ENCONTRANDO REDUÇÕES ARBITRARIAMENTE COMPLEXAS EM PROGRAMAS C AUTOMATICAMENTE

JOÃO LADEIRA REZENDE[1]; EDEVALDO BRAGA DOS SANTOS[2]; GERSON GERALDO H. CAVALHEIRO[3]

[1]*Universidade Federal de Pelotas – jplrezende@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – edevaldo.santos@inf.ufpel.edu.br*
[3]*Universidade Federal de Pelotas – gerson.cavalheiro@inf.ufpel.edu.br*

1. INTRODUÇÃO

Computadores têm se tornado crescentemente capazes de executar computações paralelamente, em múltiplos núcleos de execução. No entanto, para que essa capacidade seja devidamente aproveitada, programas precisam ser escritos levando-a em consideração: o programador geralmente deve indicar explicitamente quais computações podem ser quebradas em múltiplas partes a serem executadas paralelamente. Em programas C, essa indicação pode ser feita em uma linguagem de anotação como OpenMP (OPENMP, 2020). Assim, programas que foram escritos antes da emergência de computadores paralelos precisam ser adaptados e, ocasionalmente, parcialmente reescritos.

Para ajudar nessa adaptação, foram desenvolvidos compiladores paralelizadores: tradutores capazes de realizá-la automaticamente inserindo anotações OpenMP. Alguns exemplos desses compiladores são Cetus (LEE, 2003), Rose (QUINLAN, 2011), Par4All (AMINI, 2012) e ICC (XINMIN, 2002). Uma das tarefas principais dessas ferramentas é identificar trechos de um programa que podem ser paralelizados. Diversas técnicas são empregadas para isso. Em princípio, essas ferramentas operam sem assistência do programador e, por isso, focam em encontrar trechos de código que podem ser executados paralelamente sem grandes modificações do programa. Uma desvantagem dessa abordagem é que ela deixa passarem despercebidos outros trechos paralelizáveis.

```
int reduce(int vet[], size_t n) {
   int acc = 0;
   for (size_t i = 0; i < n; ++i)
       acc = acc + vet[i];
   return acc;
}
```

Figura 1. Exemplo de redução na linguagem C

Um tipo de computação que frequentemente pode ser quebrada em múltiplas partes a serem executadas paralelamente é o de *redução*. Uma redução é uma computação de um valor simples a partir de uma coleção de valores. Um exemplo é o laço mostrado na Figura 1, que calcula a soma de todos os números de uma sequência numérica. Compiladores paralelizadores detectam e paralelizam automaticamente reduções em programas. Eles detectam somente reduções que podem ser prontamente e trivialmente paralelizadas por meio de uma anotação OpenMP. No entanto, reduções ocasionalmente não pertencem a essa categoria, mesmo quando são passíveis de paralelização. Este trabalho propõe uma abordagem de detecção estática de reduções arbitrárias em

programas C, incluindo essas cuja paralelização exigiria uma reescrita significativa do programa.

Para garantir que paralelizem somente laços de redução que são mesmo seguramente paralelizáveis, essas ferramentas identificam um laço como redução somente se ele satisfaz algumas condições. Primeiramente, elas verificam se a variável acumuladora é referenciada em quaisquer comandos do laço além do comando que acumula um valor nela (`acc = acc + vet[i];` no exemplo da Figura 1). Se for, então o laço não é identificado como um laço de redução. Isso porque, neste caso, executar o laço paralelamente como uma redução pode alterar seu efeito.

Outra limitação dos compiladores paralelizadores é que eles não identificam reduções cuja variável acumuladora é um campo de uma estrutura, como em `e.acc+=a[i]`, ou uma variável alcançada por meio de um ponteiro, como em `*acc+=a[i]`. Também, essas ferramentas reconhecem somente reduções realizadas em laços `for`, e não em laços `while` ou `do ... while`. Ainda outra limitação notável é que elas identificam um laço de redução somente se ele contém uma expressão de atribuição da forma `acc=acc+expr` ou, equivalentemente, `acc+=expr`, onde `expr` é uma expressão arbitrária. Isso é: um laço de redução é identificado somente se sua variável acumuladora aparece em ambos os lados de uma mesma expressão de atribuição. Passa despercebido qualquer laço de redução que capture o valor corrente da variável acumuladora e a atualize em comandos diferentes.

Este trabalho objetiva propor um algoritmo de detecção estática de reduções que alivie essas restrições impostas sobre laços de redução. O texto está organizado como segue. Na Seção 2, é descrito o algoritmo propriamente dito, sua implementação e a metodologia de validação empregada. A Seção 3 discute os resultados da validação. A seção 4 então conclui o texto.

2. METODOLOGIA

Características recorrentes de laços de redução foram consideradas na elaboração de um algoritmo heurístico de identificação de reduções. Laços de redução são identificados por meio de reconhecimento de características frequentemente apresentadas por eles. Mais especificamente, é analisada cada variável que é escrita no corpo de um laço. A cada tal variável é associada uma pontuação numérica que representa a probabilidade de ela ser uma variável acumuladora de uma computação de redução. A presença de múltiplas características é observada para determinar essa pontuação. Essas características são descritas a seguir.

Primeiramente é verificado se as expressões de atribuição que escrevem a variável no corpo do laço também a referenciam no seu lado direito. Em caso positivo, a pontuação da variável é aumentada. É notável que, em caso negativo, a variável não é imediatamente descartada como candidata a acumuladora, como acontece nas ferramentas existentes. A segunda etapa de análise verifica se a variável é referenciada em comandos do laço que não são as atribuições que aparentam acumular um valor nela. Para cada uma dessas referências, a pontuação da variável é diminuída levemente — sem descartá-la imediatamente.

