Explorando Modelos GPT na Geração de Código Paralelo para Aplicações de Computação Stencil

João Vitor M. Dias, Antonio Carlos S. Beck, e Arthur F. Lorenzon[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{joao.dias, caco, aflorenzon}@inf.ufrgs.br

Resumo. *Este trabalho explora a aplicação dos modelos de linguagem GPT para a geração automática de código paralelo em aplicações de Computação Stencil, ressaltando a importância desta abordagem para reduzir o tempo de desenvolvimento e aumentar a eficiência computacional. Ao aplicar os modelos GPT-3.5 turbo e o GPT-4o em oito aplicações Stencil com estruturas de dados variadas e executadas em três arquiteturas multicore, mostramos que ambos os modelos geraram códigos paralelos que melhoraram o desempenho em relação às versões sequenciais. No entanto, o GPT-4o não demonstrou um desempenho superior ao GPT-3.5 turbo e ambos enfrentaram desafios na gestão de dependências de dados. De um modo geral, mostra-se também que, através do uso correto das diretivas e cláusulas paralelas do OpenMP para exploração do paralelismo, o GPT-3.5 turbo foi capaz de gerar códigos paralelos com desempenho 15% superior aos do GPT-4o.*

1. Introdução

A programação paralela tem sido amplamente utilizada para explorar os recursos computacionais disponíveis (e.g., núcleos de processamento e memórias *caches*) com o objetivo de reduzir o tempo de execução das aplicações. Deste modo, para uma paralelização efetiva, desenvolvedores precisam considerar quais trechos de código (e.g., laços ou funções) podem ser paralelizados e que resultarão em ganhos de desempenho [Cesare et al. 2020, Navaux et al. 2023, Marques et al. 2022]. Exemplos de aplicações amplamente utilizadas pela comunidade acadêmica e industria, incluindo simulações sísmicas e algoritmos de álgebra linear, fazem o uso da computação *Stencil*, uma classe de algoritmos em que os elementos de uma estrutura de dados (e.g., matriz, vetor, grade), são atualizadas de acordo com um padrão fixo. No entanto, para muitas aplicações que envolvem a computação *Stencil*, a exploração do paralelismo pode resultar em grande esforço para desenvolvedores, causando aumento de custo monetário no desenvolvimento de aplicações. Deste modo, o uso de modelos de inteligência artificial para geração de códigos paralelos tem se popularizado com o objetivo de reduzir o tempo de desenvolvimento.

Ferramentas como o *AutoPar* [Kalender et al. 2014] têm se mostrado eficazes na geração automática de códigos paralelo, automatizando processos que tradicionalmente requerem intervenção manual significativa. Estas ferramentas são empregadas para simplificar o desenvolvimento e aumentar a produtividade, permitindo que desenvolvedores se concentrem em aspectos mais complexos e específicos da aplicação. No entanto, mesmo essas soluções podem enfrentar dificuldades em cenários altamente complexos, incluindo a modelagem de fenômenos físicos tridimensionais, como simulações de

dinâmica de fluídos ou propagação de ondas em meios heterogêneos, onde a computação *Stencil* é usada para atualizar os valores em uma grade tridimensional. Estes exemplos justificam a crescente adoção de modelos de linguagem de grande escala (LLM - *Large Language Models*) para essa tarefa. LLMs são amplamente utilizados devido à sua capacidade de aprender padrões complexos e gerar código que explora eficientemente o paralelismo, reduzindo ainda mais o tempo de desenvolvimento e aumentando a acessibilidade da programação paralela.

Nesse contexto, a adoção de modelos da família GPT(*Generative Pre-trained Transformer*), desenvolvidos pela OpenAI, tem ocorrido devido à sua capacidade de pré-treinamento em aprendizado não supervisionado seguido de um *fine tuning*) supervisionado, resultando em avanços significativos em tarefas de processamento de linguagem natural [Radford et al. 2018]. Esses modelos, que evoluíram significativamente desde suas primeiras versões, são capazes de entender e gerar código de alta qualidade, atendendo a requisitos específicos de desempenho e escalabilidade. Através do uso de grande quantidade de dados de treinamento, os modelos GPT podem identificar padrões complexos e fornecer soluções de paralelização que seriam difíceis de alcançar por meio de métodos tradicionais. Além disso, a flexibilidade desses modelos permite que sejam aplicados em uma ampla gama de domínios, tornando-se uma ferramenta interessante para desenvolvedores que buscam otimizar códigos paralelos.

