Investigando a Efetividade de LLMs na Otimização de Código Paralelo

Pedro Cattai[1], **Alexandro Baldassin**[1]

[1] Universidade Estadual Paulista (UNESP) – Rio Claro, SP

{pedro.cattai, alexandro.baldassin}@unesp.br

Resumo. *Um dos desafios da programação paralela é o equilíbrio entre a corretude e a eficiência dos algoritmos. Com a evolução de técnicas baseadas em inteligência artificial, uma questão que se apresenta é o quanto tais técnicas conseguiriam auxiliar um programador a otimizar o código, sem comprometer sua corretude. Este artigo avalia o uso de Large Language Models (LLMs), como o ChatGPT e o GitHub Copilot, para a otimização de código paralelo escrito em OpenMP. Os resultados experimentais mostram que as otimizações do ChatGPT geralmente apresentam algum erro e não são notavelmente melhores, com raras exceções. O Copilot, entretanto, apresenta resultados mais consistentes e confiáveis.*

1. Introdução

Na área de programação paralela, um desafio é o equilíbrio entre a corretude e a eficiência dos algoritmos [4]. Por exemplo, é possível que ocorra uma perda de corretude do algoritmo após uma tentativa de otimização, devido às sutilidades do processamento paralelo. Recentemente, o campo de aprendizado de máquina evoluiu exponencialmente, permitindo o desenvolvimento de modelos computacionais mais poderosos. Um tipo de modelo que ganhou relevância é o LLM (Large Language Model) [2, 3]. Os LLMs são úteis para geração de texto, tradução automática, recomendação de conteúdo e assistentes virtuais, por exemplo. Estes modelos são treinados para realizar tarefas ligadas à linguagem humana, como completar frases ou gerar respostas de acordo com uma entrada. Se treinálos com base em códigos de programação além de texto comum, os modelos se tornam capazes de funções úteis para a programação, como completar códigos automaticamente, gerar códigos a partir de *prompts*, e também sugerir otimizações.

Levando em consideração a evolução dos LLMs e os desafios da programação paralela, este trabalho de iniciação científica realizou uma pesquisa sobre o possível desempenho de LLMs nas tarefas de otimização de programação paralela. As ferramentas usadas nesta análise foram o ChatGPT e o GitHub Copilot [3]. O ChatGPT é um produto interativo treinado para responder o usuário naturalmente e, entre suas capacidades, está a otimização de código. O GitHub Copilot, porém, foi desenvolvido especificamente para programação e foi treinado com bilhões de linhas de código. Neste trabalho é feita uma comparação entre as duas LLMs para medir a eficiência das otimizações sugeridas. Os testes utilizaram programas na linguagem C com OpenMP. De maneira geral, os resultados obtidos mostram que o ChatGPT não é uma ferramenta confiável quando se trata de programação paralela, cometendo diversos erros. O Copilot, entretanto, mostrou-se mais eficiente e seguro, apresentando resultados geralmente satisfatórios.

2. Base Teórica

Esta seção apresenta alguns aspectos relacionados à programação paralela que dificultam a vida dos programadores, justificando o uso destes problemas no trabalho realizado. Em seguida, apresenta uma explicação básica do funcionamento dos LLMs.

2.1. Desafios da programação paralela

Problemas de alta complexidade geralmente são difíceis de paralelizar e otimizar. A complexidade pode induzir uso excessivo de diretivas nas tentativas de paralelização, causando uma sobrecarga que acaba piorando o desempenho. Tome como exemplo o problema das N damas. O problema consiste em calcular a quantidade de maneiras de distribuir N damas num tabuleiro de xadrez de tamanho NxN sem que elas possam se atacar. A complexidade do cálculo aumenta rapidamente de acordo com o tamanho do tabuleiro. O algoritmo pode ser expresso também de forma recursiva na sua versão sequencial, o que traz dificuldades para a sua paralelização.

Há casos em que o problema em questão não pode ser paralelizado, por exemplo devido à *loop-carried dependence* (dependências entre iterações de um laço). Assim, é necessário alterar a maneira que o laço é escrito. Às vezes isso não é suficiente, e a solução demanda uma mudança do algoritmo em si. Isso ocorre com o algoritmo de ordenação *bubble sort*. Este algoritmo consiste em percorrer o vetor com uma ”bolha”, ordenando os números neste agrupamento e garantindo a ordenação geral com as demais vezes que percorre o vetor. É impossível paralelizar este algoritmo sem alterar sua lógica pois existe sobreposição de cada passo da bolha com o anterior.

