TaskMiner: Identificação Automática de Tarefas

Pedro Ramos, Gleison Souza, Guilherme Leobas, Fernando Magno Quintão Pereira
UFMG
[pedroramos,gleison.mendonca,guilhermel,fernando]@dcc.ufmg.br

Abstract

Este artigo apresenta TaskMiner, uma ferramenta que encontra paralelismo de tarefas em código C automaticamente. TaskMiner lida com problemas clássicos de paralelismo irregular como a delimitação simbólica de regiões de memória, remoção de dependências estáticas espúrias, detecção de condições de corrida, e, principalmente, a avaliação do custo do paralelismo. Para solucioná-los, TaskMiner implementa uma série de otimizações em representação intermediária do LLVM, que analisam código C e inserem anotações OpenMP automaticamente. TaskMiner prova-se eficaz ao encontrar paralelismo apontado manualmente em benchmarks como BSC-Bots [5], produzindo programas tão rápidos ou mais velozes que as versões sequenciais e anotadas manualmente. TaskMiner também é capaz de encontrar paralelismo escondido em programas sequenciais, sem intervenção humana.

Keywords paralelismo, tarefas, OpenMP, compiladores

ACM Reference format:
Pedro Ramos, Gleison Souza, Guilherme Leobas, Fernando Magno Quintão Pereira. 2018. TaskMiner: Identificação Automática de Tarefas. In *Proceedings of IEEE, Brasil, 2018 (SBLP'18),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introdução

É inegável que computação paralela e distribuída é hoje uma alternativa muito procurada por desenvolvedores na busca por desempenho. Programadores tem, cada vez mais, investido em paradigmas paralelos visando maior eficiência e velocidade. Nesse contexto, sistemas de anotação como OpenMP [10] e OpenACC [21] atuam como metalinguagens, conferindo ao programador poder semântico para transformar um programa originalmente sequencial em um programa paralelo. Tais sistemas mostram-se muito promissores quando combinados com aceleradores como GPUs e FPGAs [14, 18]. Entretanto, apesar de conveniente, o uso desses sistemas não é automático e requer exaustiva escrutinação de olho humano para garantia de corretude e eficiência.

Embora existam ferramentas que anotem programas automaticamente [13, 16], essas tecnologias exploram somente paralelismo de dados. Porém, grande parte do poder de sistemas de anotação provém da capacidade de se criar *tarefas* durante a execução do programa [2]. O propósito desse trabalho é apresentar um algoritmo capaz de minerar oportunidades para paralelismo de tarefas em programas sequênciais e anotá-los automaticamente. Como descrito na seção 2, essas anotações conferem ao programa irregular uma semântica de paradigmas paralelos, como aqueles em grafos e/ou em *worklists* [16].

Este artigo descreve TaskMiner, um compilador fonte-a-fonte que expõe paralelismo de tarefas em programas C/C++. TaskMiner lida com desafios relacionados ao paralelismo irregular. Primeiro, ele determina os limites simbólicos dos blocos de memória acessados dentro de cada programa. Depois, encontra fragmentos de código (laços ou funções) que podem ser efetivamente mapeados em tarefas. Então, extrai parâmetros capazes de estimar o custo da tarefa em questão. Esses parâmetros estão dinamicamente atrelados às anotações e são capazes de ativar ou desativar a criação de tarefas durante a execução do programa, ou de limitar a profundidade de tarefas geradas recursivamente. TaskMiner também determina quais variáveis serão replicadas em novas tarefas ou compartilhadas por tarefas concorrentes. Finalmente, ele mapeia todas essas informações de volta ao código fonte, produzindo anotações legíveis.

Defende-se a tese de que a anotação automática de tarefas é útil à medida que retira do programador o fardo de procurar, anotar e verificar as anotações, tornando seu processo de desenvolvimento mais eficiente. A ferramenta TaskMiner é capaz de alcançar ganhos de quase 4x em benchmarks não usualmente paralelos, como descrito na Seção 4. TaskMiner recebe como entrada um programa escrito em C e produz como saída um programa também em C, mas anotado com diretivas OpenMP legíveis. TaskMiner anota programas não triviais, envolvendo todo tipo de construto da linguagem C, como arranjos, estruturas, uniões e ponteiros para esses tipos. Alguns desses programas provém de benchmarks complexos como Bots [5] e a coleção de testes do LLVM. As versões anotadas pelo TaskMiner se aproximam das versões anotadas manualmente em desempenho e são mais rápidas que as versões sequenciais.

2 Visão geral

A construção de um compilador como o TaskMiner envolve a superação de uma série de desafios. TaskMiner recebe como entrada um programa escrito em C e retorna o mesmo programa anotado com diretivas OpenMP. O ambiente de execução OpenMP se encarrega de rastrear dependências entre ponteiros [12]. Entretanto, a inserção automática das

SBLP'18, Brasil
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

anotações é apenas a última etapa. Nesta Seção detalha-se os problemas encontrados pelo TaskMiner no percalço de mineração de tarefas. A solução para esses problemas é descrita na Seção 3.

O primeiro desafio encontrado pelo TaskMiner consiste em identificar as regiões de memória cobertas por uma tarefa. A Figura 1 ilustra esse problema e como resolvê-lo. O programa recebe uma matriz V, M × N, em formato linear, e produz um vetor U de forma que U[i] contenha a soma de todos os elementos na linha i da matriz V. TaskMiner determina que cada iteração do laço mais externo pode se tornar uma tarefa. Então, cada tarefa neste programa consiste no laço mais interno, e atravessa a região de memória entre os endereços &V + i * N e &V + i * N + M. A identificação desses limites envolve o uso de álgebra simbólica baseada na literatura de compiladores.

