Detecção de operações de redução em programas C*

João Ladeira Rezende[1], Edevaldo Braga dos Santos[1], Gerson Geraldo H. Cavalheiro[1]

[1]Centro de Desenvolvimento Tecnológico – Universidade Federal de Pelotas (UFPel)

{jplrezende,edevaldo.santos,gerson.cavalheiro}@inf.ufpel.edu.br

Abstract. *The automatic parallelization capabilities available on current compilers are limited to detecting code fragments that can be automatically parallelized and not to reporting fragments that are candidates for parallelization by manual reprogramming. In this paper, we propose a method for detecting program fragments that reflect the reduction iterative pattern that might be considered to be paralelized by a manual code refactoration. The method proposed was validated by comparing the results obtained with those required by Cetus in the BOTS benchmark. The analysis of the results indicated the validity of our approach, and the cases of false positives reported in the application of the proposed technique are extensively discussed.*

Resumo. *Os atuais compiladores dotados de recursos para paralelização automática de código limitam-se a detectar trechos de código paralelizáveis automaticamente, não reportando trechos de código candidatos a paralelização por reprogramação manual. Este artigo apresenta uma estratégia para detecção de trechos de código refletindo o padrão iterativo de redução candidatos a paralelização por transformação manual. A validação foi realizada comparando os resultados obtidos com os apresentados por Cetus sobre o benchmark BOTS. A análise dos resultados indicou a validade da proposta, sendo discutidos os casos de falsos positivos reportados na aplicação da técnica proposta.*

1. Introdução

Em código legado, a tarefa de identificar computações que podem ser executadas paralelamente pode ser custosa. Múltiplas ferramentas já foram desenvolvidas com o objetivo de auxiliar nas tarefas de identificar computações paralelizáveis e paralelizá-las estaticamente. Um padrão algorítmico que ocasionalmente é paralelizado é o de redução. Este trabalho descreve uma estratégia de detecção programática de trechos de código iterativos sequenciais que executam reduções. É descrita também uma implementação dessa estratégia para a linguagem C.

Reduções são descritas em algoritmos iterativos por laços [Jradi et al. 2018, Parhami 2002] conforme o modelo apresentado na Figura 1.(a) e exemplificado como um código C na Figura 1.(b). No modelo, um espaço de dados vet de n posições é percorrido, sendo o valor de cada posição i reduzido, de forma acumulativa, sobre a variável acc pela operação $\oplus$. No exemplo de implementação, observa-se que a variável acumuladora acc é declarada em um escopo envolvente ao laço e que, no corpo deste laço, acc ocorre tanto no lado esquerdo como no direito de uma expressão de atribuição. O operador $\oplus$,

Input: $vet[n]$
Output: acc
$acc \leftarrow acc \oplus vet[i], \forall i \in [1; n]$

(a) Forma geral de reduções

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

(b) Exemplo concreto de redução em C

Figura 1. Representações de redução

neste exemplo, é representado pelo operador $+$ (soma).

Em uma implementação paralela de um laço de redução, onde o espaço de iteração é atribuído a tarefas concorrentes, deve-se garantir que a operação $\oplus$ seja associativa e prever alguma forma de compartilhamento de acesso à variável acc. A identificação de um laço de redução candidato a paralelização requer, portanto, determinar tanto a possibilidade de realizar o compartilhamento da variável de acumulação entre as tarefas como a inexistência de efeitos colaterais no uso da variável acumuladora.

Geralmente, ferramentas estáticas de paralelização operam exigindo nenhuma ou mínima intervenção manual do programador [Harel et al. 2020]. Por isso, elas agem de forma pessimista: quando não podem garantir que uma computação pode ser seguramente executada paralelamente, deixam de paralelizá-la. Essa independência também significa que elas tomam uma atitude conservadora, que detecta somente trechos de código que podem ser paralelizados de forma direta, por meio de transformações padrões que não exigem grandes refatorações do programa — por exemplo, por meio da inserção de uma diretiva OpenMP `for` com a cláusula `reduction`.

