tempo, e é adicionado uma aresta entre esses nós caso haja intersecção nos intervalos. Em seguida, é executado um algoritmo guloso de coloração a partir do tempo de inicio. Como o grafo é de intervalos, esse algoritmo garante que a quantidade de cores utilizadas é a menor possível. Isso dá uma boa estimativa de como os trabalhos foram distribuidos pelo Makefile. Do ponto de vista de complexidade, isso garante que os gráficos podem ser gerados em $O(n \log p)$, onde n é o número de arquivos e p é o número de cores, embora o algoritmo implementado seja $O(np)$.

4.1.1 Investigação do Tempo Consumido na Compilação

Uma investigação foi conduzida com a finalidade de encontrar o gargalo principal no processo de compilação do GCC. Todos os testes efetuados foram executados em um computador com um AMD Opteron 6376 (64 núcleos) executando o Debian 10 (*Testing*).

Primeiro, foi executado um experimento com respeito ao tempo de compilação do GCC com o LTO habilitado, e outro desabilitado. Em ambos os experimentos, o processo de

bootstrap do compilador foi desabilitado. Para o caso LTO, utilizamos no máximo 64 *threads* através da diretiva de compilação `-flto=64` para permitir o uso de paralelismo nesse método. Como é possível concluir comparando as Figuras 4.4 e 4.3, o LTO consome cerca de 1 minuto a mais para compilar todo o programa (mais de 200s do LTO, contra pouco mais de 140s sem o LTO). Sendo assim, desabilitar o LTO mesmo que ele forneça um grau maior de paralelismo é mais vantajoso caso o objetivo seja compilar mais rapidamente.

Analisando o paralelismo do processo clássico na Figura 4.3, é possível notar dois itens, independente da quantidade de execuções do experimento:

- A existência de arquivos como o `gimple-match.c`, que geram um gargalo no paralelismo do GCC.
- Várias etapas sequênciais executadas pelo GNU Autoconf.

O arquivo `gimple-match.c` é gerado automaticamente compilando o arquivo `match.pd` para C. Na versão 9.0.1 do GCC, o arquivo gerado contém exatamente 99329 linhas de código. Espera-se que o tempo de compilação do `gimple-match.c` diminua, em conjunto com o tempo total de compilação, ao paralelizar o GCC com *threads*.

Analisando o tempo necessário para compilar o arquivo `gimple-match.c`, foi possível notar que:

- São necessários em média 76 segundos para compilar tal arquivo.
- 91% desse tempo (69 segundos) é utilizado na etapa de otimização e geração de código final.
- 8% desse tempo (6 segundos) é utilizado na etapa de análise léxica e sintática.
- O outro 1% está distribuído em diversas partes do compilador.

Todos estes dados foram obtidos autocompilando o GCC 9.0.1, e utilizando a ferramenta de *profiling* incorporada no GCC através das flags `-ftime-report -ftime-report-details`.

Como as otimizações do GCC são divididas em IPA, GIMPLE e RTL, é necessário executar uma granularidade mais fina na análise. Com uma simples alteração no GCC, foi possível separar a etapa IPA das demais. Bastou embrulhar as funções `ipa_passes()` e `expand_all_functions()` com duas *timevars* distintas. Assim, os seguintes dados foram obtidos:

- 75% do tempo total de compilação (57s) é gasto nos passos de otimização Intra Procedural e geração de código.
- 11% do tempo total de compilação (11s) é gasto para realizar as IPA.

Sendo assim, o principal candidato a paralelização é a função `expand_all_functions()`, responsável pelas otimizações Intra Procedural e geração de código. Entretanto, para realizar tal paralelização, será necessário documentar e remover diversas variáveis globais do GCC de maneira que seja possível realizar paralelismo com *threads*.

4.1.2 Análise do *Speedup* Máximo

Conforme analisado acima, se paralelizarmos as etapas de otimização Intra Procedural e Geração de Código, é possível paralelizar 75% do tempo total. Portanto, seja p o número de processadores utilizados. Assumindo *speedup* linear, o ganho máximo será:

$$T_p = \frac{1}{4}T_1 + \frac{3}{4p}T_1 = \frac{1}{4}\left(1 + \frac{3}{p}\right)T_1$$

Sendo assim, o *speedup* máximo por arquivo será:

$$\lim_{p\to+\infty} \frac{T_1}{T_p} = \lim_{p\to+\infty} \frac{T_1}{\frac{1}{4}\left(1 + \frac{3}{p}\right)T_1} = \lim_{p\to+\infty} \frac{4}{1 + \frac{3}{p}} = 4$$

Entretanto, é muito provável que o *speedup* a ser obtido seja menor que isto devido a mecanismos de sincronização necessários para implementar o paralelismo.

4.1.3 Arquitetura da Paralelização

A arquitetura da paralelização que será implementada no GCC é similar ao descrito por Wortman e Junkin, 1992: serão executadas um número fixo de *threads* trabalhadoras de acordo com o número de processadores disponíveis no computador. Assim que o IPA finalizar o plano de de otimização, as funções serão alimentadas em uma fila Produtor-Consumidor onde as *threads* trabalhadoras retirarão o trabalho. Cada *thread* será responsável por executar todas as passagens de otimização do GIMPLE em uma função retirada da fila produtor-consumidor. Sendo assim, duas *threads* processarão duas funções distintas. Esse processo está retratado na Figura 4.1. Em seguida, será avaliado o *speedup* ao paralelizar o GIMPLE, e será decidido qual outro problema abordar: paralelização das demais passagens RTL, ou uma tentativa de paralelizar o IPA.