A Figura 1 apresenta um outro problema: *como estimar a rentabilidade de uma tarefa?* A criação de tarefas envolve um custo alto por parte do ambiente de execução, causado por questões como alocação, escalonamento, concorrência e gerenciamento em tempo real do grafo de dependências. Idealmente, são desejáveis tarefas que executem uma quantidade de trabalho grande o suficiente para superar este custo. A quantidade de trabalho executado por uma tarefa não pode ser descoberta estaticamente [19]. Contudo, é possível aproximar-se desse custo com símbolos nativos do programa, que, em execução, serão substituídos por valores. Por exemplo, na Figura 1 sabe-se o número de instruções do laço interno. Então, aproxima-se a quantidade de trabalho executado com a expressão 5 * M. O valor de M durante a execução será determinante para a criação da tarefa. Essa checagem é feita na linha 7 do exemplo, e faz parte da sintaxe de OpenMP. Além disso, TaskMiner propõe uma estimativa confiável a respeito do limite mínimo de custo que uma tarefa deve possuir para ser criada sem produzir excesso de computação.

```
1  int foo(int* U, int* V, int N, int M) {
2    int i, j;
3    #pragma omp parallel
4    #pragma omp single
5    for(i = 0; i < N; i++) {
6      #pragma omp task depend(in: V[i*N:i*N+M]) \
7                                  if (5 * M < CUSTO_MIN)
8      for (j = 0; j < M; j++) {
9        U[i] += V[i*N + j];
10     }
11   }
12   return 1;
13 }
```

```
movl   (%rsi,%rax,4), %r11d
movslq %r9d, %rbx
addl   %r11d, (%rdi,%rbx,4)
incl   %r10d
incl   %eax
```

Figura 1. Identificando limites de memória e o custo de uma tarefa.

Esse limite considera fatores como número de núcleos disponíveis e informações a respeito do ambiente de execução em termos de instruções de máquina.

```
1  static int taskminer_depth_cutoff;
2  long long fib (int n) {
3    taskminer_depth_cutoff++;
4    long long x, y;
5    if (n < 2) return n;
6    #pragma omp task untied default(shared) \
7              if(taskminer_depth_cutoff < PROFUNDIDADE_MAX)
8    x = fib(n - 1);
9    #pragma omp task untied default(shared) \
10             if(taskminer_depth_cutoff < PROFUNDIDADE_MAX)
11   y = fib(n - 2);
12   #pragma omp taskwait
13   taskminer_depth_cutoff--;
14   return x + y;
15 }
```

Figura 2. Limitando a criação de tarefas recursivas. Exemplo retirado de [9, Fig.1].

No paralelismo de tarefas, um fator importante para bom desempenho é o uso de cortes mínimos para o número de tarefas a serem criadas [4]. Observa-se esse fator no contexto de tarefas recursivas. Uma variável de controle é inserida no código para controlar a criação de tarefas de baixa granularidade, como ilustrado na Figura 2. Um contador é associado à invocação das funções recursivas anotadas como tarefas. A checagem na linha 7 certifica-se de que o limite nunca é excedido. Esse limite é pré-determinado na fase de configuração do TaskMiner, assim como o limite de trabalho mínimo por tarefa. Portanto, a geração de código executada pelo TaskMiner é dependente de dois parâmetros, PROFUNDIDADE_MAX e CUSTO_MIN. Ambos possuem valores padrão, mas podem ser configurados pelo usuário.

3 Solução

A Figura 3 descreve o algoritmo executado pela ferramenta TaskMiner. Ele utiliza conceitos como o grafo de dependências [6] e o grafo de fluxo de controle [11]. TaskMiner identifica tarefas em programas estruturados que podem ser particionados em regiões *hammock* [6].

Definição 1 (Região *hammock*)**.** *Um grafo de fluxo de controle G é um grafo direcionado com um nó de entrada s e um nó de saída x. Uma região hammock G' é um subgrafo de G com um nó h que* domina [1] *os outros nós em G. Além disso, existe um nó $w \in G$, $w \notin G'$, tal que w* pós-domina *todo nó em G'. Nessa definição, h é o nó de entrada e w é o nó de saída de G'.*

[1]Um nó $n_1 \in G$ domina outro nó $n_2 \in G$ se todo caminho de S até n_2 atravessa n_1. Inversamente, n_1 pós-domina n_2 se todo caminho de n_2 até x deve passar por n_1.

Definição 2 (Tarefa)**.** *Em um programa P, uma tarefa T é uma tupla* (G', M_i, M_o) *formada por uma região hammock* G' *e dois conjuntos* M_i *e* M_o*, dados de memória lidos e escritos por T, respectivamente.*

Regiões de memória são descritas por variáveis do programa ou ponteiros com intervalos derrefenciáveis. Sejam duas tarefas $T_1 = (G_1, M_{i1}, M_{o1})$ e $T_2 = (G_2, M_{i2}, M_{o2})$, se $M_{o1} \cap M_{i2} \neq \emptyset$ então T_2 *depende* de T_1. Se um programa P está particionado em um conjunto de n tarefas, então ele deve respeitar a Definição 3 para ser considerado correto:

Definição 3 (Corretude)**.** *Um conjunto T de n tarefas é uma paralelização correta de um programa P se:*

