Análise Preliminar de Desempenho de Interpretação Paralela de Indivíduos de Programação Genética

Lucas Bicalho Oliveira[1], **Álvaro Luiz Fazenda**[1], **Vinícius Veloso de Melo**[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
São José dos Campos, SP – Brasil

{lucas.bicalho,alvaro.fazenda,vinicius.melo}@unifesp.br

Abstract. *Genetic Programming is a program evolution technique used nowadays, which generally requires parallelism tools to improve performance, since it is possible to find many concurrent instructions. This document demonstrates the initial performance tests for a genetic code that uses multithreaded programming using MIMD (Multiple Instruction, Multiple Data) instruction sets analyzing its current strengths and limitations, in order to find a way to improve its performance in future developments.*

Resumo. *A Programação Genética é uma técnica de evolução de programas que, por muitas vezes, requer ferramentas de paralelismo para melhorias de desempenho, uma vez que geralmente há muitas instruções que podem ser executadas concorrentemente. Esse documento demonstra os testes iniciais de desempenho de um código genético que utiliza-se de programação multithread que faz uso de um conjunto de instruções do tipo MIMD (Multiple Instruction, Multiple Data), analisando suas potencialidades e limitações atuais, de forma a encontrar um caminho que permita melhorar seu desempenho em futuros desenvolvimentos.*

1. Introdução

Após o surgimento de computadores pessoais com mais de um núcleo de processamento, oriundo da saturação do desempenho das máquinas *singlecore*, foi exigido do programador uma mudança de paradigma na programação destes dispositivos, de maneira a aproveitar eficientemente os recursos disponíveis. Assim, a programação concorrente/paralela tornou-se necessária para melhoria de desempenho de programas que apresentem grande demanda computacional. A concorrência em programação em nível de procedimentos, pode ser gerada utilizando-se de múltiplas *threads* ou múltiplos processos, dependendo da arquitetura do sistema a ser utilizado.

Muitas bibliotecas ou linguagens são projetadas e desenvolvidas para melhorar a produtividade de programação. Construções paralelas, como, por exemplo, laços paralelos do OpenMP [Chandra 2001], permitem identificar conjuntos de instruções potencialmente concorrentes, sendo esta uma abordagem comumente utilizada em aplicações numéricas.

A Programação Genética [Koza 1994] é uma técnica de algoritmo evolutivo que permite a melhoria de programas de computador para executar uma programação automática. Nesta técnica, representa-se indivíduos como expressões, as quais permitem

avaliar funções envolvendo variáveis, constantes ou ainda outras funções, as quais são recursivamente calculadas para obter um melhor valor final para o indivíduo. Essa etapa geralmente é que mais demanda poder de computação do algoritmo. Por exemplo, para tarefas de classificação ou regressão, um programa genético (PG) precisa, normalmente, utilizar um algoritmo interpretador para calcular o resultado de cada indivíduo, para cada amostra no conjunto de dados utilizado.

Neste trabalho, procurou-se avaliar a aplicação do OpenMP no código genético apresentado por Melo et al [de Melo et al. 2020], desenvolvido pelo principal autor do trabalho, mostrando os parâmetros mais sensíveis à implementação efetuada, suas limitações e propor melhorias futuras.

2. Referencial Teórico

Vários trabalhos na literatura tratam de investigar diferentes abordagens para otimizar o desempenho no processo de avaliação das expressões em PGs. Uma técnica em particular foi desenvolvida para uso em modernas CPUs (*Central Processing Units*), e busca explorar a vetorização de instruções do tipo SIMD (*Single Instruction, Single Data*), normalmente disponível nas CPUs que permitem o uso de instruções SSE (*Streaming SIMD Extensions*) [Chitty 2012] ou AVXintrinsics[1] (Advanced Vector eXtensions). Desta forma, simples instruções, tal como adições, podem ser aplicadas a conjuntos (*arrays*) de dados de forma concorrente, normalmente acelerando desempenho.

