Detecção de Padrões Paralelizáveis de Laços em Programas Sequenciais Utilizando Análise das *Abstract Syntax Trees* (AST)

Edevaldo Santos[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]Universidade Federal de Pelotas
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

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

Abstract. *The present work proposes the detection of parallel patterns in sequential programs, through the analysis of the Abstract Syntax Trees (AST), and suggesting points where parallelization policies can be inserted in the codes, in order to make them parallel . The results of the parallelization suggestions of the tool will be compared with known versions of parallel benchmarks, taking the sequential versions of the tool as input.*

Resumo. *O presente trabalho propõe-se à detecção de padrões paralelos em programas sequenciais, por meio da análise das Abstract Syntax Trees (AST), e sugerindo pontos onde se podem inserir diretivas de paralelização nos códigos, de modo a torná-los paralelos. Os resultados das sugestões de paralelização da ferramenta serão comparados com versões conhecidas de benchmarks paralelos, tendo como entrada da ferramenta as versões sequenciais desses.*

1. Introdução

Ao longo dos anos 1990 e início do século XXI, os fabricantes de computadores competiam fortemente para melhorar a performance dos processadores em virtude do aumento de seu *clock*. Uma modificação no modelo de ganho de desempenho surgiu com o advento da arquitetura de processamento *multicore*. De modo a poder aproveitar o poder computacional destas arquiteturas, fez-se uso dos conhecimentos de programação concorrente - essa mais voltada para os recursos de hardware, enquanto a programação paralela tem foco nos problemas de aplicação, neste caso o termo programação concorrente é o mais interessante, pois supõe que abaixo dele haverá uma camada que se responsabilizará por mapear a ”concorrência”da aplicação sobre o ”paralelismo”do hardware - de modo que os processadores trabalhassem de com uma maior taxa de ocupação.

As aplicações de programação paralela são as mais diversas, desde entretenimento a astronomia, passando por sequenciamento de genoma humano etc. Para um melhor aproveitamento da capacidade computacional do sistema são utilizadas diversas técnicas de programação concorrente. No entanto, para explorar eficientemente estas técnicas e o paralelismo arquitetural é necessário conhecer os gargalos em estruturas de programação em geral, a fim de melhorar seu desempenho. Isso demanda um conhecimento não somente da programação sequencial, mas também da programação paralela, de modo que se tenha ciência dos pontos onde aplicar as diretivas de compilação para a

paralelização do código, das melhores alternativas de uso dos comandos, e da análise da dependência simultânea de dados etc. Segundo [Astorga et al. 2017], identificar regiões de código a serem paralelizadas consiste em um dos maiores desafios na transformação de código sequencial em paralelo. Portanto, a proposta aqui é não precisar levar em consideração este conhecimento prévio do programador sobre onde é melhor aplicar certas estruturas de programação concorrente, neste caso o programador precisaria apenas saber programar de forma sequencial e o arcabouço daria conta de reconhecer possíveis pontos de paralelização e sugerir ao programador alternativas de diretivas para paralelizar diversos pontos do código. Isso se daria pelo reconhecimento de padrões de programação concorrente, mais especificamente focando nos laços, já que estes são exemplos de estruturas que consomem uma larga porção do tempo de processamento, portanto, otimizando estas estruturas e dividindo as tarefas, consequentemente melhora-se o desempenho da execução do programa.

2. Objetivos

O objetivo deste trabalho é prover uma ferramenta que permita identificar, em um código sequencial, trechos de código que representem padrões recorrentes de concorrência de laços. Para atingir tal objetivo geral, tais metas devem ser atingidas:

- Identificar os padrões que serão trabalhados;
- Elaborar rotinas para a detecção de estruturas de laços em programas sequenciais a partir das ASTs, de modo a sugerir pontos onde é possível paralelizar o código;
- Calibrar as tais heurísticas para saber se elas funcionam bem;
- Detectar os tipos de laços existentes e os padrões paralelos apresentados;
- Classificar os tipos de padrões quanto ao seu grau de similaridade com padrões estabelecidos;
- Determinar a distância entre os padrões para que se possa classificá-los corretamente.

3. Fundamentação Teórica

Para conhecer os esqueletos ou padrões paralelos a serem reconhecidos neste trabalho, é necessária também uma revisão sobre estes padrões [Gonzalez-Velez and Leyton 2010]. Quanto ao reconhecimento dos laços paralelos e dos padrões paralelos, conhecidos como *Parallel Patterns*, uma literatura que é referência é [Mattson et al. 2004], a qual é precursora neste assunto e traz as estruturas destes padrões. Ainda segundo [Gonzalez-Velez and Leyton 2010], Cole [Cole 1988] foi o pioneiro no campo com a introdução dos quatro esqueletos iniciais: divisão e conquista, combinação iterativa, *cluster* e fila de tarefas (*task queue*).