2.2. Large Language Models (LLMs)

Os LLMs são modelos de *deep learning* eficazes na realização de tarefas relacionadas ao processamento de linguagem natural (NLP) [1]. Sua arquitetura comum é o *transformer*, o qual processa dados através da divisão do texto em partes relevantes (*tokens*). Feito isso, o transformer trata os *tokens* matematicamente, extraindo relações linguísticas entre as partes, tudo isso paralelamente. A arquitetura dos LLMs permite que, quando treinados com uma grande quantidade de código, aprendam os padrões da programação, incluindo sintaxes de linguagens, semânticas de algoritmos e técnicas de otimização. Com isso em mente, este trabalho teve o objetivo de levar o LLM a uma resposta satisfatória através de sequências de *prompts* na forma de perguntas e correções.

3. Metodologia

Foram realizados diversos testes com o ChatGPT, incluindo geração de código paralelo baseado na descrição de um problema, paralelização de código sequencial e otimização de códigos paralelos. Os experimentos foram estruturados da seguinte maneira. Primeiramente, demos a instrução desejada diretamente para o ChatGPT. Por exemplo, para os problemas de otimização, fornecemos o código e pedimos a sua otimização diretamente. Posteriormente, se a resposta gerada apresentava algum erro de corretude ou desempenho, continuamos a conversa com o LLM para extrair uma resposta melhor.

Depois dos testes com o ChatGPT, foram escolhidos os exemplos mais relevantes para serem feitos no Github Copilot. Quando relevante, foram feitas medições de tempo

para comparar o desempenho dos códigos gerados pelos LLMs. Cada medição foi realizada cinco vezes e a média foi extraída. Na Tabela 1, as colunas representam os tempos medidos para a versão sequencial dos códigos, seguida pelas versões geradas pelas ferramentas utilizadas. Quanto aos nomes das colunas, "ChatGPT v1", por exemplo, se refere ao primeiro experimento realizado com uma geração do ChatGPT para aquele problema com oito threads. A coluna "ChatGPT v2", portanto, representa a versão aprimorada obtida ao longo da interação com o LLM.

Os experimentos foram realizados numa máquina com CPU Intel Core i7-3770 @ 3.40GHz, com quatro núcleos físicos e oito processadores lógicos. A máquina possui 8,00 GB de RAM. O sistema operacional utilizado é o Windows 10, com as execuções em si ocorrendo dentro do WSL Ubuntu 22.04.2 LTS[1].

4. Resultados

Esta seção apresenta e discute os resultados de alguns dos experimentos realizados. Primeiramente, discute o problema das N damas, mostrando como as ferramentas responderam à complexidade do programa. A seguir, discute o *bubble sort*, com foco nas respostas dos LLMs à impossibilidade de paralelização.

4.1. N Damas

Neste problema, o ChatGPT primeiramente gerou um código com sintaxe incorreta. Quando alertado, o LLM falhou em corrigir-se e, somente após recomendarmos o uso de *tasks* para lidar com a recursão, o ChatGPT alterou sua solução. Assim, o LLM gerou uma versão correta do código, porém com um desempenho pior do que a versão sequencial, como consta na Tabela 1. Com isso, o objetivo da interação com o ChatGPT passou a ser a otimização do código gerado. Nesta fase, o ChatGPT gerou uma versão com uma solução totalmente diferente, utilizando operações em bits no lugar do *backtracking* usado anteriormente. Desta vez, o desempenho foi muito melhor do que a versão sequencial do código. Notou-se que o ChatGPT é insistente em seus erros, mas se o problema for conhecido o suficiente, é possível que o modelo saiba de uma solução alternativa otimizada.