1. *T não contém dependências cíclicas;*
2. *A execução das tarefas em T em qualquer ordem determinada pelas relações de dependência devem resultar identicamente à execução sequencial de P.*

Abaixo há um exemplo de como as regiões de memória são marcadas em uma anotação de tarefa. Considere-se duas tarefas $T_{foo} = (\text{foo}, \{\text{v[i} - 1]\}, \{v[i]\})$ e $T_{bar} = (\text{bar}, \{v[i]\}, \{\})$. Dependências são identificadas na cláusula *depend*. Nesse exemplo, cada região *hammock* é constituída por uma chamada de função:

```
#pragma omp task depend(in: v[i-1]) depend(out: v[i])
v[i] = foo(&v[i-1], i);
#pragma omp task depend(in: v[i])
bar(&v[i], i);
```

TaskMiner utiliza OpenMP 4.5 para explorar o paralelismo de tarefas. Detalhes da sintaxe e semântica das anotações do OpenMP estão publicamente disponíveis[2]. As anotações utilizadas são:

- parallel: compõe um time de *threads* para executar a região anotada.
- single: especifica uma região que deve ser executada por apenas uma *thread*.
- task: cria uma nova tarefa.
- taskwait: define um ponto de sincronização.

Algumas dessas anotações são associadas a uma lista de cláusulas, a seguir:

- default([shared/private]): indica se a variável é compartilhada ou replicada entre tarefas.
- untied: indica se uma tarefa não está associada à uma única *thread*, permitindo ao ambiente balancemanto.
- depend(*in*/*out*/*inout*): determina se um dado é lido, escrito ou ambos, explicitando as dependências entre as tarefas.
- if(*condição*): define uma condição para que aquela anotação seja executada.

[2]"Summary of OpenMP 4.5 C/C++ Syntax", disponibilizado pelo grupo OpenMP

```
1  def TaskMiner(Prog, ModeloCusto):
2      CFG = GrafoFluxoControle(Prog)                          3.1
3      PDG = GrafoDependênciaPrograma(Prog)
4      Limites = AnáliseSimbolicaDePonteiros(CFG)              3.2
5      Hélices = EncontrarHélices(PDG)
6      Custo = AnálisedeRentabilidade(Tarefas, Limites, ModeloCusto)   3.3
7      Tarefas = [ ]
8      for v in Hélices:                                       3.4
9          região = v.Expandir(Custo)
10         Tarefas.Adicionar(região)                           3.5
11     Privs = AnáliseDePrivatização(Tarefas, PDG)
12     Prog.Anotar(Tarefas, Custo, Limites, Privs)
```

Figura 3. Os principais passos do algoritmo do TaskMiner.

3.1 Encontrando limites simbólicos

Para produzir uma anotação que criará uma tarefa $T = (G, M_i, M_o)$, deve-se determinar as regiões de memória M_i que T lê, e as regiões M_o em que T escreve. Para tal, recorre-se à análise de limites simbólicos.

A instrução U[i] += V[i*N + j] na linha 9 da Figura 1 contém dois acessos à memória: U[i] e V[i*N + j]. O primeiro cobre a região de memória $[\&\text{U}, \&\text{U} + (\text{M} - 1) \times \text{sizeof(int)}]$. O segundo cobre a região $[\&\text{V} + \text{N} \times \text{sizeof(int)}, \&\text{V} + ((\text{i} \times \text{M} + j) - 1) \times \text{sizeof(int)}]$. Não há código na Figura 1 que dependa do laço nas linhas 8-10. Portanto, uma anotação de tarefa deve considerar apenas a dependência de entrada, ou seja, o acesso à V. A cláusula depend na linha 6 contém uma referência a essa região.

Definição 4 (Análise de Limites Simbólicos)**.** *É uma forma abstrata de interpretação que associa uma variável inteira v a um intervalo simbólico* $R(v) = [l, u]$ *em que l e u são expressões simbólicas. Uma expressão simbólica E é definida pela gramática abaixo, em que s é um símbolo do programa:*

$$E ::= z \mid s \mid E + E \mid E \times E \mid min(E, E) \mid max(E, E) \mid -\infty \mid +\infty$$

Exemplos de símbolos de programa incluem variáveis globais, argumentos de função e retornos de funções externas. A técnica escolhida para as análises do TaskMiner é a mesma do compilador DawnCC [14]. Essa implementação é sólida o suficiente para lidar com toda a linguagem C99, sempre termina e executa em tempo quadrático no número de variáveis do programa.

Por exemplo, se aplicada à Figura 1, a análise de limites resulta em $R(\text{i}) = [0, \text{N} - 1]$, e $R(\text{j}) = [0, \text{M} - 1]$. O acesso à memória V[i*N + j] na linha 9, combinado a essa informação, fornece a região simbólica que aparece na anotação da linha 6 na Figura 1.

3.2 Mapeando regiões em tarefas

Uma *tarefa candidata* é um conjunto de instruções que podem executar paralelamente. Para identificá-las, TaskMiner faz uso do *Grafo de Dependências*.

Definição 5 (Grafo de Dependências (PDG) [6])**.** *Dado um programa P, seu PDG contém um vértice para cada instrução*

$s \in P$. Existe uma aresta de s_1 a s_2 se o último depende do primeiro. s_2 tem uma dependência de dados de s_1 se lê dados que s_1 escreve. s_2 tem uma dependência de controle de s_1 se s_1 controla a execução de s_2 a partir do resultado de um branch, por exemplo.