Considerando o destacado anteriormente, este trabalho contribui para o campo da computação paralela ao investigar a viabilidade do uso dos modelos da família GPT para a geração de código paralelo em aplicações que usam Computação *Stencil*. Assim, os objetivos deste artigo são os seguintes: (*i*) avaliar como os modelos da família GPT podem ser empregados para automatizar a criação de código paralelo, abordando a importância de melhorar o desempenho e reduzir o tempo de desenvolvimento; (*ii*) comparar diferentes versões dos modelos GPT para entender como as variações na arquitetura e treinamento afetam a qualidade do código gerado e o desempenho das aplicações; e, (*iii*) analisar o tratamento de códigos com dependências de dados pelos modelos.

Utilizando duas versões da família GPT (*3.5-turbo* e *4o*) para gerar código paralelo em oito aplicações do tipo *Stencil*, com diferentes dimensões de estruturas de dados (1D, 2D e 3D) e executadas em três arquiteturas *multicore* com diferentes números de núcleos computacionais, demonstramos que: (*i*) Ambas as versões foram capazes de produzir código paralelo com OpenMP que resultou em ganhos de desempenho em relação à versão sequencial na maioria das aplicações. Contudo, embora o *GPT-4o* tenha gerado código mais genérico e abrangente, isso não se traduziu em um desempenho superior na maioria dos casos; (*ii*) Nenhuma das versões foi capaz de identificar automaticamente a dependência de dados entre as *threads* e gerar código paralelo correto para aplicações com essa característica. No entanto, quando explicitamente informada sobre a dependência de dados, a versão *GPT-3.5 turbo* conseguiu gerar códigos paralelos que tratavam corretamente essa dependência em duas aplicações. Por outro lado, o *GPT-4o* não conseguiu lidar com a dependência de dados, mesmo com um *prompt* explícito; (*iii*) Em termos de desempenho geral, considerando todas as aplicações e arquiteturas, a versão *GPT-3.5 turbo* proporcionou uma aceleração média 15% superior em comparação com os códigos gerados pelo *GPT-4o*.

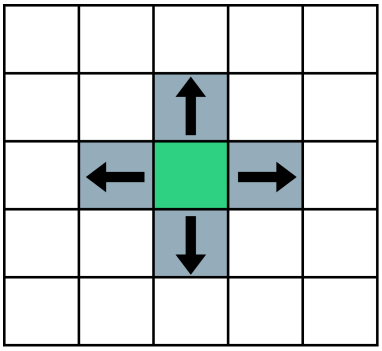


Figura 1. Stencil 2D

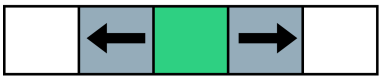


Figura 2. Stencil 1D

2. Fundamentação Teórica

Nesta Seção, fornecemos a fundamentação teórica relacionado à LLMs, com foco nos modelos GPT 3.5-turbo e 4o, além de uma visão geral sobre o uso do Padrão Stencil. Também, listamos os trabalhos relacionados realizados nesta área de pesquisa.

2.1. LLMs - Large Language Models

LLMs são modelos treinados com grandes quantidades de dados para realizar tarefas de processamento de linguagem natural, que usualmente utilizam a arquitetura de rede neural *transformer* [Vaswani et al. 2023]. Essa arquitetura se diferencia por usar mecanismos de atenção ao invés de redes neurais convolucionais ou recorrentes. O fluxo de dados em uma arquitetura *transformer* para processamento de linguagem natural inicia com a *tokenização* das palavras, seguida pela vetorização dos *tokens* gerados para formar *word embeddings*. Após a codificação do *input* estar pronta, os vetores passam por uma camada de atenção, a qual atribui pesos da importância de cada vetor em relação aos outros vetores do *input* para o cálculo do *output token*. Essa camada é baseada em *multiheaded self attention*, na qual cada cabeça de atenção do modelo é um espaço vetorial e está relacionada a uma possível relação entre os vetores. Após, a previsão dos *outputs tokens* é feita a partir de redes neurais diretas.