A implementação de esqueletos propõe que padrões recorrentes na programação paralela sejam abstraídos e fornecidos aos desenvolvedores em forma de funções, com especificações que transcendem as variações de arquiteturas, e que permitem o aumento do desempenho. Esqueletos podem solucionar problemas de desenvolvimento de software paralelo [Cole 2004], da seguinte forma:

- Simplificando a programação paralela por meio do aumento do nível de abstração;
- Aumentando a portabilidade e o reuso de código;

- Melhorando o desempenho por meio de implementações otimizadas de esqueletos para arquitetura específicas;
- Oferecendo implementações, testadas e bem documentadas, de padrões frequentemente utilizados na programação paralela.

Na seção seguinte são descritos alguns conceitos que devem ser considerados na implementação dos esqueletos.

Segundo Cole [Cole 2004], quatro princípios devem ser observados no projeto e na aplicação de esqueletos paralelos:

- Oferecer o esqueleto de forma simples;
- Integrar os esqueletos com o paralelismo *ad-hoc*;
- Acomodar a diversidade;
- Mostrar o *pay-back*.

Baseado em suas funcionalidades, os esqueletos podem ser divididos em três categorias: paralelização de dados, paralelização de tarefas e resolução de problemas, conforme a Tabela 1.

Tabela 1. Classificação dos Esqueletos [Gonzalez-Velez and Leyton 2010]

Classificação	**Escopo dos Esqueletos**	**Exemplos de Esqueletos**
Paralelismo de Dados	Estruturas de Dados	*Map*, *Reduce*, *MapReduce*, *Fork*, *Scan*, *Zip*
Paralelismo de Tarefas	Tarefas	*Seq*, *Farm*, *Pipe*, *If*, *For*, *While*
Resolução de Problemas	Famílias de Problemas	Divisão e Conquista, Programação Dinâmica, *Branch* and *Bound*

Com relação à Paralelização de Dados, os esqueletos são voltados para problemas da camada de aplicação nos quais a concorrência se encontra na aplicação de uma mesma operação a um determinado conjunto de dados. A forma como a operação é aplicada caracteriza cada um dos esqueletos.

Quanto à Resolução de Problemas, a denominação destes esqueletos reflete sua natureza, a qual descreve inteiramente um comportamento algorıtmico de determinadas classes de problemas.

Já o Paralelismo de Tarefas permite identificar uma determinada ordem de execução de operações em função de uma sequência ou condição pré-determinada, podendo também identificar iterações condicionais.

4. Metodologia

Com o intuito de auxiliar o programador a paralelizar seu código e obter um melhor desempenho, o resultado do trabalho será a criação de uma ferramenta capaz de identificar seções de código paralelizáveis de algoritmos iterativos, que são comumente resolvidos em laços com estratégias de redução, *scan*, *stencil*, *mapreduce* etc.

A validação será feita pelo uso de *benchmarks* conhecidos, como PARSEC, NAS etc. Serão implementadas versões sequenciais destes códigos paralelos e então serão rodadas as versões paralelas originais destes códigos frente às sugestões de paralelização propostas pelo detector implementado neste trabalho, a fim de que se comparem ambas as versões.

É importante, para isso, classificar as estruturas detectadas, encaixando-as dentro do padrão correto, de modo a elencá-las dentro de seu tipo específico, para isso será necessário determinar a distância entre dois tipos diferentes, para que a ferramenta possa efetivamente diferenciá-los.

A partir disso serão feitas análises das expressões regulares de cada padrão detectado, para então analisar as ASTs dos programas sequenciais a fim de buscar determinar os padrões paralelos em questão.

Partindo de *benchmarks* conhecidamente paralelos, como NPB, Parsec..., serão elaboradas versões sequenciais dos programas para, a partir de suas ASTs, determinar as distâncias entre elas a fim de determinar se estão no mesmo caso ou não.

5. Conclusão

Pretende-se, ao final deste trabalho, obter uma ferramenta que detecte e aponte possíveis pontos de paralelização em códigos sequenciais, facilitando a vida de programadores que não têm um vasto conhecimento de programação concorrente, ou seja, a grande maioria dos programadores.

Referências

Astorga, D., Dolz, M., Miguel Sánchez, L., García, J., Danelutto, M., and Torquati, M. (2017). Finding parallel patterns through static analysis in c++ applications. page 109434201769563.

Cole, M. (2004). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389 – 406.

Cole, M. I. (1988). *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis. AAID-85022.

Gonzalez-Velez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. 40:1135–1160.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming*. Software Patterns Series. Pearson Education.