Tarefas candidatas são **hélices em moinhos**. *Moinhos* são uma família de grafos [20]. Moinhos existem como subgrafos do grafo de dependências.

Definição 6 (Moinho)**.** *Um moinho é um grafo $G_w = G_c \cup G_{v1} \cup \ldots \cup G_{vn}$ formado por componentes fortemente conexos G_c (centro do Moinho) e n componentes, não necessariamente fortes, G_{vi} chamados de hélices, tais que:*

1. *para qualquer ordenação topológica de G_w, e os nós $n_c \in G_c$, $n_v \in G_{vi}$, n_c está à frente de n_v;*
2. *para qualquer i e j, $1 \leq i < j \leq n$, G_{vi} e G_{vj} não compartilham vértices (portanto, compartilhar arestas também é impossível).*

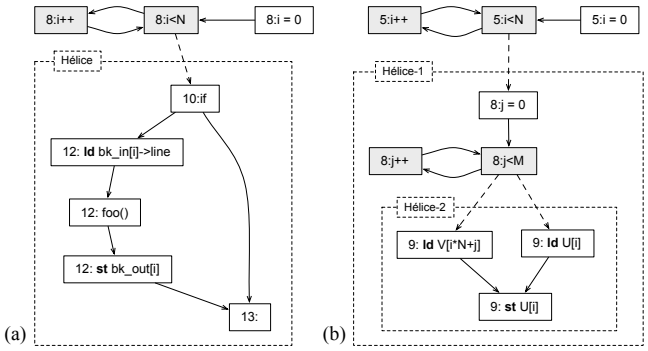

Figura 4. Exemplos de grafos de dependências. Nós que formam centros de moinhos são coloridos de cinza. Arestas sólidas representam dependências de dados; arestas pontilhadas dependências de controle. Hélices de moinhos aparecem dentro de retângulos pontilhados. Na imagem, vemos o grafo para a Figura 1 em (b).

A Figura 4 mostra dois grafos de dependências e seus moinhos. A estrutura de moinhos naturalmente evidencia tarefas candidatas. Hélices correspondem a partes do programa que tem alta probabilidade de serem executadas diversas vezes, pois surgem de um laço: o centro do moinho. Duas execuções de uma mesma hélice podem ser paralelas. Esta é uma propriedade do PDG e também uma das motivações originais para este trabalho. Essas duas execuções podem ser vistas como duas instâncias da mesma estrutura: a hélice, partindo do mesmo centro do moinho. Uma ordenação topológica desse grafo não impõe ordem entre as duas hipotéticas réplicas da mesma hélice.

Moinhos possuem propriedades estruturais para identificação de tarefas e são encontrados em uma simples busca em profundidade no PDG. Entretanto, nem todo moinho evidencia tarefas rentáveis. Ainda, alguns moinhos não podem ser anotados devido à inabilidade da análise de ponteiros em encontrar limites de acesso à memória. As próximas seções tratam desses problemas.

3.3 Estimando a lucratividade de tarefas

O benefício de tarefas depende de dois fatores: custo de execução e paralelismo. A dificuldade surge em encontrar o ponto mediano entre tarefas grandes e tarefas pequenas. Tarefas muito pequenas são mais paralelas, pois tendem a ter menos dependências. Entretanto, se uma tarefa é muito pequena, então os ganhos de performance são eclipsados pelo custo do ambiente de execução em manter essa tarefa ativa, e o paralelismo extra não compensa. Por outro lado, se tarefas são muito grandes, então não há paralelismo: tarefas individuais executarão sequencialmente. TaskMiner marca como tarefa as hélices que são maximais em termos de custo de execução, ou seja, as menores hélices que estão acima do custo de execução no ambiente escolhido.

O tamanho de uma tarefa é dado pelo seu número de instruções. A mesma região de um programa pode levar à criação de tarefas com tamanhos diferentes. Portanto, o tamanho real de uma tarefa só é conhecido ao final de sua execução. Para julgar a rentabilidade de uma tarefa, deve-se aproximar esse tamanho estaticamente. Assim, define-se a noção de *Trabalho Estatico Estimado*:

Definição 7 (Trabalho Estático Estimado (TEE))**.** *Seja $G = G_1 \cup G_2 \cup \ldots \cup G_n$ uma partição de um grafo de controle de um G em n regiões* hammock disjuntas. *Trabalho Estático Estimado ($W(G)$) é definido como um número real não negativo, tal que $W(G) = W(G_1) + W(G_2) + \ldots + W(G_n)$.*

O objetivo é obter um *TEE* próximo ao comportamento dinâmico dos programas. A heurística escolhida para este trabalho utiliza a análise de limites descrita na Seção 3.1 para inserir informações dinâmicas estaticamente através da construção de expressões simbólicas que representam o número de iterações de laços. Essas expressões são inseridas em condicionais para a criação da tarefa, como na linha 7 da Figura 1.

As regras declarativas da Figura 5 esboçam as heurísticas utilizadas para computar o TEE para uma região *hammock S*. A regra LoopInf aplica-se a laços cujo número de iterações não pode ser limitado por uma expressão simbólica computada estaticamente. Assume-se que laços executam ao menos 10 vezes [23]. A regra LoopExp representa laços que não podem ser analisados com a análise de limites simbólicos. A função auxiliar *Iter*(S) retorna uma expressão simbólica que representa a variedade de valores cobertos pelo código em S que controla o número de iterações do laço.