Os dois modelos utilizados neste trabalho são da família de modelos GPT. A série de modelos GPT da OpenAI, foi introduzida pela primeira vez por [Radford et al. 2018]. A família de modelos GPT-3 foi lançada em 2020, treinada a partir de um vasto conjunto de textos disponíveis na internet utilizando aprendizado não supervisionado, o que lhe confere a habilidade de prever a próxima palavra em uma sequência com base no contexto anterior. Uma característica notável do GPT-3 é sua capacidade de desempenhar diversas funções com uma quantidade mínima de dados de treinamento, conforme destacado em [Brown et al. 2020]. Neste estudo, utilizamos a versão GPT-3.5-turbo. Já a família GPT-4, lançada em 2023, trouxe inovações como a capacidade de processar entradas de texto e imagem. Uma das principais características do GPT-4 é o aprimoramento do modelo através do Aprendizado por Reforço com Feedback Humano (RLHF), que melhora a qualidade das respostas oferecidas aos usuários, conforme discutido em [Achiam et al. 2023]. Em nosso trabalho, utilizamos a versão GPT-4o.

2.2. Padrão Stencil

O Padrão Stencil refere-se a uma classe de algoritmos baseada na atualização de pontos de uma grade de valores com base em informações dos pontos vizinhos. Esses algoritmos

Algorithm 1 Algoritmo Stencil 2D

Inicializa a matriz A com valores iniciais
Defina o número de iterações N
for cada iteração n de 1 até N **do**
    for cada célula (i, j) na matriz A **do**
        Atualize $A[i, j]$ usando os valores dos vizinhos
        $A[i, j] = \frac{A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1]}{4}$
    end for
end for

utilizam um padrão fixo de seleção dos pontos vizinhos, denominado *stencil*, e aplicam uma função de transição para modificar os valores do arranjo [Cattaneo et al. 2015]. Estes padrões podem ter diferentes formatos, de acordo com as características da aplicação.

Como podemos verificar nas Figuras 1 e 2 os padrões *Stencil* podem possuir diferentes formatos, variando de acordo com a especificação da aplicação. Algumas maneiras de estruturar o laço de aplicação do Padrão Stencil são as apresentadas nos algoritmos 1 e 2. No Algoritmo 1, vemos que a paralelização não pode ser realizada de modo simplificado, uma vez que o laço possui uma dependência de dados. O laço do Algoritmo 2 não possui dependências e pode ser paralelizado de maneira simples, já que a malha é atualizada primeiramente em um *buffer T*. O padrão de escolha dos vizinhos de 1 pode ser encontrado no Algoritmo 1 e o padrão de escolha dos elementos vizinhos de 2 pode ser encontrado em Algoritmo 2.

2.3. Trabalhos Relacionados

Diversos estudos investigaram a capacidade de modelos generativos em produzir códigos paralelos. Por exemplo, [Godoy et al. 2023] examinou o modelo Codex na paralelização de algoritmos como AXPY, GEMV, GEMM, SpMV, e Stencil Jacobi em Fortran, Python, e Julia. Embora ambos os trabalhos comparem a resposta do modelo ao *prompt* de paralelização, Godoy et al. usaram uma métrica de proficiência baseada nas dez primeiras sugestões do Codex. Concluíram que o Codex apresenta melhores resultados em linguagens populares, corroborando com nossos achados de que prompts detalhados melhoram a qualidade do código gerado.

Outro estudo relevante é de [Valero-Lara et al. 2023], que comparou a acurácia dos modelos Llama-2 e GPT-3 usando GitHub Copilot para gerar aplicações de alto de-

Algorithm 2 Algoritmo Stencil 1D

Inicialize a matriz A com valores iniciais
for cada iteração n de 1 até N **do**
    Inicialize um array temporário T com os mesmos valores de A
    for cada elemento i no array A **do**
        Compute o novo valor de $A[i]$ considerando seus vizinhos
        $T[i] = \frac{A[i-1]+A[i]+A[i+1]}{3}$
    end for
    Copie os valores do array T de volta para o array A
    for cada elemento i no array A **do**
        $A[i] = T[i]$
    end for
end for

Tabela 1. Características das Aplicações Utilizadas

Aplicação	*S1D*	*S2D_v1*	*S2D_v2*	*S2D_v3*	*S3D*	*Dep_S1D*	*Dep_S2D_v1*	*Dep_S2D_v2*
Número de elementos por dimensão	10000000	8000	8000	40000	256	2000000	40000	8000
Dimensões	1	2	2	2	3	1	2	2

sempenho em C++, Fortran, Python, e Julia. Concluíram que, embora o GPT-3 produza códigos mais confiáveis, o Llama-2 pode gerar soluções mais otimizadas quando acerta. Em [Buscemi 2023], a produção de códigos do GPT-3.5 foi avaliada em diversas áreas da Ciência da Computação e dez linguagens de programação. Os resultados indicaram que a corretude dos códigos varia conforme a linguagem, com melhor desempenho em linguagens de alto nível, alinhando-se com nossos resultados ao considerar a robustez da linguagem na qualidade do código. Por fim, [Kalender et al. 2014] propôs um método automatizado para paralelizar aplicações recursivas através de análise estática. Embora eficaz em gerar paralelização, o desempenho depende da complexidade e das características do código recursivo, evidenciando a importância do contexto específico da aplicação na eficácia da paralelização, um ponto também relevante para nosso estudo.