A estratégia proposta representa uma abordagem otimista. Um laço iterativo é classificado como contendo uma operação de redução mesmo quando a paralelização não pode ser realizada de forma automática, como no caso da operação de redução ser uma de um conjunto de instruções executadas no corpo da iteração. Isso é feito por meio de um algoritmo heurístico de reconhecimento de padrões aplicado de forma estática. Laços de redução são identificados por meio de reconhecimento de características frequentemente apresentadas por eles. As tarefas de confirmar que o trecho apresenta uma redução paralelizável, isolá-la e paralelizá-la ficam como responsabilidade do programador. Essa abordagem objetiva evitar perda de oportunidades de paralelismo que passariam despercebidas nas ferramentas existentes.

A estratégia foi implementada para a linguagem C reaproveitando componentes do compilador Clang. Essa implementação foi testada executando-a sobre o código fonte do BOTS (*Barcelona OpenMP Tasks Suite*), uma suíte de benchmarks de computação paralela. Os resultados mostraram que a estratégia é capaz de detectar reduções que não são encontradas por ferramentas existentes, mas com diversos falsos positivos.

O restante deste trabalho está organizado como segue. A Seção 2 descreve ferra-

*Este trabalho foi parcialmente financiado pelo projeto "GREEN-CLOUD: Computação em Cloud com Computação Sustentável" (#16/2551-0000 488-9 – PRONEX FAPERGS/CNPq) e pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de financiamento 001.

mentas que realizam detecção de reduções em código C estaticamente. As seções 3 e 4 apresentam, respectivamente, o algoritmo proposto de detecção de reduções e sua implementação. A Seção 5 apresenta os resultados do estudo de caso sobre o código fonte do BOTS. As conclusões são apresentadas na Seção 6.

2. Trabalhos relacionados

Rose [Quinlan and Liao 2011], Par4All [Amini et al. 2012] e Cetus [Lee et al. 2003] são exemplos de ferramentas capazes de detectar reduções em C estaticamente. Estas ferramentas dispõem de técnicas para detecção de paralelismos e automatizam a paralelização introduzindo, no programa fonte original, diretivas OpenMP. Estas ferramentas reconhecem, de forma genérica, um laço `for` como um laço de redução se seu corpo contém uma expressão de atribuição da forma `acc=acc+expr`, ou mesmo `acc+=expr`, onde `acc` é uma variável escalar declarada fora do laço, não referenciada em qualquer outro comando do laço, e `expr` é uma expressão arbitrariamente complexa. Este mecanismo de reconhecimento é documentado em Cetus [Bae et al. 2013], no entanto, pode ser identificado nos códigos das outras ferramentas disponibilizados nos repositórios oficiais[1,2]. A Intel também mantém um compilador paralelizador de C e C++, chamado *Intel C Compiler* ou *ICC* [Tian et al. 2005], que reconhece e paraleliza laços de redução usando um algoritmo análogo às demais ferramentas citadas. Na sequência, as limitações identificadas destas ferramentas são caracterizadas, sendo considerado que a principal razão destas limitações é o fato de que estas se propõem a paralelizar o código de forma automática.

Em comum, essas ferramentas limitam-se a identificar laços de redução apenas quando a variável acumuladora não é referenciada em qualquer outra parte do corpo do laço além da expressão de acumulação. Não sendo o caso, executar o laço paralelamente pode alterar seu efeito, não sendo passível de paralelização automática com as técnicas por elas empregadas. A inexistência de outras referências à variável candidata a variável acumuladora no laço não é verificada apenas para o propósito de garantir que o laço de redução não terá seu efeito alterado ao ser executado paralelamente; ela é verificada também porque ela própria é um dos principais indicadores da ocorrência de uma redução.

Outra situação é que essas ferramentas apenas identificam variáveis acumuladoras quando essas ocorrem em ambos lados esquerdo e direito de uma mesma expressão de atribuição. Em uma efetiva redução, essa limitação não é de fato um impeditivo, desde que seja observado que a variável acumuladora primeiro ocorra no lado direito de uma atribuição para então, após uma sequência de atribuições, ocorrer no lado esquerdo. A identificação de um laço candidato a incluir redução depende, assim, da análise das dependências entre as instruções.

As ferramentas citadas falham, também, em não detectar situações onde a variável acumuladora é representada em campos de estruturas, como em `e.acc+=a[i]`, pois estas não são paralelizáveis em OpenMP[3]. Elas também não reconhecem variáveis acumuladoras acessadas por uma desreferência a um ponteiro, como `*acc+=a[i]`.