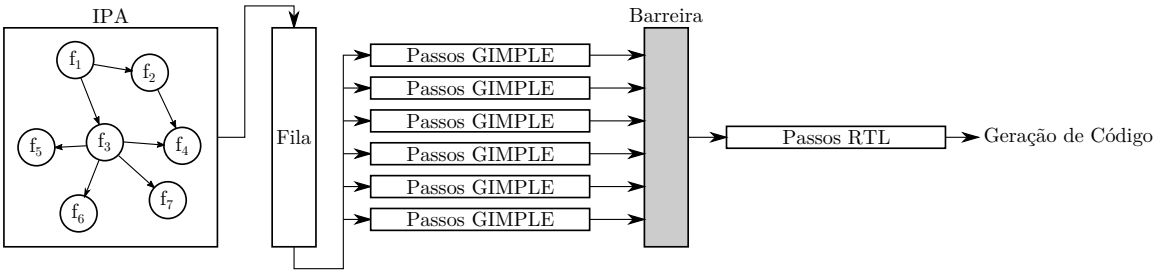


Figura 4.1: *Esquema de paralelização dos passos de otimização.*

4.1.4 Experimentos e Metodologia

Para validar os resultados obtidos, serão executados dois experimentos:

1. Comparação do tempo de compilação do arquivo `gimple-match.c` antes e após a paralelização.
2. Comparação do tempo total de compilação do GCC antes e após a paralelização.

Tal comparação será realizada utilizando técnicas de inferência estatística, conforme descrito por Morettin e BUSSAB, 2017: serão coletadas diversas amostras a respeito do tempo de compilação. Em seguida, será feita alguma suposição a respeito da distribuição das amostras de acordo com o resultado de um gráfico Q-Q para então calcular um estimador para o valor esperado e variância. Com isso, serão calculados os intervalos de confiança sobre a média do tempo de execução para afirmar que ela diminui após a paralelização. Caso não seja possível encontrar uma distribuição adequada, será utilizado o Teorema do Limite Central com um número grande de amostras. Em seguida, esse resultado será comparado com o tempo de compilação utilizando a estrutura LTO, também utilizando as mesmas técnicas estatísticas.

4.1.5 Cronograma

As atividades serão dispostas pelos próximos meses na seguinte forma:

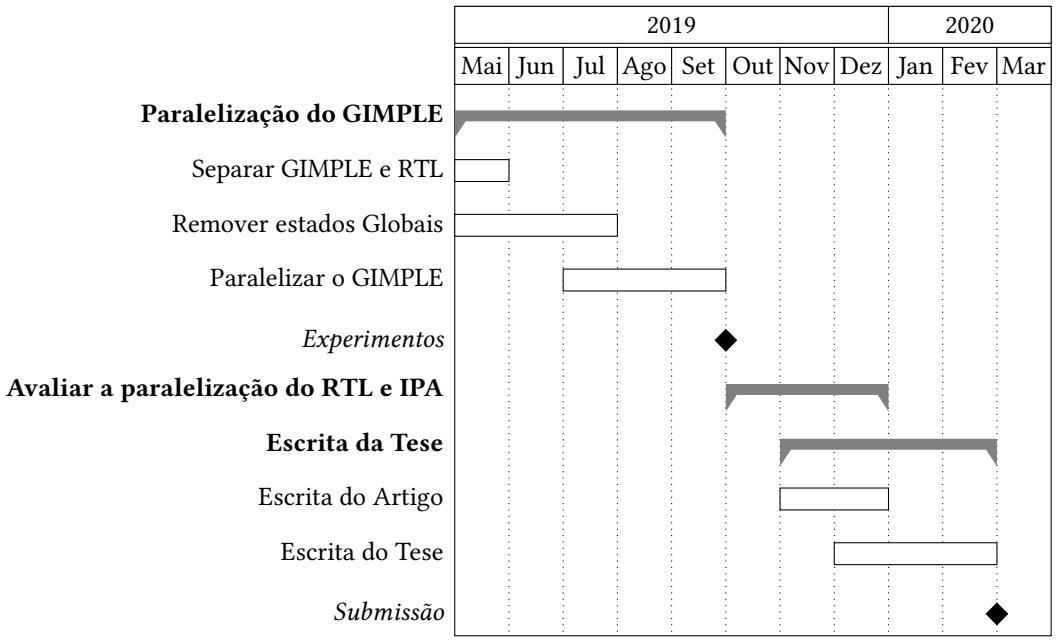


Figura 4.2: *Cronograma das Atividades.*

- Separar GIMPLE e RTL: O RTL tem dependências com o *Back End* do GCC, sendo assim os passos GIMPLE e RTL serão separados para que seja possível paralelizar os passos GIMPLE.
- Remover estados globais: O GCC tem muitos estados globais referentes aos passos GIMPLE que deverão ser adaptados para a paralelização.
- Paralelizar o GIMPLE: Será adicionado código para que os passos do GIMPLE executem em paralelo, e em seguida os experimentos serão executados.
- Avaliar a paralelização do RTL e IPA: Dependendo do ganho obtido na etapa anterior, será avaliado o ganho em paralelizar os passos IPA ou o RTL.
- Escrita de um Artigo Científico. Tempo destinado a escrita de um artigo a respeito dos resultados do trabalho.