De maneira similar a [Buscemi 2023], neste artigo analisamos a produção de código por ferramentas de inteligência artificial, porém focamos especificamente na área de programação de alto desempenho. Embora análises como [Kalender et al. 2014], [Valero-Lara et al. 2023] e [Godoy et al. 2023] abordem a produção de código paralelo automática, nenhuma dessas análises focou em aplicações com o padrão Stencil e analisou os detalhes dos erros presentes nos códigos resultantes, como fizemos neste artigo. Ademais, entre os trabalhos que utilizam LLMs, nenhum analisou o ganho de desempenho pelos códigos produzidos pelas inteligências artificiais.

3. Metodologia

3.1. Aplicações

Oito aplicações que exploram o padrão de computação *Stencil* com diferentes características de tamanho de entrada e dimensões da estrutura de dados (e.g., 1D, 2D e 3D) foram utilizadas, conforme mostrado na Tabela 1. Cada uma das aplicações é destacada na Figura 3, a qual destaca o trecho de código relacionado ao padrão *Stencil*, que será utilizado para paralelização. Importante destacar que as aplicações *Dep_S1D*, *Dep_S2D_v1* e *Dep_S2D_v2*possuem dependência de dados entre threads dentro dos laços de iteração do padrão *Stencil*. As aplicações foram escritas pelos próprios autores em linguagem C, de maneira sequencial, seguindo direções fornecidas por [Eijkhout 2010], [Schmidt et al. 2017] e [Sterling et al. 2017].

As versões paralelas geradas pelos modelos *GPT-3.5 turbo* e *GPT-4o* utilizam diretivas paralelas da interface de programação paralela OpenMP. Assim, cada aplicação sequencial foi submetida através da plataforma OpenAI API[1] para paralelização. A escolha da plataforma se deu por sua simplicidade e facilidade de acesso à modelos previamente treinados em grandes quantidades de dados quando comparada a outras interfaces, como por exemplo, a oferecida pela plataforma Hugging Face [2]. Para a

[1]A plataforma pode ser acessada através do seguinte link: https://openai.com/index/openai-api

[2]A plataforma pode ser acessada através do seguinte link: `https://huggingface.co/`