O Modelo de Custo. O TaskMiner recebe um modelo de custo como parâmetro. Um modelo de custo é uma coleção de parâmetros que determinam o impacto da criação e manutenção de tarefas em uma determinada arquitetura. A literatura lista técnicas analíticas e empíricas para a construção do

$$[\textsc{Instr}] \quad \text{W(instr) = TEE, como especificado.}$$

$$[\textsc{Seq}] \quad \frac{W(S_1) = w_1 \qquad W(S_2) = w_2}{W(S_1; S_2;) = w_1 + w_2}$$

$$[\textsc{Branch}] \quad \frac{W(S_1) = w_1 \qquad W(S_2) = w_2 \qquad W(S_3) = w_3}{W(\text{if}(S_1)\ S_2;\ \text{else}\ S_3;) = w_1 + \max(w_2, w_3)}$$

$$[\textsc{LoopInf}] \quad \frac{W(S_1) = w_1 \qquad W(S_2) = w_2 \qquad Iter(S_1) = \infty}{W(\text{while}(S_1)\ S_2;) = 10 \times (w_1 + w_2)}$$

$$[\textsc{LoopExp}] \quad \frac{W(S_1) = w_1 \qquad W(S_2) = w_2 \qquad Iter(S_1) = E}{W(\text{while}(S_1)\ S_2;) = E \times (w_1 + w_2)}$$

Figura 5. Estimando o custo das tarefas.

modelo [18]. Para os experimentos descritos na Seção 4, utilizou-se constantes relacionadas à criação de *threads* em uma determinada arquitetura. Por exemplo, a criação de tarefas foi estimada em 500 ciclos para a arquitetura em que os experimentos foram realizados.

3.4 Expansão de tarefas

Expansão de tarefas consiste em encontrar a menor tarefa que é grande o suficiente para compensar o custo de criação de *threads*. A Figura 6 mostra o algoritmo de expansão. Inicia-se o algoritmo atribuindo REGIÃO_PAI a um grafo *hammock* H que corresponde a uma hélice em um moinho W. A decomposição *hammock* de um programa estruturado é uma árvore [6], em que os vértices representam grafos *hammock*. H_1 é filho de H_2 na árvore se: (i) $H_1 \subset H_2$, e (ii) para qualquer nó H_x, se $H_1 \subset H_x$ então $H_2 \subset H_x$. Nese contexto, H_2 é *pai* de H_1.

```
1 def expand(HELICE, CUSTO):
2   PodeExpandir = True
3   while (PodeExpandir and TEE(HELICE) < CUSTO):
4     REGIAO_PAI = HELICE.getParent()
5     HELICE.basicblocks.add(REGIAOPARENTE)
6     HELICE.resolverDependencias()
7     PodeExpandir = VerificarExpansao(HELICE)
```

Figura 6. Descoberta de tarefas em regiões *hammock*. CUSTO é o custo de criação e escalonamento de *threads*.

A cada nova expansão, checa-se se a expansão é válida. A expansão é válida se: (i) HÉLICE está contida em um moinho W; (ii) as regiões de memória em HÉLICE podem ser analisadas de acordo com a seção **??**; (iii) HÉLICE não depende de uma outra hélice no mesmo moinho W.

O programa na Figura 7 (a) contém dois moinhos. O primeiro consiste do laço da linha 1. O outro, aninhado, é o laço

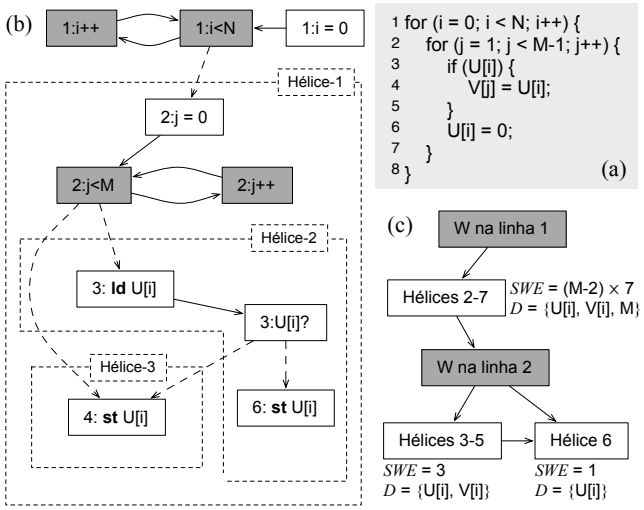

Figura 7. (a) Exemplo de laço duplamente aninhando analisado pelo TaskMiner. (b) O Grafo de Dependências do programa. (c) A decomposição desse laço em moinhos e hélices. D é o conjunto de variáveis das quais as hélices dependem.

da linha 2. As hélices que surgem desses moinhos estão evidenciadas na Figura 7 (b). Na Figura, há 3 tarefas candidatas, cada uma formada por uma hélice. Para encontrá-las, aplica-se a função de expansão (Fig. 6) nas hélices mais internas: Hélice-2 pode ser expandida para incluir Hélice-3.

A expansão ocorre até o limite definido pelo CUSTO, que é determinado pelo modelo de custo da Seção 3.3. Se a lucratividade é menor que este custo, então a criação da tarefa não é desejável. Se a tarefa é expandida demais, há perda de paralelismo: o código executa sequencialmente. Portanto, determinar o CUSTO é essencial para o desempenho do TaskMiner.