Por fim, outra limitação notável é que as abordagens das ferramentas estudadas limitam-se a detectar reduções em laços `for`, desconsiderando outros tipos de laços ite-

[1]https://github.com/rose-compiler/rose

[2]https://github.com/Par4All/par4all

[3]https://www.openmp.org/spec-html/5.0/openmpsu107.html

rativos [Prema and Jehadeesan 2013]. Isso pode ser explicado pois laços `for` são os únicos que podem ser anotados como uma redução em OpenMP. Embora, na linguagem C, o comando `for` não possua como entrada a lista de valores a serem assumidos pela variável de controle de iteração, sua sintaxe permite que o espaço de iteração seja inferido, caso o comando na sua forma padrão. Nos demais comandos de iteração a estrutura da linguagem não indica como serão evoluídos os valores da variável de iteração, tornando tanto a análise como a paralelização automática atividades muito complexas.

Foi possível confirmar experimentalmente o potencial e as limitações do Cetus e do ICC. Nas demais, foi possível apenas analisar os códigos disponíveis nos repositórios, uma vez que não foi possível operacionalizá-las. Na sequência deste texto, o problema de detectar reduções não identificadas nas ferramentas estudadas é objeto de estudo. É apresentada e discutida uma estratégia estática de detecção de reduções para identificação de laços contendo padrões de redução que sejam candidatos à paralelização, mesmo que necessitando intervenção do programador para obtenção desta paralelização.

3. Algoritmo de identificação de reduções em C

Características recorrentes de laços de redução foram consideradas na elaboração de um algoritmo heurístico para identificação de laços de redução, Figura 2. São considerados quaisquer laços, não apenas os definidos em comandos `for`. A presença de diferentes unidades sintáticas são exploradas para estimar a probabilidade de cada laço conter uma redução. No algoritmo, para cada *laço* identificado, *variáveis_externas_modificadas(laço)*, é o conjunto das variáveis declaradas fora do laço *laço*, com escopo válido em *laço*, e que são lado esquerdo de pelo menos uma expressão de atribuição. A cada variável *possível_acumulador* é associado um valor *possível_acumulador.pontuação*, que representa a probabilidade desta ser um acumulador. O operador $\overset{+}{\leftarrow}$ acrescenta um determinado valor à pontuação da variável, enquanto $\overset{-}{\leftarrow}$ diminui.

Cada variável que é escrita no corpo de um laço é investigada com o objetivo de determinar a probabilidade dessa ser um acumulador em uma operação de redução. Essa probabilidade é representada por uma pontuação numérica. O algoritmo inicializa a pontuação de cada candidata a acumulador com um valor inicial na linha 4, e então executa uma série de análises à procura de evidências de que ela seja ou não seja uma variável acumuladora, enquanto atualiza a pontuação de acordo. Os valores assumidos para determinar as probabilidades foram definidos empiricamente, carecendo de validação e calibração futuras. São feitas quatro análises.

A primeira análise, na linha 5, verifica se, no laço, as expressões de atribuição que escrevem essa variável também têm alguma referência a ela nos seus lados direitos (como em `acc=acc + vet[i]`). Caso positivo, a pontuação é aumentada. Isso é porque, se o valor atribuído à variável é obtido em função de seu valor anterior, é mais provável que esteja ocorrendo uma acumulação.

A segunda análise, na linha 8, verifica o número de vezes em que essa variável é referenciada em comandos do laço que não a escrevem. Para cada uma dessas referências, a pontuação é diminuída levemente. Diferentemente das ferramentas existentes, o algoritmo não descarta imediatamente todo laço que consulte sua variável candidata a acumuladora ao longo das suas iterações. Isso permite que um laço de redução se distancie da forma fixa de laços de redução que é reconhecida pelas ferramentas existentes.