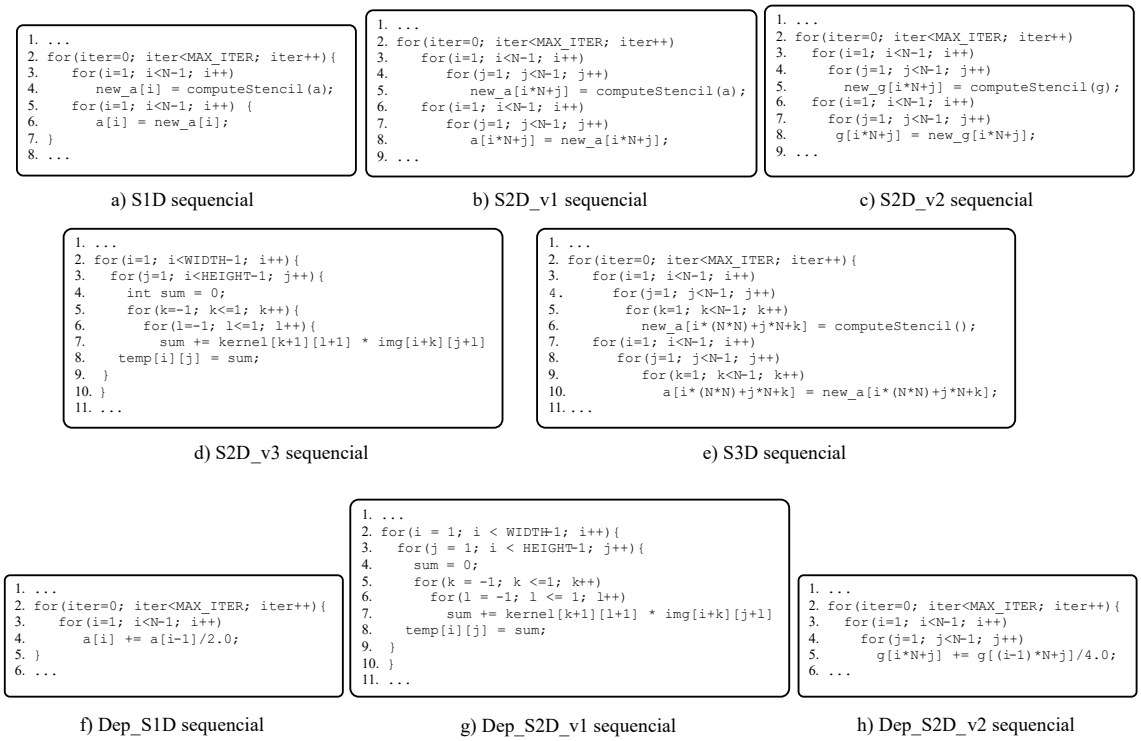


Figura 3. Snapshot das aplicações utilizadas

geração do código paralelo em cada modelo, o seguinte *prompt* de execução foi utilizado: `Parallelize this code using OpenMP : + <Code>`. Após a geração do código paralelo, cada aplicação foi compilada utilizando o compilador GCC 12.2.0 com a *flag* de otimização `-O3` e *flag* de compilação `fopenmp` para habilitar a geração de código paralelo por conta do OpenMP.

3.2. Ambiente de Execução

Cada aplicação gerada na etapa anterior da metodologia (e.g., sequencial e paralelas) foram executadas em três arquiteturas *multicore*: ***hype***, uma máquina bi-processada com Intel Xeon E5-2650 v3 Haswell com 40 núcleos disponíveis através da tecnologia *hyperthread* (20 físicos + 20 lógicos), com frequência de operação de 2.3GHz e 128GB de memória RAM; ***tupi***, uma máquina Intel i9-14900KF com 32 núcleos disponíveis, com frequência de operação base de 3.2GHz e 64GB de memógia RAM; e ***phoenix***, uma máquina bi-processada Intel Xeon Gold 5317, com 48 núcleos disponíveis(24 + 24) com frequência de operação de 3.0GHz e 128GB de memória RAM. Nas três máquinas, o sistema operacional utilizado foi Linux Debian 12 [3].

As aplicações paralelas geradas pelos dois modelos GPTs foram configuradas para explorar o paralelismo com o número de *threads* igual o número de núcleos disponíveis na arquitetura: 40 na máquina *hype*, 32 na *tupi* e 48 na *phoenix*. Para configurar o número de *threads*, foi utilizado a variável de ambiente `OMP_NUM_THREADS=N`, onde *N* é o número de *threads*. Já as configurações de mapeamento de dados e alocação de *threads* foram utilizadas as padrões do sistema operacional.

[3]Alguns experimentos deste trabalho utilizaram os recursos da infraestrutura PCAD, http://gppd-hpc.inf.ufrgs.br, no INF/UFRGS.

Tabela 2. Comparação dos códigos gerados pelos modelos GPT com a solução *gold*

Aplicação	GPT-3.5 turbo	GPT-4o
S1D	Igual	Adição da cláusula `shared`
S2D_v1	Igual	Adição da cláusula `collapse(2)`
S2D_v2	Igual	Adição da cláusula `collapse(2)`
S2D_v3	Igual	Adição da cláusula `collapse(2)`
S3D	Igual	Adição da cláusula `collapse(3)`
Dep_S1D	Não tratou dependência de dados	Não tratou dependência de dados
Dep_S2D_v1	Não tratou dependência de dados	Não tratou dependência de dados
Dep_S2D_v2	Não tratou dependência de dados	Não tratou dependência de dados