- Escrita da Dissertação: Tempo destinado a escrita do texto final da dissertação.

Uma ilustração da distribuição dessas tarefas no tempo está representada na Figura 4.2.

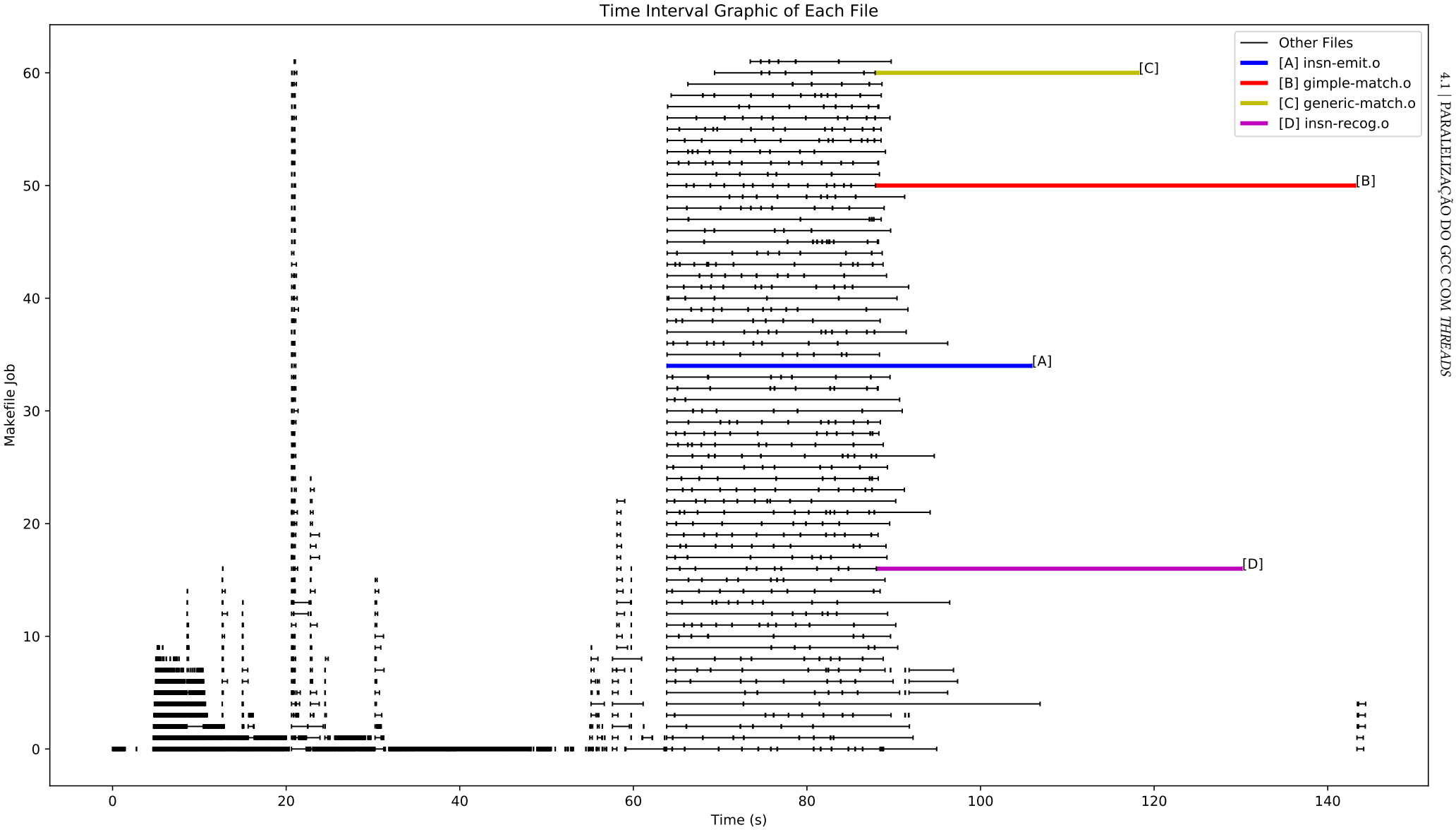


Figura 4.3: *Tempo corrido na compilação do GCC em um processador de 64 núcleos.* ***Sem*** *LTO, sem Bootstrap.*