O artigo de Melo et al [de Melo et al. 2020] demonstrou uma evolução na técnica citada propondo o uso de instruções do tipo MIMD no interpretador do PG, permitindo realizar até quatro instruções matemáticas diferentes simultaneamente. O trabalho desenvolvido avalia o desempenho em um única geração do procedimento usual de um PG (não incluindo as etapas de seleção, cruzamento e mutação), uma vez que o foco do trabalho reside na investigação de desempenho do interpretador MIMD desenvolvido, o qual avalia indivíduos aleatoriamente gerados.

3. Resultados Preliminares

Nos experimentos, usou-se dados sintéticos para avaliação de desempenho paralelo do interpretador. Todos os indivíduos de cada *dataset* apresentam 10 variáveis aleatoriamente distribuídas no intervalo $[-1.0, 1.0]$. O código possui três parâmetros de entradas. O primeiro parâmetro (*size*) corresponde a quantidade de registros no *dataset*, sendo definidos os valores: 10k, 50k, 100k; o segundo (*evals*) corresponde ao tamanho da população (quantidade de indivíduos), onde cada indivíduo representa uma transformação a ser aplicada ao *dataset*, sendo definidos: 1k, 5k, 10k; e o terceiro (*depth*) corresponde à profundidade máxima da árvore utilizada, onde cada nó da árvore representa uma operação aritmética realizada por cada indivíduo. O modo MIMD permite que se use árvores com menor altura em relação ao modo SIMD, visto que cada operação MIMD engloba várias operações mais simples do tipo SIMD.

A versão do código testada incorpora a técnica de blocagem, em que as funções recebem os índices de início e fim, representando um subconjunto de dados dispostos em uma matriz, contendo dados e operações a serem realizadas em cada indivíduo. A

[1] `https://software.intel.com/en-us/node/523876`

ideia da blocagem é aproveitar os dados na memória cachê, permitindo o reuso de dados para o calculo da expressão em sua totalidade. Além disso, a divisão em blocos possui a vantagem de permitir que cada *thread* trabalhe em blocos diferentes.

Ao se medir os tempos de execução variando os parâmetros de entrada entre diferentes valores e executando-se com diferentes números de *threads*, pode-se obter alguns resultados preliminares que retratam algumas características da aplicação. O código foi testado com o terceiro parâmetro fixado com o valor 5, ou seja, uma árvore com grau de profundidade 5. Em testes anteriores, percebeu-se claramente que esse é o parâmetro de entrada que mais influencia os resultados de execução. Então, chegou-se a esse valor como ideal para obter resultados significativos em um tempo hábil para se executar uma grande combinação de entradas.

Considerando uma análise comparativa entre três diferentes valores de entrada para o segundo parâmetro, fixando-se o primeiro parâmetro como um valor significativo, obteve-se um resultado previsto, em que os tempos de execução para cada variação comportam-se de forma proporcional. Isso pode ser comprovado ao se analisar o *speedup* (Figura 1), em que a curva para os três valores de entrada são quase iguais, ou seja, isso implica que o segundo parâmetro possui um comportamento regular e sua variação influencia fracamente nos resultados de desempenho em comparação com os demais parâmetros.