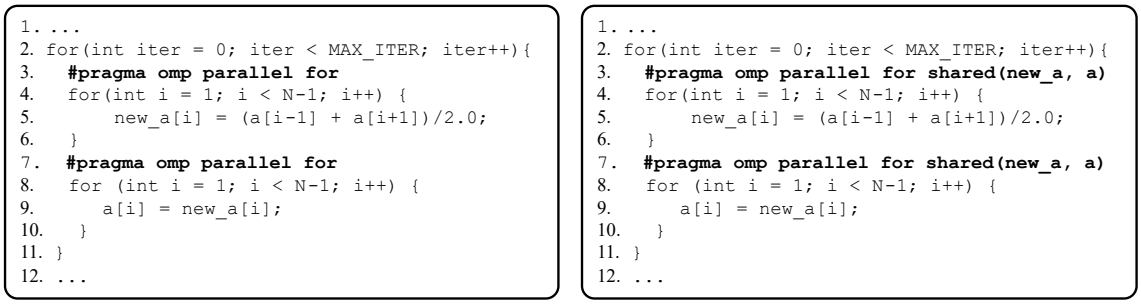
4. Resultados

4.1. Análise dos Algoritmos Paralelos Gerados

Inicialmente são apresentados e discutidos os resultados da etapa de geração do código paralelo através de cada modelo (*GPT-3.5 turbo* e *GPT-4o*). O objetivo desta análise é verificar a validade dos códigos gerados em termos de compilação, sem avaliar ainda a correção dos resultados da execução. A ênfase está na capacidade dos códigos de serem compilados corretamente, resultando em arquivos binários válidos. Importante destacar que, neste estágio, a análise focou exclusivamente na integridade sintática, léxica e semântica dos códigos, sem identificar erros em nenhum desses aspectos. Após a análise, constatou-se que todos os códigos paralelos gerados foram compilados com sucesso pelo compilador utilizado, confirmando sua conformidade com os requisitos de compilação.

A partir deste ponto, iniciamos a análise da corretude dos códigos gerados, comparando-os com uma solução de referência desenvolvida pelos autores, conforme as diretivas especificadas em [Eijkhout 2010], [Schmidt et al. 2017] e [Sterling et al. 2017]. O objetivo desta fase da análise é determinar a similaridade entre o código paralelo gerado e o código *gold*, a qual contém a implementação ideal fornecida pela suíte de aplicações utilizada, avaliando se o código gerado segue fielmente as diretivas e padrões estabelecidos. A comparação é realizada com base em critérios que incluem a precisão das diretivas paralelas inseridas, a integridade funcional e a conformidade com as especificações do código *gold*.

A Tabela 2 destaca a comparação entre os códigos gerados através de cada modelo GPT com a solução *gold*. A análise revelou que, embora ambos os modelos tenham gerado códigos compiláveis, nenhum deles foi capaz de produzir um código paralelo correto para todas as aplicações sequenciais. O modelo *GPT-3.5 Turbo* gerou código idêntico à solução *gold* em todos os casos em que não havia dependência de dados entre as *threads* na região paralela, o que sugere que o modelo lida bem com cenários de paralelismo independente em aplicações com padrão *Stencil*. Por outro lado, o modelo *GPT-4* não só replicou corretamente as diretivas paralelas (`pragma omp parallel for`), mas também introduziu cláusulas adicionais para o controle de compartilhamento de dados. No caso da aplicação *S1D*, o *GPT-4o* incluiu a cláusula `shared` para gerenciar o compartilhamento de variáveis entre *threads*, enquanto nas versões *S2D* e *S3D*, o modelo utilizou a cláusula `collapse` para otimizar a distribuição da carga de trabalho.



```
1. ...
2. for(int iter = 0; iter < MAX_ITER; iter++){
3.   #pragma omp parallel for
4.   for(int i = 1; i < N-1; i++) {
5.        new_a[i] = (a[i-1] + a[i+1])/2.0;
6.   }
7.   #pragma omp parallel for
8.   for (int i = 1; i < N-1; i++) {
9.      a[i] = new_a[i];
10.   }
11. }
12. ...
```

a) S1D gerado por GPT-3.5 Turbo

```
1. ...
2. for(int iter = 0; iter < MAX_ITER; iter++){
3.   #pragma omp parallel for shared(new_a, a)
4.   for(int i = 1; i < N-1; i++) {
5.        new_a[i] = (a[i-1] + a[i+1])/2.0;
6.   }
7.   #pragma omp parallel for shared(new_a, a)
8.   for (int i = 1; i < N-1; i++) {
9.      a[i] = new_a[i];
10.   }
11. }
12. ...
```

b) S1D gerado por GPT-4o

Figura 4. Códigos gerados para a aplicação *S1D*

A abordagem mais avançada do *GPT-4o*, que introduz as cláusulas `shared` e `collapse`, pode ser entendida em como os modelos de linguagem de grande escala aprendem e processam informações. Isto é, o *GPT-4o* foi treinado com um conjunto de dados mais abrangente e variado, o que lhe permite identificar e aplicar padrões que podem ser necessários para garantir a eficiência e corretude em cenários mais complexos de programação paralela. A inclusão destas cláusulas sugere que o *GPT-4o* foi capaz de reconhecer potenciais oportunidades de otimização na distribuição da carga de trabalho. No entanto, a complexidade adicional introduzida pela cláusula `collapse` pode não ser benéfica em todos os contextos, especialmente onde a quantidade de iterações é suficientemente grande para cada conjunto de laços, podendo levar a uma perda de desempenho na aplicação paralela. Também, a tendência do *GPT-4o* em gerar código com tais cláusulas pode ser vista como uma tentativa de generalizar a solução para uma gama maior de cenários paralelos, evidenciando a capacidade do *GPT-4o* de aplicar técnicas de otimização conhecidas na literatura.