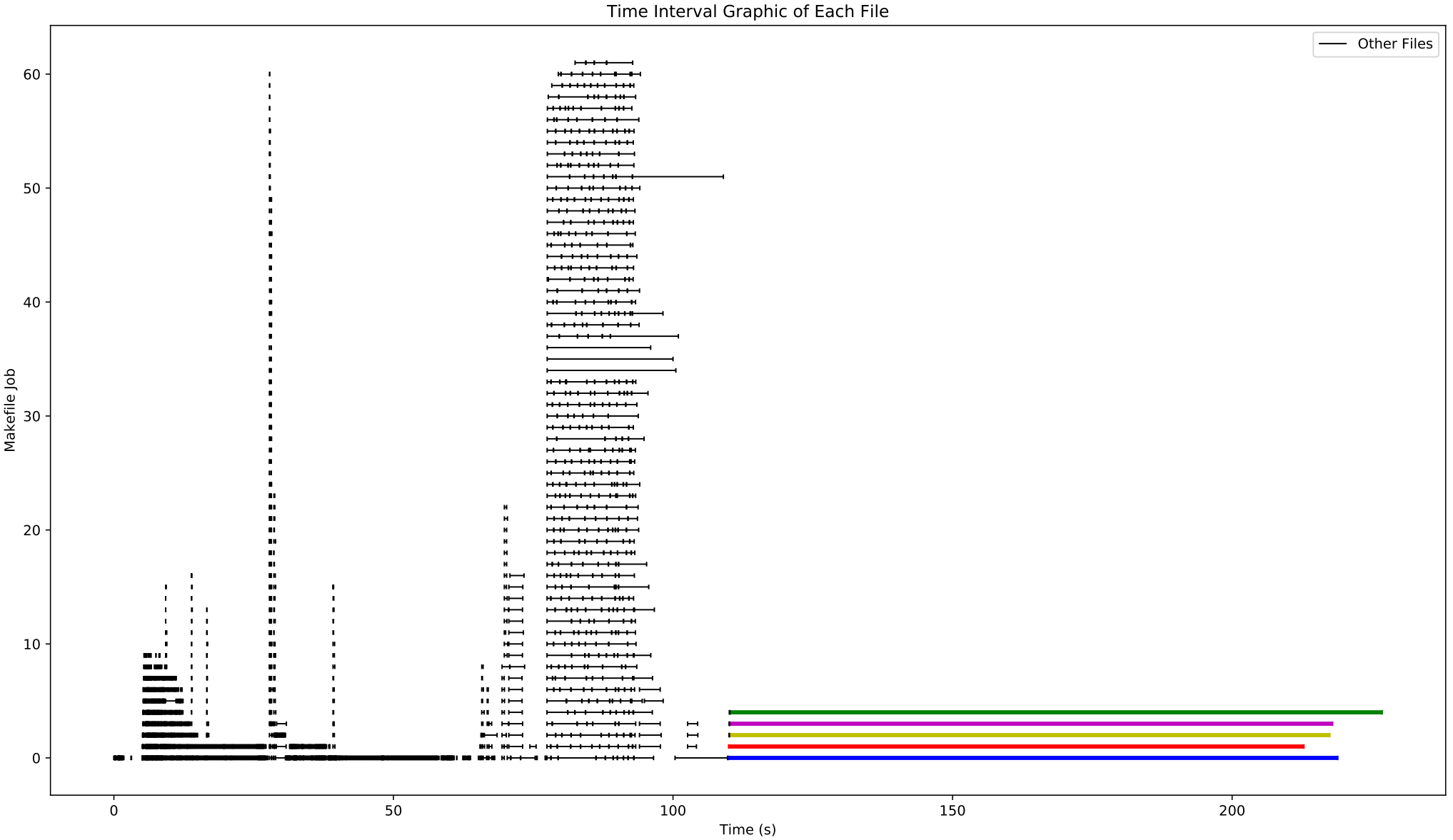
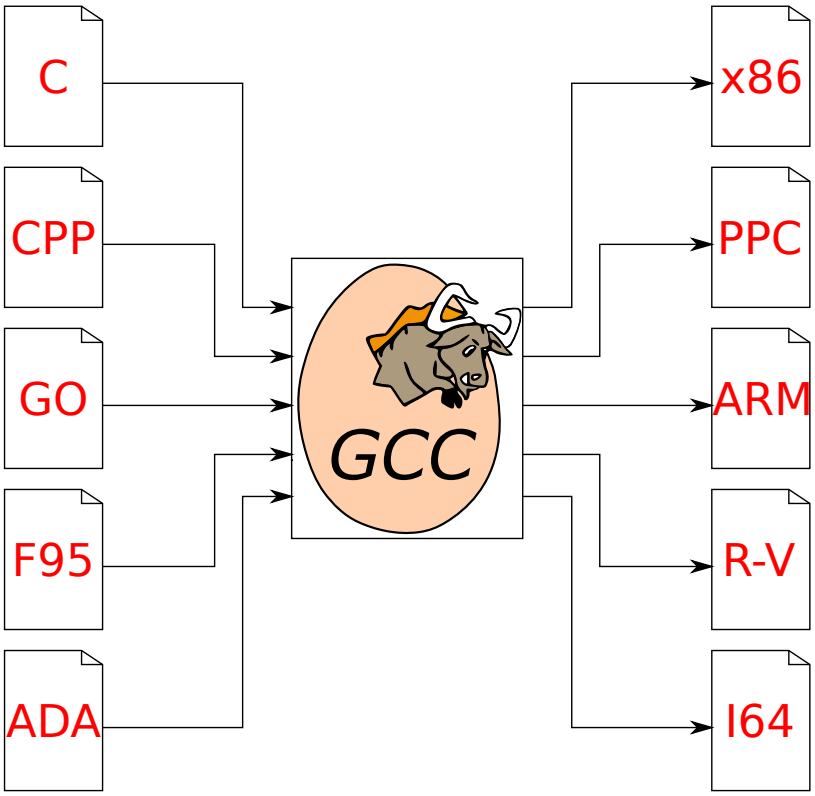