3.5 Anotando o código fonte

As análises deste artigo foram implementadas na representação intermediária (R.I.) do LLVM devido à facilidade do uso de análises como evolução escalar [7, p.18] e outras. Porém, as anotações são inseridas em código fonte C, então grande parte deste trabalho é o mapeamento de informações da R.I. do LLVM de volta para código C.

Árvore de escopos: A árvore de escopos é uma estrutura utilizada pelo TaskMiner para mapear regiões *hammock* em construtos C, como blocos while e if-then-else. Cada nó nesta árvore contém *metadados* provenientes de informação de depuração obtida durante a compilação para R.I. É possível achar o número da linha relacionada à cada nó na árvore de escopos, permitindo ao TaskMiner a anotação das regiões *hammock* no código fonte de maneira precisa.

Limitando a criação de tarefas recursivas Para evitar a criação excessiva de tarefas pequenas, TaskMiner dá a opção aos usuários de limitar a criação de tarefas. Por exemplo,

./Taskminer -r 12 limita o número de tarefas criadas a um total de 12 tarefas. Essa solução é implementada na fase de anotação. Essa solução traz claros benefícios em benchmarks recursivos, como mostra-se na Seção 4. A variável, chamada de taskminer_depht_cutoff na Figura 2, é global e ligada estaticamente. No começo de cada função recursiva esta variável é incrementada para indicar a criação de uma nova tarefa, e decrementada ao sair da função. A Figura 2 ilustra a limitação de tarefas durante invocação recursiva. O parâmetro PROFUNDIDADE_MAX é determinado pelo usuário.

Embora simples, a medida implementada é genérica o suficiente para lidar com tarefas difíceis de se limitar simbolicamente. Há outras formas de limitar o número de tarefas recursivas [9]. Futuramente, pretende-se explorar melhores maneiras de limitar criação de tarefas pequenas.

4 Avaliação Experimental

TaskMiner foi implementado em LLVM 3.9. Os experimentos foram executados em uma máquina de 12 núcleos 64-bits Intel(R) Xeon(R) CPU E5-2620, 2.00GHz, com 32K de cache L1, 256K de cache L2 e 15M de cache L3, e 15Gb de memória RAM. Utilizou-se OpenMP 4.5.

4.1 Desempenho

A Figura 8 compara o tempo de execução de programas produzidos pelo TaskMiner contra versões originais dos mesmos programas anotados manualmente. Os *benchmarks* utilizados neste experimento são do BSC-Bots [5] e do Swan [15]. Ambos vem com versões sequenciais e paralelas (anotadas manualmente). Os códigos foram compilados com gcc-6.

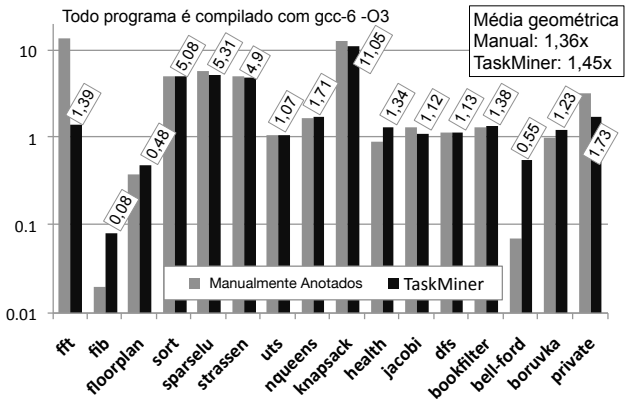

Figura 8. Comparações de performance. Anotações automáticas comparam-se favoralmente a intervenções manuais, e levam à ganhos de desempenho. No eixo y, ganho de desempenho de cada versão anotada pelo TaskMiner sobre a versão anotada manualmente. Os números nas caixas representam o crescimento em velocidade obtido.

As versões anotadas pelo TaskMiner foram mais velozes que as sequenciais em 13 dos 16 programas. A maioria são algoritmos Dividir & Conquistar clássicos como multiplicação de matrizes de Strassen, knapsack (Problema da Mochila) e fft (Transformada Rápida de Fourrier). Em 6 deles, as versões automáticas foram próximas e até levemente superiores às versões anotadas manualmente. Em 3, TaskMiner produziu versões piores que as versões sequenciais: fib, floorplan e bell-ford. Ainda assim, produziu versões mais rápidas que as anotadas manualmente. Uma inspeção de bell-ford indica que a versão paralelizada manualmente dispara um número grande tarefas pequenas. O modelo de custo do TaskMiner poda a criação de tarefas pequenas, mas não o suficiente para vencer o desempenho da versão sequencial. Este experimento demonstra que TaskMiner pode trazer ganhos consideráveis se comparados às versões sequenciais e manualmente anotadas.

4.2 Otimizações

Ambas as formas de otimizar a criação de tarefas descritas neste artigo são baseadas na ideia de *poda*: evita-se a criação de tarefas consideradas não-lucrativas. Limita-se a criação de tarefas ditas não lucrativas de acordo com o modelo de custo, conforme explicado na Seção 3.3; e limita-se a criação de tarefas muito profundas na pilha de recursão, como descrito na Seção 3.5. A Figura 9 ilustra os benefícios dessas duas técnicas, presentes em 6 dos benchmarks vistos anteriormente (sec 4.1).