Como exemplo, a Figura 4 ilustra as estratégias de paralelização da aplicação S1D (código sequencial mostrado na Figura 3). Os laços paralelizáveis trivialmente são os da linha 3 e 5, uma vez que não existe dependência de dados interno ao laço. Assim, o código gerado pelo modelo *GPT-3.5 turbo*, ilustrado na Figura 4a, contempla a adição das diretivas `#pragma omp parallel for` antes das estruturas de repetição nas linhas 3 e 7, sendo similar à versão *gold*. Por outro lado, a versão paralela gerada pelo modelo *GPT-4o* incluiu a cláusula `shared` para os dois vetores em adição à diretiva de paralelização.

No entanto, nenhum dos modelos foi capaz de identificar corretamente a dependência de dados entre *threads* dentro de uma região paralela nas aplicações *Dep_S1D*, *Dep_S2D_v1* e *Dep_S2D_v2*. Nestes cenários, o código gerado, embora compilável, ficou incorreto. Esta incapacidade pode ser atribuída a limitações intrínsecas à forma como esses modelos de linguagem processam e geram código. Os modelos da família GPT, apesar de serem avançados em compreensão e geração de linguagem natural, não possuem uma compreensão profunda e contextual de conceitos específicos de programação, como dependência de dados em computação paralela. Assim, esses modelos podem falhar em capturar todas as informações necessárias para assegurar a corretude do código gerado. Esta limitação, em parte, pode ser atribuída ao fato de que o treinamento dos modelos GPT não envolve a execução e verificação de código, mas sim a imitação de padrões textuais, o que pode ser insuficiente para resolver corretamente questões específicas como as dependências de dados em ambientes paralelos.

```
1. ...
2. for(i = 1; i < WIDTH-1; i++){
3.   for(j = 1; i < HEIGHT-1; j++){
4.     sum = 0;
5.     #pragma omp parallel for private(k,l,sum)
6.     for(k = -1; k <=1; k++){
7.       #pragma omp parallel for private(l,sum)
8.       for(l = -1; l <= 1; l++){
9.         sum += kernel[k+1][l+1] * img[i+k][j+l]
10.       }
11.     }
12.    temp[i][j] = sum;
13.  }
14. }
```

a) Dep_S2D_v1 gerado por GPT-3.5 Turbo

```
1. ...
2. for(i = 1; i < WIDTH-1; i++){
3.   for(j = 1; i < HEIGHT-1; j++){
4.     sum = 0;
5.     #pragma omp parallel for collapse(2)
6.     for(k = -1; k <=1; k++){
7.       for(l = -1; l <= 1; l++){
8.         sum += kernel[k+1][l+1] * img[i+k][j+l]
9.       }
10.     }
11.    temp[i][j] = sum;
12.  }
13. }
```

b) Dep_S2D_v1 gerado por GPT-4o

Figura 5. Códigos gerados para a aplicação Dep_S2D_v1

Um exemplo deste cenário é destacado na Figura 5, que ilustra as versões paralelas geradas para a aplicação *Dep_S2D_v1* utilizando os modelos GPT e a versão *gold*, com a paralelização indicada pelos autores em [Eijkhout 2010], [Schmidt et al. 2017] e [Sterling et al. 2017]. Esta aplicação é um pouco mais complexa do que a destacada na Figura 4, pois conta com 4 laços de repetição, em que, os dois laços mais externos percorrem uma imagem (representada como matriz 2D), e os dois laços internos, nas linhas 5 e 6 (Figura 3g), realizam uma computação interna entre os elementos da matriz, a qual será atualizada na linha 8 em uma matriz de saída. Portanto, a paralelização indicada contém apenas a adição da diretiva `#pragma omp parallel for collapse(2) private(j)` antes do laço mais externo. Esta paralelização está correta pois divide o processamento da matriz de entrada entre as diferentes *threads*. Por outro lado, as versões geradas pelos modelos *GPT-3.5 turbo* e *GPT-4o* exploraram o paralelismo nos laços internos da aplicação (Figuras 5b e 5c). Neste caso, nenhuma delas se atentou ao fato da dependência de dados na variável `sum`, que neste cenário, será atualizada por várias *threads* de modo simultâneo, gerando resultado incorreto.

