Uma proposta de algoritmo de detecção automática de operações de redução em programas C sequenciais

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

Resumo. *Compiladores paralelizadores buscam otimizar programas sequenciais pela exploração dos recursos paralelos oferecidos pelo hardware. As estratégias atualmente empregadas, no entanto, não são eficientes na detecção de paralelismo em laços cujo corpo das iterações seja representado por um bloco de comandos. Este artigo propõe uma heurística de detecção de paralelismo em laços contendo reduções de variáveis em um bloco de comandos.*

1. Introdução

Diversos compiladores paralelizadores operam sobre o código de programas escritos na linguagem C com o objetivo de explorar a concorrência detectável em programas sequenciais na exploração dos recursos paralelos oferecidos pelo hardware disponível. Alguns exemplos [Prema et al. 2019] são: Cetus [Lee et al. 2003], Par4All [Amini et al. 2012], e Rose [Quinlan 2000]. Essas ferramentas consistem em tradutores fonte-para-fonte capazes de identificar regiões que podem ser executadas paralelamente e anotá-las automaticamente com diretivas OpenMP apropriadas.

Uma transformação usualmente implementada em tais compiladores é a identificação e paralelização de operações de redução realizadas em laços. Uma redução é uma computação iterativa que produz um único valor a partir de uma sequência de valores, tipicamente por meio de aplicações sucessivas de uma operação binária, por exemplo, a soma. Um trecho de código representando a ocorrência de uma redução na linguagem C é mostrado na Figura 1.(a). Neste código, a variável `acc` sofre sucessivas modificações com a semântica do operador +=. A execução paralela de uma redução é obtida particionando a sequência de valores em múltiplas subsequências e incumbindo a redução de cada uma dessas subsequências a um thread distinto.

```
1 int reduce(int array[], size_t size) {
2     int i, acc = 0;
3     for ( i = 0; i < size; ++i )
4         acc = acc + array[i];
5     return acc;
6 }
```

(a)

```
1 int reduce(int array[], size_t size) {
2     int i, acc = 0;
3     for ( i = 0; i < size; ++i ) {
4         acc = acc + array[i];
5         printf("%d ", acc);
6     }
7     return acc;
8 }
```

(b)

Figura 1. Trechos com redução: (a) caso trivial, (b) ocorrência de efeito colateral.

As técnicas de detecção de reduções empregadas pelos compiladores paralelizadores existentes apresentam limitações. Em particular, elas reconhecem somente reduções que podem ser paralelizadas por meio de transformações diretas feitas por eles próprios.

Este artigo propõe uma estratégia de detecção estática de reduções em programas C que objetiva identificar mesmo reduções que não podem ser seguramente paralelizadas sem envolvimento do programador. O objetivo final é desenvolver uma ferramenta que sugira reduções a serem analisadas manualmente. No estágio atual da pesquisa, o objetivo é a detecção de oportunidades de paralelização, não a transformação do código.

2. Estado da Arte

Compiladores paralelizadores buscam identificar reduções que podem ser paralelizadas automaticamente por meio de transformações diretas do código. Essa transformação geralmente consiste na inserção de uma diretiva OpenMP `for` com a cláusula `reduction` ou ainda na exploração de recursos vetoriais como AVX, que estão disponíveis em alguns processadores. Compiladores paralelizadores ignoram reduções cuja paralelização exigiria refatoração significativa do código. Eles geralmente operam exigindo nenhuma ou mínima intervenção manual do programador [Harel et al. 2020]. Também existem ferramentas que apenas detectam reduções, sem fazer esforço para paralelizá-las. Um exemplo é a Parallel Pattern Analyzer Tool [Brown et al. 2020].

Um laço de redução é paralelizado automaticamente somente se sua variável acumuladora (`acc` no exemplo da Figura 1.(a)) não é referenciada em qualquer outra parte do corpo do laço além da expressão de atribuição que acumula valores nela (`acc = acc + array[i]` na Figura 1.(a)). Quando esse não é o caso, executar o laço paralelamente pode alterar seu efeito. Se há quaisquer outros comandos no corpo do laço que consultem o valor da variável acumuladora ao longo das suas iterações e o laço é executado paralelamente, esses comandos observarão uma sequência de valores diferente da que seria observada em uma execução sequencial dele. Por exemplo, o laço mostrado na Figura 1.(b), se executado sobre um array contendo os dados `{1, 1, 1, 1}`, tem como saída `1 2 3 4`: uma sequência que gradualmente leva ao valor final da redução. Mas, se o laço é anotado com a diretiva `#pragma omp parallel for reduction(+: acc)` e executado em um sistema com quatro processadores, ele pode resultar em `1 1 1 1`.