Com o Copilot, o primeiro código gerado estava correto e com desempenho melhor que o sequencial, como descrito na Tabela 1. Portanto, já avançamos para a fase de otimização. Então, o Copilot realizou algumas melhorias de desempenho, entretanto, cometeu um erro lógico ao realizar uma paralelização. Ao ser alertado, foi gerada uma versão correta do código. Desta vez, o desempenho foi ainda melhor. Porém, a versão mais rápida obtida foi a gerada pelo ChatGPT. Isso ocorreu pois o ChatGPT alterou a abordagem utilizada para solucionar o problema, como descrito anteriormente. O Copilot, no entanto, focou em melhorar o desempenho da abordagem já em uso.

4.2. Bubble sort

Considerando que este algoritmo não é paralelizável sem alterar sua lógica, realizamos estes testes para observar como o ChatGPT responderia a este fato. Começamos pedindo a geração de um código que executasse o *bubble sort* paralelamente. O LLM não considerou a impossibilidade da paralelização e tentou fazê-la. O código gerado incluia um erro

[1]Todo o código produzido, bem como as interações com os LLMs usadas neste artigo podem ser encontrados em `https://github.com/Pedro-Cattai/IC_Otimizacao`.

Tamanho	Sequencial	ChatGPT v1	ChatGPT v2	Copilot v1	Copilot v2
8x8	0,51 ms	3,63 ms	0,66 ms	0,74 ms	0,46 ms
12x12	301,84 ms	1032 ms	8,12 ms	136,14 ms	39,63 ms
14x14	11487,4 ms	34174 ms	225,8 ms	4989,7 ms	1406,2 ms

Tabela 1. N Damas - medições de tempo

sintático no OpenMP que foi resolvido quando indicado, porém a geração ainda tentava paralelizar o bubble sort. A partir deste ponto, a interação consistiu em tentar explicar para o LLM que o *bubble sort* não é paralelizável desta forma, a fim de guiá-lo para sugerir um algoritmo substituto, como por exemplo o *odd-even sort* [5]. Entretanto, as respostas permaneceram insistentes na paralelização do *bubble sort*, e somente quando explicitamente alertado que o ChatGPT sugeriu a troca de algoritmo. O resultado deste teste usando o GitHub Copilot foi o oposto. Quando pedida a geração de um bubble sort paralelo usando OpenMP, o Copilot imediatamente aplicou a lógica do *odd-even sort*, separando as fases do *bubble sort* em pares e ímpares, a fim de evitar as dependências que impediam a paralelização.

5. Conclusão

Com base nos resultados obtidos ao neste trabalho, observamos que o ChatGPT não é uma ferramenta confiável ou eficiente no quesito de programação paralela, mostrando resultados geralmente negativos e com diversos erros, salvo em raras exceções. O GitHub Copilot se mostrou mais adequado, gerando resultados melhores e evitando erros simples. Na paralelização, o ChatGPT apresenta comportamentos similares. Quando o problema é simples, a paralelização ocorre facilmente. Mas quando há alguma complicação, como no caso do *bubble sort*, os limites do modelo se mostram através da sua insistência nos erros. Novamente, o Copilot é superior, pois lidou imediatamente com a questão da paralelização do *bubble sort*, e raramente apresentou erros na paralelização. Na otimização, o ChatGPT pode providenciar soluções alternativas mais rápidas do que a apresentada, como visto no exemplo das N Damas. Entretanto, em boa parte dos casos as otimizações sugeridas pelo modelo eram superficiais, ou já eram garantidas pelas opções de otimização do compilador.

Referências

[1] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2023. arXiv: `2307.06435 [cs.CL]`.

[2] E. Opara, Mfon-Ette A. e T. C. Aduke. "ChatGPT for Teaching, Learning and Research: Prospects and Challenges". Em: *Global Academic Journal of Humanities and Social Sciences* 5.2 (mar. de 2023), pp. 33–40.

[3] C. Stokel-Walker e R. Van Noorden. "What ChatGPT and generative AI mean for science". Em: *Nature* (fev. de 2023), pp. 214–216.

[4] H. Sutter e J. Larus. "Software and the Concurrency Revolution: Leveraging the Full Power of Multicore Processors Demands New Tools and New Thinking from the Software Industry." Em: *Queue* 3.7 (set. de 2005), pp. 54–62.

[5] P. Tarasiuk e M. Yatsymirskyy. "Optimized Concise Implementation of Batcher's Odd-Even Sorting". Em: *2018 IEEE Second International Conference on Data Stream Mining Processing (DSMP)*. 2018, pp. 449–452.