Entrada: conjunto *laços* dos laços de um programa C
Saída: conjunto *reduções* de prováveis laços de redução

```
1  reduções ← ∅
2  para cada laço ∈ laços faça
3      para cada possível_acumulador ∈ variáveis_externas_modificadas(laço) faça
4          possível_acumulador.pontuação ← 0.5
5          se possível_acumulador também aparece no lado direito das atribuições que o
             modificam então
6              possível_acumulador.pontuação +← 0.3
7          fim
8          para cada referência a possível_acumulador que ocorre no corpo do laço mas
             fora das atribuições que o modificam faça
9              possível_acumulador.pontuação −← 0.1
10         fim
11         se possível_acumulador é declarado imediatamente antes do laço então
12             possível_acumulador.pontuação +← 0.2
13         fim
14         se o nome de possível_acumulador tem como subpalavra "acc", "total", ou
             "sum" então
15             possível_acumulador.pontuação +← 0.5
16         fim
17         se possível_acumulador.pontuação ≥ 0.7 então
18             reduções ← reduções ∪ { laço }
19         fim
20     fim
21 fim
```

Figura 2. Algoritmo heurístico de detecção de reduções

No entanto, na medida em que um laço se distancia dessa forma fixa, se torna menos provável que ele seja um laço de redução. Para contrabalançar esse indício negativo, o algoritmo procura também a presença de outras características indicativas da ocorrência de uma redução. Essas características, verificadas pelas etapas de análise descritas a seguir, são causadas por convenções e estilos de programação, e não são inerentes a reduções.

A terceira análise, na linha 11, 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.(b), sua pontuação é aumentada. Essa é uma característica ocasional, embora não necessária, de reduções. É uma prática frequente declarar cada variável perto do ponto em que ela é usada pela primeira vez[4]. Essa prática não é predominante, e nem é sempre possível. Mesmo assim, ela significa que, quando uma variável é declarada imediatamente antes de um laço, é maior a probabilidade de o propósito desse laço ser popular essa variável. A quarta e última análise então verifica, na linha 14, se o nome da variável tem como subpalavra "acc", "total" ou "sum".

Finalmente, na linha 17, verifica-se se a pontuação final é maior que um limiar mínimo. Se for, a variável é considerada um provável acumulador.

[4] `https://wiki.c2.com/?DeclareVariablesAtFirstUse`

4. Implementação do algoritmo

O algoritmo foi implementado na forma de uma aplicação do Clang e encontra-se disponível para colaborações[5]. Na implementação, o programa fonte é percorrido, identificando a ocorrência de laços iterativos, sejam eles `for`, `while` ou `do ... while`. Para cada laço identificado é associada uma lista de descritores de variáveis *Candidata a Variável Acumuladora* (CVA). Cada descritor, além da identificação da CVA, contém campos a serem preenchidos durante o processo de análise para, na conclusão do processo, quantificar sua pontuação como CVA. Cada procedimento que executa uma etapa da análise sobre todas as CVAs de um laço é denominado uma *varredura de análise*.

A primeira varredura de análise identifica CVAs existentes. Todas variáveis declaradas fora do laço e utilizadas no lado esquerdo de uma expressão de atribuição são consideradas CVAs. A segunda varredura de análise determina a distância, em termos de linhas de código, entre a declaração de cada CVA e o laço.

Então é executada uma varredura de análise para identificar se cada CVA do laço participa em ambos lados esquerdo e direito em expressões de atribuição. Para tal, é verificado se a subárvore sintática que representa o lado direito da expressão de atribuição contém alguma subárvore igual àquela que representa seu lado esquerdo, que é a própria CVA. Esta varredura também considera operadores de atribuição aritmética, como +=.

A próxima varredura de análise é a responsável por contar o número de vezes em que cada CVA é referenciada dentro do laço mas fora das atribuições que a modificam. São procuradas, na subárvore sintática que descreve o corpo do laço, outras subárvores sintáticas iguais às subárvores sintáticas que representam as CVAs. A quinta, e final, varredura de análise verifica se o identificador empregado para nomear as CVAs contém termos pertence ao conjunto { “acc”, “total”, “sum” }, como `totalValue` ou `sumCosts`.

O processo conclui determinando a pontuação de cada CVA pela acumulação dos valores obtidos nas varreduras de análise. O grau de sensibilidade é determinado por um parâmetro no lançamento da ferramenta (opção `--min-score`). Um laço que contém pelo menos uma CVA com pontuação igual ou maior que esse limiar mínimo é considerado uma provável redução. Como saída, são apresentadas as prováveis reduções detectadas. Se foi passada a opção `--verbose`, cada provável redução é seguida por um relatório das informações adquiridas sobre suas CVAs.