Figura 4.4: *Tempo corrido na compilação do GCC em um processador de 64 núcleos.* ***Com*** *LTO, sem Bootstrap.*

Apêndice A

GSoC Proposition



Giuliano Belinassi
Timezone: GMT−3:00
University of São Paulo – Brazil
IRC: giulianob in #gcc
Email: giuliano.belinassi@usp.br
Github: https://github.com/giulianobelinassi/
Date: April 1, 2019

A.1 About Me

Computer Science Bachelor (University of São Paulo), currently pursuing a Masters Degree in Computer Science at the same institution. I've always been fascinated by topics such as High-Performance Computing and Code Optimization, having worked with a parallel implementation of a Boundary Elements Method software in GPU. I am currently conducting research on compiler parallelization and, therefore, a GSoC internship working with the GNU Compiler Collection (GCC) on a related topic is an excellent opportunity for me to become more involved with the Free Software community.

Skills: Strong knowledge in C, Concurrency, Shared Memory Parallelism, command line utilities (grep, sed…) and other typical programming tools.

A.1.1 Contributions to GCC

I've submitted some patches, mainly adding inline optimizations to trigonometric functions. This kind of patch requires exaustive testing to guarantee that the optimization does not yield severely incorrect results. This work resulted in a blog post[1] about the patch *Optimize sin(arctan(x))*. I did this blog post both to register how to add this kind of optimization and also to encourage newcomers to contribute to GCC.

Name	Status
Optimize sin(arctan(x))	`Accepted`
Optimize sinh(arctanh(x))	`Accepted`
Fix typo 'exapnded'	`Accepted`
Split 'opt and gen' variable	`Working on`
Update sin(arctan(x)) test	`Waiting Stage1`
Fix PR89437	Wilco Dijkstra version accepted

Tabela A.1

A.2 Parallelization Project

While looking for topics in the compiler field that touched subjects that I am interested for my masters thesis, I found a parallelization project proposed in GSoC 2018. With this in mind, I started a discussion in the mailing list to understand what that project is about, which was parallelizing GCC internals to be able to compile big files faster[2]. As can be seen in the discussion, I started to work on this subject way before the list of GSoC accepted organizations was made public.

As stated in PR84402[3], there is a parallelism bottleneck in GCC concerning huge files (with hundreds of thousands of lines of code). In the course of the discussion, Bin Cheng[4]

[1] https://flusp.ime.usp.br/gcc/2019/03/26/making-gcc-optimize-some-trigonometric-functions/
[2] https://gcc.gnu.org/ml/gcc/2018-11/msg00073.html
[3] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84402
[4] https://gcc.gnu.org/ml/gcc/2018-12/msg00079.html

reported that he is affected by this issue, stating that parallelizing the compiler may solve his problem. These discussions demonstrate the community interest in this project.

A.2.1 Current Status

In PR84402, Martin Liška posted a graphic showing the existence of a parallelism bottleneck in GCC compilation due to huge files such as `gimple-match.c` in a 128-cores machine that `make -j128` alone could not alleviate. He also posted an amazing patch to GNU Make to collect the data and a script to plot the graphic, which I used to reproduce the same behaviour in a 64-cores machine that is available at my university.

Unfortunately, I found this approach not easy to replicate, as it requires compiling and installing a custom version of Make and generates gigabytes of data which require parsing by a script that often crashes, as it struggles to generate a very large SVG. With this in mind, I created a set of tools[5] using a completely distinct approach, which generates less data, is more stable, and plots better graphics, such as the one in Figure A.1.

I also explored the GCC codebase in order to find the performance bottleneck for such huge files. I compiled GCC with the `-disable-checking` and `-O2` flags, which give a more performant (but less reliable) compiler and used this version to compile `gimple-match.c` (the largest file in GCC). In this context, I found that the method `finalize_compilation_unit()` of class `symbol_table` takes around 50*s* of the compilation time, with the `expand_all_functions` routine taking most of it. Therefore, this routine is a strong candidate for parallelization.

Currently, my strategy to parallelize `expand_all_functions()` is to perform the `GIMPLE` processing step in parallel, as suggested by Richard Biener[6]: each function may be independently processed by the many passes of `GIMPLE`. There is also a pipelined alternative: a distinct thread is responsible for a given `GIMPLE` pass and each function is fed sequentially to each of them. This allows many functions to be processed simultaneously, each by a different pass.

Furthermore, I am also studying the theoretical background behind `GIMPLE` and `cgraphs`. I have read the `GIMPLE` documentation[7] and I am looking at how `cgraph` works internally, both in theory and within GCC.

A.2.2 Planned Tasks

I plan to use the GSoC time to develop the following topics:

- Week [1, 7] – May 6 to June 21:
  Refactor `cgraph_node::expand()` and `expand_all_functions()`, splitting `IPA`, `GIMPLE` and `RTL` passes, and some documentation about the global states of the `GIMPLE` passes.