Em 3 benchmarks, private, boruvka e bellman-ford, o modelo de custo evita a criação de tarefas em laços que inicializam estruturas de dados. Por exemplo, o laço a seguir, na linha 75 do programa bellmanford.c, seria paralelizado pelo TaskMiner, caso o modelo de custo o marcasse como lucrativo:

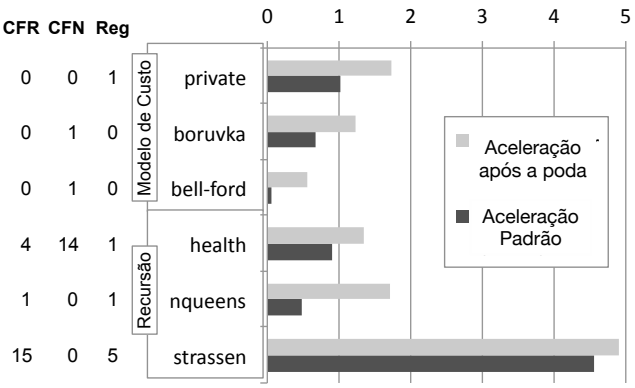

Figura 9. Benefícios de limitar a criação de tarefas. (Secs. 3.3 e 3.5). **CFR**: tarefas criadas em Chamadas de Funções Recursivas. **CNR**: tarefas interprocedurais criadas ao redor de Chamadas Não-Recursivas. **Reg**: tarefas envolvendo Regiões, sem chamadas de função.

```
for (long unsigned i = 0; i < N; i++)
  for (long unsigned j = 0; j < N; j++)
    *(G + i * N + j) = rand();
```

A limitação de tarefas recursivas (Sec. 3.5), embora simples, é efetiva em eliminar o excesso de tarefas pequenas. A Figura 9 mostra esse impacto em três programas: health, nqueens e strassen. Estes programas foram projetados para ilustrar paralelismo em paradigma Dividir & Conquistar, e contém muitas chamadas recursivas. A poda em níveis mais altos da árvore de recursão leva a bons ganhos de desempenho. Caso não houvesse essa otimização de poda, então perda de desempenho seria observada em health e nqueens, provando que as otimizações são eficazes e melhoram a qualidade do código gerado pelo TaskMiner.

4.3 Versatilidade

Os benchmarks das seções anteriores foram projetados especialmente para o estudo de paralelismo irregular. Para determinar se o TaskMiner consegue encontrar paralelismo em programas comuns, executou-se o mesmo sobre 219 programas C disponíveis na suíte de testes do LLVM, e comparou-se o desempenho da versão anotada automaticamente pelo TaskMiner com a versão original sequencial. TaskMiner anota um programa se: (i) consegue encontrar limites simbólicos para todos os acessos à memória que ocorrem em uma hélice; e (ii) a hélice é ampla o suficiente para pagar pelo custo de criação de tarefas no ambiente de execução. Sob esses termos, descobriu-se tarefas em 63 desses programas. A Figura 10 relaciona o número de tarefas ao número de instruções em 30 dos maiores programas utilizados. Como demonstrado neste experimento, TaskMiner é capaz de anotar um número não trivial de benchmarks reais.

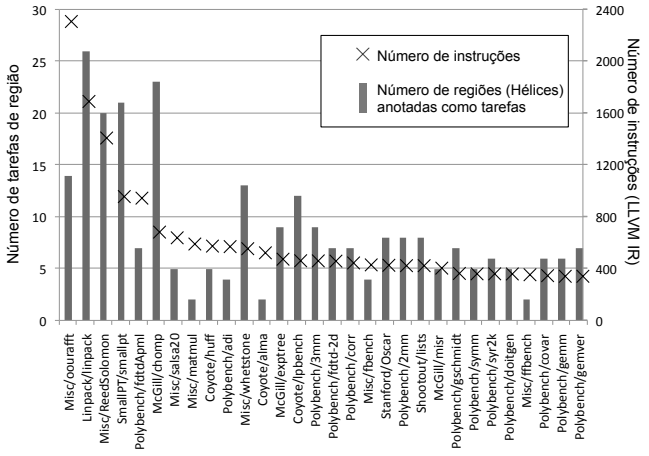

Figura 10. Relação entre o número de regiões de tarefas e tamanho do programa. Cada programa é um arquivo C completo.

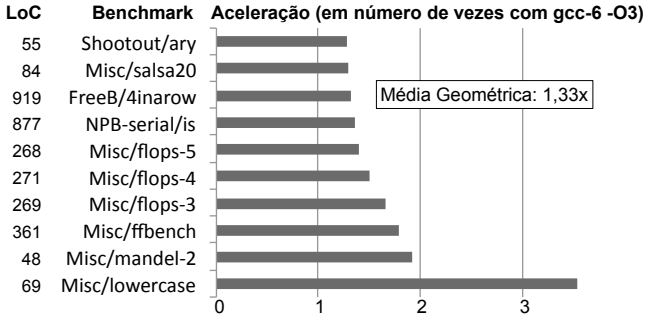

Figura 11. Melhorias de desempenho obtidas pelo TaskMiner quando aplicado à suíte de testes do LLVM. Quanto maior a barra, maior o ganho de desempenho. **LoC** significa "Linhas de Código".

Dos benchmarks anotados, 27 não demonstraram aumento de desempenho em relação à versão sequencial, pois as tarefas anotadas eram muito pequenas e pouco influenciaram na execução total do programa. Em 17 deles, observou-se perda de velocidade: neste caso, interações entre estruturas de dados forçaram o ambiente de execução do OpenMP a serializar a execução das tarefas. Essas perdas foram inferiores à 10%.