5. Validação da implementação

Para mensurar a contribuição do algoritmo proposto, sua implementação foi executada sobre o código fonte da suíte de benchmarks de computação paralela BOTS (*Barcelona OpenMP Tasks Suite*) [Duran et al. 2009]. Esta suíte foi concebida para facilitar a avaliação de implementações de OpenMP, providenciando um conjunto de 9 programas paralelos. O código fonte desse conjunto de benchmarks consiste em 6.082 linhas de código C, excluindo comentários e linhas vazias. O total de comandos de iteração no conjunto de programas do BOTS é 167, dos quais, 139 utilizando o comando `for` e 28 com estruturas `while`. Destes últimos, apenas 1 caso na forma `do ... while`.

O resultado obtido sobre o BOTS foi comparado ao apresentado, para o mesmo código, por Cetus, na sua versão 1.4.4, aplicando os argumentos

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

`-parallelize-loops=4` e `-reduction=2`. Como resultado, ele aponta a existência de 13 reduções no conjunto de programas analisado. Em uma inspeção manual, concluiu-se que todas elas são reduções verdadeiras.

Além das reduções encontradas pelo Cetus, a implementação do algoritmo aqui proposto aponta outros 44 possíveis casos de redução passível de paralelização. Apenas 9 desses laços realmente acumulam um valor em uma variável ao longo das suas iterações. A Tabela 1 resume os resultados deste experimento, quantificando os laços indicados pelas ferramentas e quantas dessas indicações foram errôneas.

Tabela 1. Caracterização dos laços indicados pelas ferramentas

	Cetus	Proposta		NIP
		IC	NIC	
Reduções verdadeiras	13	11	9	2
Falsos positivos	0	35		-
Total	13	55		2

IC: Identificado por Cetus NIC: Não Identificado por Cetus
NIP: Não Identificado pela Proposta

Nas subseções seguintes são discutidas as três situações observadas: casos em que Cetus identificou redução e a nossa ferramenta não, casos em que a nossa ferramenta identificou redução e Cetus não e, por fim, casos em que a nossa ferramenta identificou códigos candidatos que, na verdade, não apresentam reduções (falsos positivos).

5.1. Casos não identificados

Como primeiro passo da avaliação da corretude da implementação proposta, foi verificado se ela detecta como prováveis reduções todas as reduções identificadas pelo Cetus. A implementação realizada, diferentemente do Cetus, não identificou duas reduções nas quais o operador unário de incremento ++ foi utilizado. No Cetus, essas expressões são reconhecidas como equivalentes a expressões de atribuição da forma `acc=acc+1`. De fato, estas expressões são equivalentes. A implementação do algoritmo aqui proposto não implementa detecção de acumulações feitas dessa forma.

Está sendo avaliado o interesse de incorporar esta regra. Um aspecto a ser considerado é o baixo custo computacional da operação em relação ao custo de sua execução paralela. Como no presente trabalho as questões de pesquisa se ativeram a paralelização de tarefas de maior grau de granularidade, este caso não foi considerado. No entanto, sua inclusão como uma nova regra é totalmente viável e deverá ser avaliada.

5.2. Reduções verdadeiras

A estratégia proposta apresentou 9 sugestões de paralelização de laços não apontadas por Cetus que, após análise, se mostraram como viáveis. Um exemplo é mostrado na Figura 3. Este laço percorre uma lista encadeada, acumulando somas de dados obtidos dela em 9 campos de uma estrutura `t_res`. A ferramenta aponta como acumuladores todos esses campos. Outros 4 casos, dos 9 em que foi observada efetivamente a ocorrência de redução, também percorrem listas encadeadas.

```
while (vlist)
    {
        p_res = get_results(vlist);
        t_res.hosps_number     += p_res.hosps_number;
        t_res.hosps_personnel += p_res.hosps_personnel;
        t_res.total_patients  += p_res.total_patients;
        [...]
        vlist = vlist->next;
    }
```

Figura 3. Laço reduzindo valores de uma lista encadeada (programa *Health*)