[5] https://github.com/giulianobelinassi/gcc-timer-analysis
[6] https://gcc.gnu.org/ml/gcc/2019-03/msg00249.html
[7] https://gcc.gnu.org/onlinedocs/gccint/

- Week 8 – June 24 to 28: **First Evaluation**
  Deliver a patch with the refactored version of both functions mentioned above, plus the partial documentation with regard to `GIMPLE`.
- Week [9, 11] – July 1 to 19:
  Continue documentating and refactoring the `GIMPLE` passes, and prototype a parallelization of `expand_all_functions()`.
- Week 12 — July 22 to 26: **Second Evaluation**
  Attend to Debconf 19 (Curitiba, Brazil). Please let me know if someone else from GCC will attend.
  Deliver a working non-optimized parallel version of `expand_all_functions()` with the point of being correct in a multi-threaded environment.
- Week [13, 15] — July 29 to August 16:
  Iteractively improve the current implementation.
- Week 16 - August 19: **Final evaluation**
  Optimize the current implementation, so that there is a speedup over the sequential version when compiling `gimple-match.c` and reducing the total compilation time in `GCC` compilation without bootstrap.

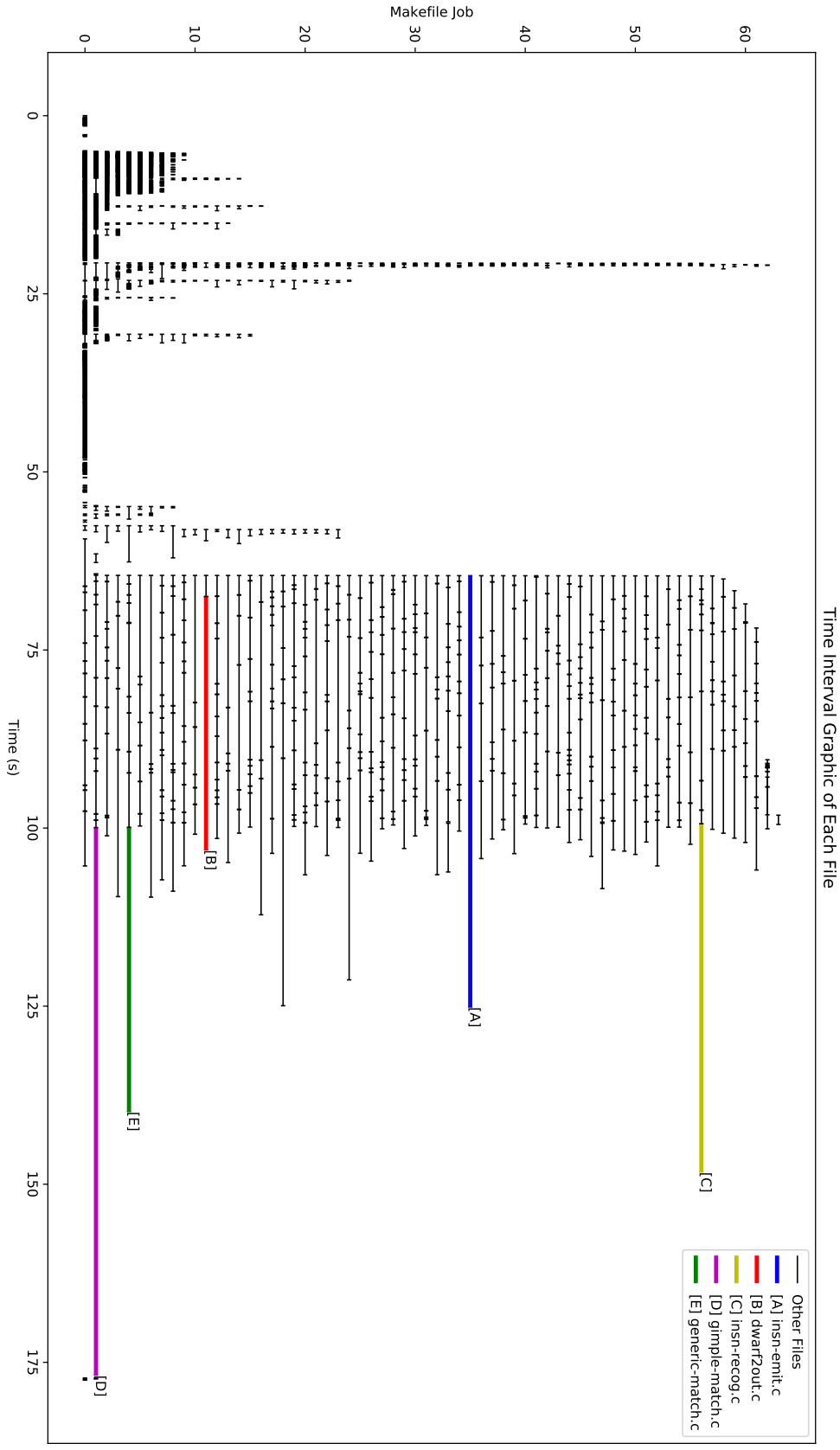


Figura A.1: *Elapsed time analysis in GCC compilation for a 64 cores machine, No bootstrap*

Referências

[*sem_overview Man Page* 2019] *sem_overview Man Page*. 2019 (citado na pg. 18).

[*shmget Man Page* 2019] *shmget Man Page*. 2019 (citado na pg. 18).

[Appel 2004] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004 (citado nas pgs. 8, 9).

[atomsymbol 2019] atomsymbol. *Phoronix: GCC's Potential GSoC Projects Include Better Parallelizing The Compiler*. 2019. url: https://www.phoronix.com/forums/forum/software/programming-compilers/1078118-gcc-s-potential-gsoc-projects-include-better-parallelizing-the-compiler?p=1078145%5C#post1078145 (citado na pg. 3).