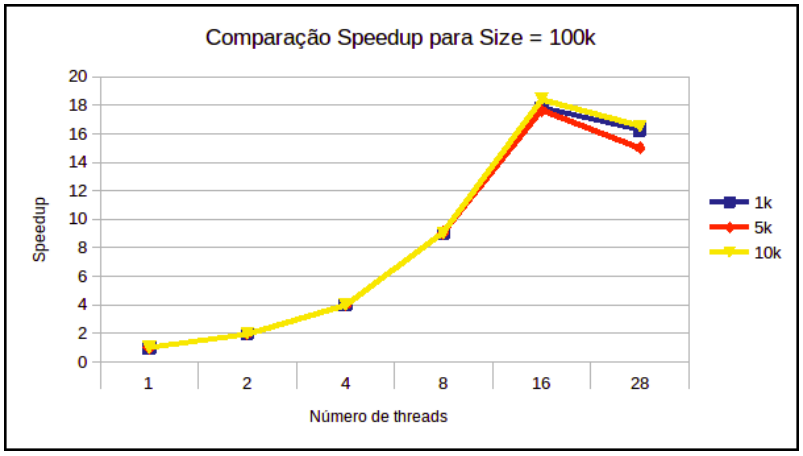

Figura 1. Comparação *Speedup* para *Size* = 100k

Já ao se fazer uma análise comparativa entre três diferentes valores de entrada para o primeiro parâmetro, fixando-se o segundo parâmetro como um valor significativo, conforme consta na Figura 2, nota-se que o desempenho já se torna muito mais desigual entre os valores. Isso significa que o primeiro parâmetro influencia muito mais no tempo de execução em comparação com o segundo parâmetro, embora ainda seja menos relevante que o terceiro parâmetro, o qual manteve-se fixo para ambos os testes.

Fica evidente um comportamento superlinear ao se executar com 8 e com 16 *threads*. No segundo caso, chega-se a atingir uma eficiência de mais de 115% para o maior valor de entrada. Tal efeito pode ter ocorrido em função de um melhor uso de memória cachê. Todavia, maiores investigações serão conduzidas no futuro para identificar esse comportamento.

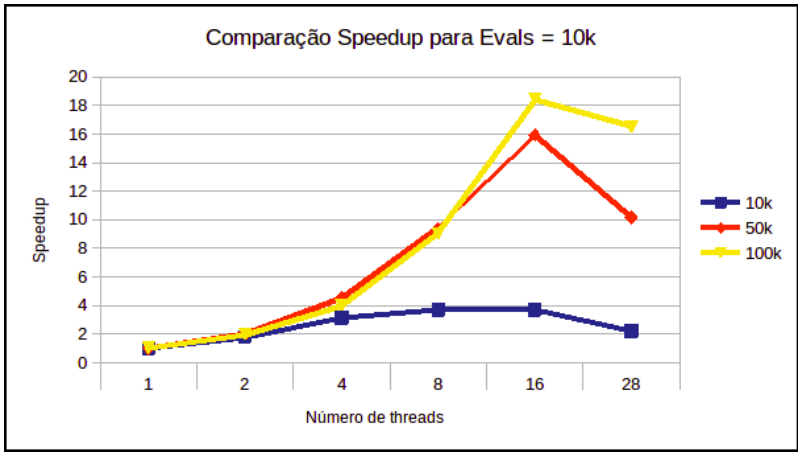

Figura 2. Comparação *Speedup* para *Evals* = 10k

Entretanto, também fica mais visível que o comportamento para execuções com 28 *threads* piora significativamente em comparação com as quantidades inferiores de *threads*. A eficiência cai de um nível superlinear para cerca de 60%. Foi também identificado que o trecho do código que exige maior demanda computacional (gargalo) corresponde ao trecho responsável pela avaliação das expressões na forma MIMD, aplicada a cada indivíduo.

4. Conclusão e Trabalhos Futuros

Tendo em vista os comportamentos discutidos acima, há a intenção de se realizar uma instrumentação fina no código para se identificar o quanto é gasto em execução em cada trecho, e se há algum ponto específico de gargalo, onde o tempo de execução não se altera ou diminui de forma desprezível com o aumento do número de *threads*. Duas frentes serão melhor investigadas: a identificação de possíveis trechos seriais (Lei de Amdhal) e o efeito de tempos de sincronização e coordenação de tarefas concorrentes pelo OpenMP.

A partir de uma futura detalhada instrumentação do código, juntamente com testes adequados, pretende-se identificar as razões que atualmente levam a uma queda abrupta no desempenho a partir de 12 *threads*/processadores. Pretende-se também realizar um processo semelhante de paralelismo no código utilizando-se a ferramenta OpenACC, além de realizar execuções em aceleradores, possivelmente GPUs.

Referências

Chandra, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, CA.

Chitty, D. M. (2012). Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing*, 16(10):1795–1814.

de Melo, V. V., Fazenda, Á. L., Sotto, L. F. D. P., and Iacca, G. (2020). A mimd interpreter for genetic programming. In Castillo, P. A., Jiménez Laredo, J. L., and Fernández de Vega, F., editors, *Applications of Evolutionary Computation*, pages 645–658. Springer International Publishing, Cham.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112.