Outro exemplo de redução identificada pela ferramenta proposta é mostrado na Figura 4. O BOTS inclui um benchmark *sort*, que implementa um algoritmo de ordenação de arrays. Para propósito de testes, esse benchmark define também uma função `sort_verify`, mostrada na Figura 4, que verifica se está ordenado um array que contém todos os números naturais menores que o seu tamanho. O laço da função percorre o array, verificando se cada posição dele contém seu próprio índice.

```
int sort_verify(void) {
  int i, success = 1;
  for (i = 0; i < bots_arg_size; ++i)
    if (array[i] != i)
      success = 0;

  return success ? BOTS_RESULT_SUCCESSFUL : BOTS_RESULT_UNSUCCESSFUL;
}
```

Figura 4. Redução para verificar a ordenação de um array (programa *Sort*)

Por fim, o terceiro caso identificado pela ferramenta implementada, e não por Cetus, refere-se a reduções sobre uma variável acumuladora externa alcançada por meio de um ponteiro. A Figura 5 mostra um exemplo.

```
for (i = 0; i < n; i++) {
    a[j] = (char)i;
    if (ok(j + 1, a)) {
        nqueens(n, j + 1, a, &res);
        *solutions += res;
    }
}
```

Figura 5. Redução em acumulador alcançado por ponteiro (programa *NQueens*)

Esses laços não podem ser paralelizados trivialmente por meio de diretivas OpenMP. Em um uso da ferramenta implementada como uma ferramenta de paralelização, ficaria a cargo do programador usuário decidir se vale a pena realizar as modificações necessárias nos códigos para paralelizá-los.

Paralelização do laço mostrado na Figura 3 exigiria refatoração substancial do código. Poderia-se transformar a lista encadeada em um array de elementos posiciona-

dos adjacentemente na memória para que o laço então pudesse ser paralelizado por meio da cláusula `reduction` do OpenMP. Alternativamente, poderia-se particionar explicitamente a lista em segmentos e dedicar uma tarefa separada à travessia de cada um deles. O laço da Figura 4 poderia ser executado paralelamente como uma redução se fosse usado o operador lógico de conjunção (`&&`) ou o de disjunção (`||`) como operador de acumulação. Já a execução paralela do laço da Figura 5 como uma redução em OpenMP poderia ser realizada executando a redução sobre uma variável local, e só então atualizando a variável externa `*solutions` com o resultado final.

5.3. Identificações falso-positivas

Nesta seção são discutidos os 35 falsos positivos gerados pela abordagem proposta, visando identificar lacunas na especificação das regras empregadas.

O primeiro caso de falso positivo analisado, correspondente a 17 dos 35 casos, é apresentado na Figura 6. A implementação apontou a variável `t` como provável acumuladora de uma redução. Na realidade, essa variável é usada somente para guardar um resultado intermediário em cada iteração do laço. Um fato que pode ser observado quando se inspeciona o corpo da função que engloba esse laço é que `t` não é referenciada em qualquer lugar fora do laço, exceto na própria declaração dela. Isso implica que a variável não é lida depois do laço. Esse é um sinal de que `t` não é realmente um acumulador de redução. Essa verificação pode ser implementada em um melhoramento futuro do algoritmo: se o valor adquirido por uma variável em um laço não é consultado depois do laço, então é improvável que essa variável tenha agido como um acumulador de redução nesse laço.

```
for (j = 1; j <= N; j++) {
    HH[j] = t = t - gh;
    DD[j] = t - g;
}
```

Figura 6. Laço falsamente apontado como redução (programa *Alignment*)

O estudo do código do BOTS mostrou que é frequente o uso de variáveis declaradas externamente a um laço para armazenar resultados intermediários das suas computações, sendo seu valor final não utilizado ao final do laço. Estes falsos positivos poderiam ser evitados descartando CVAs que não são lidas depois do possível laço acumulador.

Os testes da implementação do algoritmo também evidenciaram outro problema que é solucionável por pequenas adições às regras de detecção. A implementação aponta como prováveis reduções laços que percorrem listas encadeadas, como mostrado na Figura 7. Laços desse tipo apresentam múltiplas das mesmas características apresentadas por laços de redução. Nesse laço, o ponteiro `aux` é apontado como um provável acumulador. Também há outros falsos positivos nos quais o aparente acumulador é na verdade um ponteiro. Esse problema pode ser solucionado descartando CVAs que denotam ponteiros[6]. Dos falsos positivos emitidos pela ferramenta no BOTS, 7 seriam evitados dessa forma. Desses falsos positivos evitáveis, 3 são tais que também seriam resolvidos descartando CVAs que não são posteriormente lidas, como descrito anteriormente.