[Barenghi *et al.* 2015] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella e Matteo Pradella. "Parallel parsing made practical". Em: *Sci. Comput. Program.* 112.P3 (nov. de 2015), pgs. 195–226. issn: 0167-6423. doi: 10.1016/j.scico.2015.09.002. url: https://doi.org/10.1016/j.scico.2015.09.002 (citado na pg. 25).

[Belinassi 2018] Giuliano Belinassi. *PR86829 Missing sin(atan(x)) optimization*. 2018. url: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86829 (citado na pg. 12).

[Belinassi 2019] Giuliano Belinassi. *GCC Compilation Timing Analysis*. 2019. url: https://github.com/giulianobelinassi/gcc-timer-analysis (citado na pg. 28).

[Blindell 2016] Gabriel Hjort Blindell. *Instruction selection: principles, methods, and applications*. Springer, 2016 (citado nas pgs. 11, 12).

[Briggs *et al.* 2007] Preston Briggs *et al. WHOPR - Fast and Scalable Whole Program Optimizations in GCC*. 2007. url: https://gcc.gnu.org/projects/lto/whopr.pdf (citado na pg. 23).

[Cheng 2018] Bin Cheng. *Paralelize the compilation with Threads*. 2018. url: https://gcc.gnu.org/ml/gcc/2018-12/msg00079.html (citado nas pgs. 3, 28).

[Cytron *et al.* 1991] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman e F Kenneth Zadeck. "Efficiently computing static single assignment form and the

control dependence graph". Em: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pgs. 451–490 (citado na pg. 9).

[Dijkstra 1965] Edsger W Dijkstra. "Over de sequentialiteit van procesbeschrijvingen". Em: (1965) (citado na pg. 18).

[GAO 2017] GAO. *Management Attention Is Needed to Successfully Modernize Tax Processing Systems*. 2017. URL: https://web.archive.org/web/20180924110604/https://waysandmeans.house.gov/wp-content/uploads/2017/10/20171004OS-Testimony-Powner.pdf (citado na pg. 1).

[Glanville e Graham 1978] R. Steven Glanville e Susan L. Graham. "A new method for compiler code generation". Em: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: ACM, 1978, pgs. 231–254. DOI: 10.1145/512760.512785. URL: http://doi.acm.org/10.1145/512760.512785 (citado na pg. 11).

[Glek e Hubicka 2010] Taras Glek e Jan Hubicka. "Optimizing real world applications with gcc link time optimization". Em: *arXiv preprint arXiv:1010.2196* (2010) (citado nas pgs. 3, 23).

[GitHub 2017] GitHub. *GitHub: The state of the Octoverse 2017*. 2017. URL: https://web.archive.org/web/20190215123508/https://octoverse.github.com/2017/ (citado nas pgs. 1, 2).

[GNU 2019a] GNU. *GCC Releases History*. 2019. URL: https://www.gnu.org/software/gcc/releases.html (citado na pg. 12).

[GNU 2019b] GNU. *GCC RTL Representation*. 2019. URL: https://gcc.gnu.org/onlinedocs/gccint/RTL.html (citado na pg. 13).

[GNU 2019c] GNU. *GENERIC Documentation*. 2019. URL: https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html (citado na pg. 12).

[GNU 2019d] GNU. *GIMPLE Documentation*. 2019. URL: https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html (citado na pg. 13).

[Hopcroft 1971] John Hopcroft. "An n log n algorithm for minimizing states in a finite automaton". Em: *Theory of machines and computations*. Elsevier, 1971, pgs. 189–196 (citado na pg. 7).

[Johnson *et al.* 2017] Teresa Johnson, Mehdi Amini e Xinliang David Li. "Thinlto: scalable and incremental lto". Em: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pgs. 111–121. ISBN: 978-1-5090-4931-8. URL: http://dl.acm.org/citation.cfm?id=3049832.3049845 (citado na pg. 25).

[Jeffrey e Ullman 2007] Alfred V Aho Ravi Sethi Jeffrey e D Ullman. *Compilers, Principles, Techniques, and Tools. Second Edition*. 2007 (citado nas pgs. 1, 6).

[Koes e Goldstein 2008] David Ryan Koes e Seth Copen Goldstein. "Near-optimal instruction selection on dags". Em: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2008, pgs. 45–54 (citado na pg. 12).

[Kramer *et al.* 1994] Robert Kramer, Rajiv Gupta e Mary Lou Soffa. "The combining dag: a technique for parallel data flow analysis". Em: *IEEE Transactions on Parallel and Distributed Systems* 5.8 (1994), pgs. 805–813 (citado nas pgs. 22, 25).

[Knuth 1965] Donald E Knuth. "On the translation of languages from left to right". Em: *Information and control* 8.6 (1965), pgs. 607–639 (citado na pg. 7).

[Krohn 1975] Howard E. Krohn. "A parallel approach to code generation for fortran like compilers". Em: *SIGPLAN Not.* 10.3 (jan. de 1975), pgs. 146–152. issn: 0362-1340. doi: 10.1145/390015.808414. url: http://doi.acm.org/10.1145/390015.808414 (citado na pg. 21).

[Khedker *et al.* 2009] Uday Khedker, Amitabha Sanyal e Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009 (citado na pg. 10).