Obteve-se aumento de desempenho acima de 5% em 19 programas. Em Misc/lowercase, o fator de crescimento foi acima de 3,5x, e em 3 casos, acima de 1.5x. A Figura 11 mostra os 10 maiores acréscimos de desempenho obtidos pelo TaskMiner. Enfatiza-se que esse experimento *não contou com nenhum tipo de intervenção humana*. Portanto, TaskMiner é capaz de encontrar paralelismo automaticamente sem custo de programação.

5 Trabalhos Relacionados

Compiladores modernos adicionaram suporte ao paralelismo de tarefas do OpenMP apenas recentemente. Uma vez que a infra-estrutura necessária para a implementação de tarefas é nova, não existem ferramentas além do TaskMiner capazes de anotar programas automaticamente com diretivas de tarefas. Exemplos de modelos que permitem paralelismo de tarefas são OpenMP 4.X, StarSs [17] e OmpSs [3]. Por exemplo, Tareador [1] apresenta uma interface gráfica para identificação de potencial paralelismo e anotação de código sequencial.

A técnica utilizada pelo TaskMiner para identificação de tarefas se assemelha às análises usadas durante geração de código para máquinas de fluxo de dados. Muitas abordagens utilizam grafos de fluxo de dados para explorar paralelismo, como Tarefas Movidas à Dados [22], em que o programador pode manipular cláusulas para determinar os argumentos da tarefa antes de sua execução. Paralelismo baseado em função também foi proposto para utilizar argumentos de funções como especificadores de dependências entre tarefas [8].

Muitos sistemas foram desenvolvidos para extrair paralelismo de tarefas a partir de código sequencial. Entretanto, a mineração e inserção automática dessas diretivas não consta como objetivo em nenhum desses trabalhos, e TaskMiner segue como primeira ferramenta capaz de anotar automaticamente código com paralelismo irregular de maneira correta.

6 Conclusão

Este trabalho apresentou TaskMiner, uma ferramenta capaz de identificar automaticamente oportunidades de paralelismo de tarefas e anotá-las com diretivas OpenMP. TaskMiner implementa algoritmos de análise de código em representação intermediária do LLVM e contorna problemas clássicos no paralelismo irregular. TaskMiner prova-se eficiente e competitivo, sendo capaz de anotar programas não triviais e encontrar paralelismo escondido sem custo adicional.

Embora já possa ser usado em produção de código paralelo, TaskMiner ainda apresenta espaço para melhorias. Em particular, TaskMiner implementa uma heurística simples demais para estimar o custo de uma tarefa. Dada a sua simplicidade, essa heurística não é firme o suficiente para garantir a lucratividade de uma anotação. Como trabalho futuro, pretende-se implementar heurísticas de estimativa de custo que sejam mais precisas, para refinar o algoritmo do TaskMiner.

Referências

[1] Eduard Ayguadé, Rosa M. Badia, Daniel Jiménez, José R. Herrero, Jesús Labarta, Vladimir Subotic, and Gladys Utrera. 2015. Tareador: A Tool to Unveil Parallelization Strategies at Undergraduate Level. In *WCAE*. ACM, New York, NY, USA, 1:1–1:8.

[2] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *Trans. Parallel Distrib. Syst.* 20, 3 (2009), 404–418.

[3] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters* 21, 02 (2011), 173–193.

[4] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. An Adaptive Cut-off for Task Parallelism. In *SC*. IEEE, Piscataway, NJ, USA, 36:1–36:11.

[5] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP*. IEEE, Washington, DC, USA, 124–131.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[7] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012), 28 pages.

[8] Gagan Gupta and Gurindar S. Sohi. 2011. Dataflow Execution of Sequential Imperative Programs on Multicore Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 59–70.

[9] Shintaro Iwasaki and Kenjiro Taura. 2016. Autotuning of a Cut-Off for Task Parallel Programs. In *MCSoC*. IEEE, Piscataway, NJ, USA, 353–360.

[10] Julien Jaeger, Patrick Carribault, and Marc Pérache. 2015. Fine-grain Data Management Directory for OpenMP 4.0 and OpenACC. *Concurr. Comput. : Pract. Exper.* 27, 6 (2015), 1528–1539.

[11] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. ACM, New York, NY, USA, 194–206.

[12] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. 2011. A Runtime Implementation of OpenMP Tasks. In *IWOMP*. Springer, Heidelberg, Germany, 165–178.

[13] Gleison Souza Diniz Mendonça, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Fernando Magno Quintão Pereira, Márcio Machado Pereira, and Guido Araújo. 2016. Automatic Insertion of Copy Annotation in Data-Parallel Programs. In *SBAC-PAD*. IEEE, New York, 34–41.

[14] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Trans. Archit. Code Optim.* 14, 2 (2017), 13:1–13:25.

[15] Rubens E.A. Moreira, Sylvain Collange, and Fernando Magno Quintão Pereira. 2017. Function Call Re-Vectorization. In *PPoPP*. ACM, New York, NY, USA, 313–326.

[16] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *PLDI*. ACM, New York, NY, USA, 12–25.

[17] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.* 23, 3 (2009), 284–299.

[18] Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 50:1–50:28.

[19] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[20] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. 2008. Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves. *J. Autom. Reason.* 40, 4 (2008), 307–326.

[21] OpenACC Standard. 2013. *The OpenACC Programming Interface*. Technical Report. CAPs.

[22] S. Tasirlar and V. Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *International Conference on Parallel Processing*. IEEE Computer Society, Taipei City, Taiwan, 652–661.

[23] Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *MICRO*. ACM, New York, NY, USA, 1–11.