[6]É importante observar que acumuladores como o `*results` mostrado na Figura 5 ainda seriam detectados. `*results` não é um ponteiro, apesar de `results` ser.

```
while (aux->forward != NULL)
    aux = aux->forward;
```

Figura 7. Laço que percorre uma lista encadeada (programa *Health*)

```
while (list != NULL)
    {
        p = list;
        list = list->forward;
        if (village->hosp.free_personnel > 0) {
            village->hosp.free_personnel--;
            p->time_left = sim_assess_time;
            p->time += p->time_left;
            removeList(&(village->hosp.waiting), p);
            addList(&(village->hosp.assess), p);
        } else {
            p->time++;
        }
    }
```

Figura 8. Falsa redução em elementos de uma lista encadeada (programa *Health*)

Diferentemente do Cetus, a implementação do algoritmo proposto é capaz de detectar reduções que são feitas em campos de estruturas, os quais podem ser acessados por meio dos operadores `.` e `->`. Isso a força a enfrentar falsos positivos que não são enfrentados pelo Cetus. Um exemplo é apresentado na Figura 8. Esse laço percorre uma lista encadeada. A implementação diz que `p->time` é um provável acumulador. Essa é uma conjectura inicial razoável, já que `p->time` é evidentemente aumentado em cada iteração do laço. No entanto, o objeto nomeado por `p` é alterado em cada iteração. Portanto, cada iteração na verdade escreve um campo de uma estrutura diferente. Esse falso positivo poderia ser contornado incorporando a percepção desse fato no algoritmo. É verificado um total de 3 falsos positivos nesta situação.

A Tabela 2 caracteriza os falsos positivos evitáveis emitidos pela implementação.

Resultado da análise	Casos
Não utilizada após o laço	17
Ponteiro	4*
Campo em estrutura diferente a cada iteração	3
Outros casos	11
Total	35

* A observação mostrou um total de 7 casos onde a variável candidata a acumulador se enquadrava no caso de ser ponteiro e não ser utilizada após o laço.

Tabela 2. Caracterização da variável candidata a acumulador nos falsos positivos

Parte dos 11 falsos positivos restantes talvez possa ser evitada calibrando-se os pesos associados a cada evidência observada pelo algoritmo. Essa calibragem seria feita

considerando a frequência em que essas características aparecem em laços de redução verdadeiros. Outra possibilidade é ajustar o limiar mínimo de pontuação que faz uma candidata a acumuladora ser considerada uma provável acumuladora (que é 0,7 no algoritmo proposto). Isso deve ser feito cuidadosamente. Quando a implementação é executada com um limiar mínimo igual a 0,8, 14 dos falsos positivos somem — mas também somem 2 das reduções verdadeiras. Limiares diferentes podem ser experimentados passando um argumento da forma `--min-score=<number>` para a execução.

6. Conclusão

Estratégias para detecção e paralelização automática de laços que são comumente usadas nos compiladores paralelizadores, como Cetus, se limitam à detecção de laços que podem ser paralelizados de forma automática. Neste trabalho foi apresentada uma heurística para detecção de laços de redução aplicada sobre códigos fonte de programas C que, diferentemente das propostas de trabalhos relacionados, procura identificar também laços que não podem ser paralelizados sem assistência do programador. A heurística consiste na avaliação de um conjunto de regras e na aplicação de uma pontuação sobre a influência no atendimento a cada uma das regras para um laço ser caracterizado como um *laço de redução paralelizável*. Não foram realizados estudos complementares para identificar a melhor distribuição de pontuação entre as regras apresentadas, mas a aplicada de forma empírica, mostrou-se já satisfatória.