[Lattner e Adve 2002] Chris Lattner e Vikram Adve. "The llvm instruction set and compilation strategy". Em: *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS* (2002) (citado na pg. 8).

[Lincoln 1970] Neil Lincoln. "Parallel programming techniques for compilers". Em: *SIGPLAN Not.* 5.10 (out. de 1970), pgs. 18–31. issn: 0362-1340. doi: 10.1145/987475.987478. url: http://doi.acm.org/10.1145/987475.987478 (citado na pg. 21).

[Liška 2014] Martin Liška. "Optimizing large applications". Em: *arXiv preprint arXiv:1403.6997* (2014) (citado na pg. 23).

[Liška 2018] Martin Liška. *PR84402*. 2018. url: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84402 (citado na pg. 28).

[LLVM 2019a] LLVM. *LLVM Compiler Infrastructure: FAQ*. 2019. url: https://web.archive.org/web/20160410185222/https://www.redhat.com/magazine/002dec04/features/gcc/ (citado na pg. 5).

[LLVM 2019b] LLVM. *The LLVM Target-Independent Code Generator*. 2019. url: https://llvm.org/docs/CodeGenerator.html#instruction-selection-section (citado na pg. 12).

[Love 2005] Robert Love. *Linux Kernel Development (2Nd Edition) (Novell Press)*. Novell Press, 2005. isbn: 0672327201 (citado na pg. 18).

[Lee e Ryder 1994] Yong -Fong Lee e Barbara G. Ryder. "Effectively exploiting parallelism in data flow analysis". Em: *The Journal of Supercomputing* 8.3 (nov. de 1994), pgs. 233–262. ISSN: 1573-0484. DOI: 10.1007/BF01204730. URL: https://doi.org/10.1007/BF01204730 (citado nas pgs. 22, 25).

[Lumsdaine *et al.* 2007] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson e Jonathan Berry. "Challenges in parallel graph processing". Em: *Parallel Processing Letters* 17.01 (2007), pgs. 5–20 (citado na pg. 3).

[Morettin e BUSSAB 2017] Pedro Alberto Morettin e WILTON OLIVEIRA BUSSAB. *Estatística básica*. Editora Saraiva, 2017 (citado na pg. 31).

[McKenney 2018] Michael E. McKenney. *Review of the System Failure That Led to the Tax Day Outage*. 2018. URL: https://web.archive.org/web/20180924110604/https://waysandmeans.house.gov/wp-content/uploads/2017/10/20171004OS-Testimony-Powner.pdf (citado na pg. 2).

[Mickunas e Schell 1978] M. Dennis Mickunas e Richard M. Schell. "Parallel compilation in a multiprocessor environment (extended abstract)". Em: *Proceedings of the 1978 Annual Conference*. ACM '78. Washington, D.C., USA: ACM, 1978, pgs. 241–246. ISBN: 0-89791-000-1. DOI: 10.1145/800127.804105. URL: http://doi.acm.org/10.1145/800127.804105 (citado na pg. 21).

[Novillo 2016] Diego Novillo. *From Source to Binary: The Inner Workings of GCC*. 2016. URL: https://web.archive.org/web/20160410185222/https://www.redhat.com/magazine/002dec04/features/gcc/ (citado na pg. 5).

[Pacheco 2011] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011 (citado na pg. 17).

[Pennello e DeRemer 1978] Thomas J. Pennello e Frank DeRemer. "A forward move algorithm for lr error recovery". Em: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: ACM, 1978, pgs. 241–254. DOI: 10.1145/512760.512786. URL: http://doi.acm.org/10.1145/512760.512786 (citado na pg. 21).

[Ripken 1977] Rip771 K Ripken. "Formale Beschreibung von Maschinen, Implementieurungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen". Tese de dout. PhD Dissertation, Technische Universitat Munchen, Munich, West Germany, 1977 (citado na pg. 11).

[Ritchie 1979] Dennis M Ritchie. "A tour through the unix c compiler". Em: *UNIX Programmer's Manual* 2 (1979) (citado na pg. 6).

[Rupp 2018] Karl Rupp. *42 Years of Microprocessor Trend Data*. 2018. URL: https://github.com/karlrupp/microprocessor-trend-data (citado nas pgs. 16, 17).

[Sipser 2012] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012 (citado na pg. 7).

[Thompson 1968] Ken Thompson. "Programming techniques: regular expression search algorithm". Em: *Commun. ACM* 11.6 (jun. de 1968), pgs. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: http://doi.acm.org/10.1145/363347.363387 (citado na pg. 7).

[Mark Thierry Vandevoorde 1988] Mark Thierry Vandevoorde. *Parallel compilation on a tightly coupled multiprocessor*. Systems Research Center, 1988 (citado na pg. 21).

[Mark T Vandevoorde e Roberts 1988] Mark T Vandevoorde e Eric S Roberts. "Workcrews: an abstraction for controlling parallelism". Em: *International Journal of Parallel Programming* 17.4 (1988), pgs. 347–366 (citado nas pgs. 22, 25).

[Wortman e Junkin 1992] D.B. Wortman e M.D. Junkin. "A concurrent compiler for modula-2+". Em: 27 (jan. de 1992), pgs. 68–81. DOI: 10.1145/143103.120025 (citado nas pgs. 22, 25, 30).