Para compensar, pela sua tolerância a evidências negativas da ocorrência de redução, o algoritmo também procura outras características indicativas de redução que não são verificadas pelas ferramentas existentes. Essas

características, verificadas nas análises descritas a seguir, são causadas por convenções e estilos de programação, e não são inerentes a reduções.

A terceira análise observa a distância entre a declaração da variável e o laço no qual ela possivelmente acumula um valor. Se ela é declarada na linha de código imediatamente anterior ao laço, como na Figura 1, sua pontuação é aumentada. A quarta e última análise então verifica se o nome da variável tem como subpalavra “acc”, “total” ou “sum”. Em caso positivo, sua pontuação é aumentada.

O algoritmo foi implementado[1] para a linguagem C, reaproveitando componentes do compilador Clang. Essa implementação foi validada por meio de uma comparação dos seus resultados com os resultados emitidos pelo compilador paralelizador Cetus quando executado sobre o código fonte da suíte de benchmarks de computação paralela BOTS (*Barcelona OpenMP Tasks Suite*) (DURAN, 2009), que contém 167 laços no total.

3. RESULTADOS E DISCUSSÃO

O Cetus identificou corretamente 13 laços de redução no código analisado. A implementação do algoritmo aqui proposto indicou 11 desses mesmos laços, mais 44 outros. Apenas 9 desses outros laços realmente acumulam um valor em uma variável ao longo das suas iterações — os outros 35 são falsos positivos.

A perda de 2 das reduções detectadas pelo Cetus ocorre por causa de uma lacuna na implementação que será preenchida no futuro: ela não é capaz de detectar acumulações que são feitas por meio do operador unário de incremento ++ em vez de por meio de uma atribuição explícita.

Os casos detectados pela ferramenta proposta mas não pelo Cetus incluem reduções que reduzem valores contidos em listas encadeadas, uma redução feita por meio de uma atribuição que não é da forma `acc=acc+expr`, e reduções feitas em variáveis acumuladoras externas alcançadas por meio de ponteiros. De fato, essas são reduções que não podem ser paralelizadas simplesmente pela adição de anotações OpenMP. Paralelização delas exigiria no mínimo uma reescrita pequena do código.

Os 35 falsos positivos emitidos pela ferramenta ocorreram por motivos diversos. Deles, 17 foram causados por uma ocorrência frequente no código fonte do BOTS que não foi prevista durante a elaboração do algoritmo: frequentemente são declaradas fora de laços variáveis que são usadas somente no corpo de um laço e depois desprezadas. Essas variáveis acabam sendo consideradas como possíveis acumuladoras pelo algoritmo, apesar de ser impossível uma variável servir como acumuladora de forma útil se ela nunca é lida depois do laço no qual ela é escrita. Esses 17 falsos positivos podem ser evitados descartando variáveis candidatas a acumuladoras que não são lidas depois do aparente laço redutor.

Dos falsos positivos restantes, 4 são laços que na verdade simplesmente percorrem uma lista encadeada. Esses laços são confundidos com reduções porque apresentam múltiplas das mesmas características de laços de redução. Por exemplo, um desses falsos positivos contém o comando `aux=aux->forward;`, o qual atribui à variável `aux` um valor em função do valor anterior dela. Falsos positivos desse tipo podem ser evitados descartando candidatas a variáveis acumuladoras que são ponteiros.

[1] https://github.com/JoaoLRezende/reduction-detector

Três dos falsos positivos restantes são causados por uma consequência não prevista da capacidade da ferramenta de reconhecer reduções feitas em campos de estruturas alcançadas por meio de um ponteiro. Por exemplo, um dos falsos positivos é causado pelo fato de que a ferramenta reconhece que uma variável `p->time` é aumentada em cada iteração de um laço, mas sem perceber que o ponteiro `p` na verdade aponta para uma estrutura diferente em cada iteração. Esses falsos positivos podem ser contornados verificando se ponteiros usados com o operador -> são escritos no corpo dos laços analisados.

4. CONCLUSÕES

Este trabalho relatou o desenvolvimento de uma estratégia heurística de detecção estática de computações de redução em programas C que, diferentemente das ferramentas atualmente disponíveis, objetiva detectar reduções cuja paralelização exigiria intervenção do programador. Os testes de validação iniciais mostraram resultados satisfatórios. Apesar desses resultados incluírem múltiplos falsos positivos, muitos deles podem ser contornados por meio de modificações ligeiras à ferramenta implementada. Essas modificações serão feitas na sequência da pesquisa.

5. REFERÊNCIAS BIBLIOGRÁFICAS

AMINI, M. et al. Par4all: from convex array regions to heterogeneous computing. In: **2ND INTERNATIONAL WORKSHOP ON POLYHEDRAL COMPILATION TECHNIQUES, IMPACT**. Citeseer, 2012.

DURAN, A. et al. Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: **2009 INTER. CONF. ON PARALLEL PROCESSING**. IEEE, 2009, p.124-131.

LEE, S.; JOHNSON, T.; EIGENMANN, R. Cetus: an extensible compiler infrastructure for source-to-source transformation. In: **INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING**. 2003, Springer. p.539.

OPENMP. **OpenMP Application Programming Interface 5.1**. OpenMP Architecture Review Board, nov. 2020. Acessado em 7 ago. 2021. Online. Disponível em https://www.openmp.org/specifications/.

QUINLAN, D.; LIAO, C. The ROSE source-to-source compiler infrastructure. In: **CETUS USERS AND COMPILER INFRASTRUCTURE WORKSHOP, IN CONJUNCTION WITH PACT**. 2011, ACM.

XINMIN, T. et al. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. **Intel Technology Journal**, v.6, n.1, p.1, 2002.