O trabalho ainda mostrou a validação da heurística materializando-a na forma de uma ferramenta. Os casos de estudo, para análise dos resultados, foram realizados sobre programas de um benchmark concebido para avaliar desempenho de implementações OpenMP. Os resultados foram comparados com os fornecidos por Cetus. As diferenças foram analisadas com o objetivo de avaliar a estratégia proposta. De modo geral, os resultados obtidos se mostraram satisfatórios e motivam o prosseguimento da pequisa.

Na validação, foram analisados trechos de código apontados pela abordagem proposta. Foram determinados quantos dos laços apontados consistem de fato em reduções e quantos são falsos positivos. Foram investigadas as causas dos falsos positivos e identificadas alterações no modelo proposto para elevar a taxa de sucesso na identificação de reduções. Também foram identificados os casos de redução que são apontados por Cetus mas não pela proposta apresentada. Neste caso, embora considerada simples a alteração a ser incluída no modelo para também identificar o caso, que se trata do incremento (ou decremento) de uma variável contadora, entendeu-se ser necessário ampliar o estudo sobre a situação apresentada para indicar o real interesse de paralelização desses trechos.

Na paralelização de um programa, geralmente é desejável paralelizar laços que executem computações suficientemente custosas para compensar o sobrecusto de gestão da concorrência. O algoritmo proposto objetiva detectar quaisquer reduções, mesmo aquelas em que o ganho de desempenho somente possa ser obtido por interferência direta do programador, refatorando o código para efetivamente explorar a redução paralela identificada. A aplicação do algoritmo proposto como uma ferramenta de paralelização teria seu potencial totalmente explorado quando em associação a um mecanismo capaz de estimar a efetividade da paralelização de cada laço avaliado como candidato à paralelização. O desenvolvimento desse mecanismo para estimar desempenho em si abre um novo campo de trabalho, sendo uma frente futura de pesquisa em aberto.

Trabalhos futuros também podem focar no aperfeiçoamento da heurística proposta. De imediato, identifica-se que podem ser calibradas as pontuações envolvidas no algoritmo, como sugerido na seção 5. Também podem ser contempladas as observações identificadas na etapa de validação, como modificar o algoritmo para considerar acumulações feitas por meio de um operador unário de incremento e impedir a detecção de ponteiros como CVAs e de variáveis que não são lidas depois do laço. Dentro deste mesmo contexto, poderia ser contemplada a identificação de laços de redução que delegam a modificação do acumulador a uma função externa (com uma chamada de função sobre o acumulador a cada iteração `i` como em `accumulate(i,&accumulator)`). Esta situação não foi considerada no estudo realizado.

Como trabalho futuro também podem ser estendidos os estudos sobre as condições que indicam a ocorrência de um laço de redução que possa ser paralelizado e mesmo a incorporação de regras da identificação de laços paralelizáveis de outra natureza, como stencil e pipeline.

Referências

Amini, M. et al. (2012). Par4all: From convex array regions to heterogeneous computing. In *2nd International Workshop on Polyhedral Compilation Techniques, Impact*.

Bae, H., Mustafa, D., Lee, J.-W., Lin, H., Dave, C., Eigenmann, R., Midkiff, S. P., et al. (2013). The Cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. (2009). Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *2009 Inter. Conf. on Parallel Processing*, pages 124–131. IEEE.

Harel, R., Mosseri, I., Levin, H., Alon, L.-o., Rusanovsky, M., and Oren, G. (2020). Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming*, 48(1):1–31.

Jradi, W. A. R., Dantas do Nascimento, H. A., and Santos Martins, W. (2018). A fast and generic GPU-based parallel reduction implementation. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 16–22.

Lee, S.-I., Johnson, T. A., and Eigenmann, R. (2003). Cetus–an extensible compiler infrastructure for source-to-source transformation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 539–553. Springer.

Parhami, B. (2002). *Introduction to Parallel Processing: Algorithms and Architecture*. Kluwer Academic, New York.

Prema, S. and Jehadeesan, R. (2013). Analysis of parallelization techniques and tools. *International Journal of Information and Computation Technology*, 3(5):471–478.

Quinlan, D. and Liao, C. (2011). The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. ACM.

Tian, X., Hoeflinger, J. P., Haab, G., Chen, Y.-K., Girkar, M., and Shah, S. (2005). A compiler for exploiting nested parallelism in OpenMP programs. *Parallel Computing*, 31(10-12):960–983.