Outra limitação da abordagem tomada por compiladores paralelizadores existentes é que ela reconhece um laço de redução somente se ele consulta o valor corrente da variável acumuladora, produz um valor novo a partir dele e o atribui à variável acumuladora em uma única expressão de atribuição, como no laço da Figura 1. Passa despercebido qualquer laço de redução que capture o valor da variável acumuladora, realize uma sequência de operações sobre ele e atualize a variável acumuladora em múltiplos comandos. Compiladores paralelizadores operam, usualmente, somente sobre laços `for` [Prema and Jehadeesan 2013], desconsiderando outros tipos de laços.

O compilador paralelizador Cetus reconhece um laço `for` como um laço de redução se seu corpo contém uma expressão de atribuição da forma `acc = acc + expr` onde `acc` é uma variável escalar declarada fora do laço que não é referenciada em qualquer outra parte do laço e `expr` é uma expressão arbitrariamente complexa [Bae et al. 2013]. Outros compiladores paralelizadores tomam abordagens semelhantes. A inexistência de outras referências à variável `acc` dentro do laço é essencial à garantia de que um valor é acumulado nela. Essa inexistência garante que o laço não contém outros comandos que impeçam essa acumulação (por exemplo, `acc = 0;`).

Isso significa que a inexistência de outras referências à aparente 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. Isso implica que um algoritmo de detecção de reduções, se fosse deixar de fazer essa verificação, teria que investigar outras evidências de que um valor é acumulado naquela variável.

3. Abordagem Adotada

A estratégia aqui proposta toma uma abordagem heurística de reconhecimento de padrões. Laços de redução são identificados por meio de reconhecimento de características frequentemente apresentadas por eles. O algoritmo é listado na Figura 2. Nele, são usadas as notações seguintes. Se $laço$ é um laço, $variáveis\_externas\_modificadas(laço)$ é o conjunto de toda variável que é declarada fora do laço $laço$ mas é o lado esquerdo de pelo menos uma expressão de atribuição no corpo dele. A cada variável v é associado um valor $v.pontuação$, que é um número que representa a probabilidade de v ser uma variável acumuladora. Uma instrução $var \overset{+}{\leftarrow} num$ é equivalente a $var \leftarrow var + num$. $var \overset{-}{\leftarrow} num$ é equivalente a $var \leftarrow var - num$.

O algoritmo inicializa a pontuação de cada possível acumulador em um valor inicial, e então executa uma série de testes à procura de evidências de que ele seja ou não seja um acumulador enquanto atualiza a pontuação de acordo. O valor inicial da pontuação e os valores que são somados ou subtraídos dela em cada etapa dessa versão inicial do algoritmo são palpites. Eles serão calibrados com base nas frequências observadas dessas características em laços de redução de suítes de benchmark de computação paralela e de sistemas de software de fonte aberta.

4. Conclusão e Trabalhos Futuros

O algoritmo está sendo implementado na forma de uma aplicação do Clang, que providencia um conjunto de bibliotecas de processamento de código C. Depois que for terminada uma versão inicial da implementação, serão experimentadas diversas variações do algoritmo, atribuindo diferentes pesos às características observadas por ele, com o objetivo de identificar os pesos que levam à melhor taxa de detecção de reduções.

Referências

Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J. O., Pasquier, F.-X., Péan, G., Villalon, P., et al. (2012). Par4all: From convex array regions to heterogeneous computing. In *2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)*. Citeseer.

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.

Brown, C., Janjic, V., Barwell, A., Thomson, J., Lozano, R. C., Cole, M., Franke, B., Garcia-Sanchez, J. D., Astorga, D. D. R., and MacKenzie, K. (2020). A hybrid approach to parallel pattern discovery in c++. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 187–191. IEEE.

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

reduções $\leftarrow \emptyset$
para cada *laço* $\in$ *laços* **faça**
  para cada *possível_acumulador* $\in$ *variáveis_externas_modificadas(laço)* **faça**
    possível_acumulador.pontuação $\leftarrow 0.5$
    se $possível\_acumulador$ *é declarado imediatamente antes do laço* **então**
      possível_acumulador.pontuação $\overset{+}{\leftarrow} 0.2$
    fim
    se $possível\_acumulador$ *também aparece no lado direito das atribuições que o modificam* **então**
      possível_acumulador.pontuação $\overset{+}{\leftarrow} 0.3$
    fim
    para cada *referência a* $possível\_acumulador$ *que ocorre no corpo do laço mas fora das atribuições que o modificam* **faça**
      possível_acumulador.pontuação $\overset{-}{\leftarrow} 0.1$
    fim
    se *o nome de* $possível\_acumulador$ *tem como subpalavra "acc", "total", ou "sum"* **então**
      possível_acumulador.pontuação $\overset{+}{\leftarrow} 0.5$
    fim
    se $possível\_acumulador.pontuação \geq 0.7$ **então**
      reduções $\leftarrow$ *reduções* $\cup$ { *laço* }
    fim
  fim
fim

Figura 2. Algoritmo heurístico de detecção de reduções

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.

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.

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

Prema, S., Nasre, R., Jehadeesan, R., and Panigrahi, B. (2019). A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience*, 31(17):e5168.

Quinlan, D. (2000). Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226.