Nos casos em que os modelos geraram código paralelo incorreto, foi adicionada uma nova mensagem ao *prompt* de execução, indicando a presença de dependências de dados entre as *threads* e a necessidade de tratamento dessas dependências. O *prompt* atualizado fornecido aos modelos para as aplicações *Dep_S1D*, *Dep_S2D_v1* e *Dep_S2D_v2* foi: `The provided code was not parallelized correctly due to data dependency. Hence, provide a new parallel version in openmp that considers the data dependency among threads.`
Após esta segunda rodada de geração de código, observou-se que apenas o modelo *GPT-3.5 Turbo* foi capaz de gerar o código paralelo correto para as aplicações *Dep_S1D* e *Dep_S2D_v2*. O *GPT-4o*, por outro lado, não conseguiu corrigir adequadamente as dependências de dados, resultado em códigos paralelos ainda incorretos. Isso pode ser atribuído ao fato de que, apesar de o *GPT-4o* ter mostrado uma maior sofisticação na adição de cláusulas de controle, ele ainda pode ter dificuldades em compreender e implementar soluções corretas para problemas de dependências de dados em paralelos.

Por fim, para a análise de desempenho que será realizada na próxima Seção, serão considerados os códigos gerados corretamente por cada modelo GPT. Nos casos em que não foi possível obter códigos paralelos corretos, a versão sequencial será utilizada para analisar o desempenho, como é o caso da aplicação *Dep_S2D_v1* para ambos modelos, e

Figura 6. Aceleração de desempenho comparado à versão sequencial para cada aplicação em cada arquitetura alvo

as aplicações *Dep_S2D_v1* e *Dep_S2D_v2* para o modelo GPT-4o.

4.2. Análise de Desempenho dos Algoritmos Paralelos Gerados

A Figura 6 apresenta os resultados de ganho de desempenho das versões paralelas geradas pelo *GPT-3.5 turbo* e *GPT-4o*, comparado à versão sequencial de cada aplicação. Portanto, valores acima de 1.0 significam que as versões geradas aceleraram os códigos sequenciais. Os resultados são mostrados para as três máquinas alvo: *hype*, *tupi* e *phoenix*. O último conjunto de barras de cada figura contém a média geométrica dos resultados obtidos em cada aplicação.

Na arquitetura *Hype*, o *GPT-3.5 turbo* apresentou melhor desempenho em várias aplicações, especialmente em *S2D_v3* e *S3D*, onde obteve as maiores acelerações comparadas ao *GPT-4o* (considerando apenas as versões de código que ambos foram capazes de gerar). Isso sugere que a adição das cláusulas *shared* e *collapse* adicionaram um sobrecusto extra para o gerenciamento do paralelismo e distribuição da carga de trabalho, o qual não foi explorado nesta arquitetura. Também, para as aplicações com dependência de dados (*Dep_S1D* e *Dep_S2D_v2*, a versão paralela gerada pelo *GPT-3.5 turbo* foi capaz de melhorar o desempenho comparado à versão sequencial, o que significa que a adição de código para tratar a dependência valeu a pena em termos de desempenho. De um modo geral, considerando a média de todas as aplicações nesta arquitetura, ambos os modelos melhoraram o desempenho da versão sequencial, porém, *GPT-3.5 turbo* obteve um desempenho 44.2% superior ao *GPT-4o*.

Para as demais arquiteturas, o comportamento segue muito similar. Isto é, para as aplicações mais simples, incluindo *S1D* e *S2D_v1*, os dois modelos geraram códigos paralelos com desempenho similar. Porém, para aplicações um pouco mais complexas, o *GPT-3.5 turbo* foi capaz de atingir melhor desempenho. É importante destacar que para as

arquiteturas *tupi* e *phoenix*, a tentativa do modelo *GPT-3.5 turbo* em gerar código paralelo para a aplicação com dependência de dados *Dep_S2D_v2* resultou em perda de desempenho comparado à versão sequencial. Esta razão pode estar ligado, principalmente, pela quantidade de instruções a mais que devem ser executadas pelas *threads* para garantir a corretude dos resultados. Por fim, ao comparar a média geométrica das aplicações em cada uma destas arquiteturas, *tupi* e *phoenix*, o modelo *GPT-3.5 turbo* foi apenas 1.2% e 5.8% melhor que o *GPT-4o*, respectivamente.

O desempenho superior do *GPT-3.5 turbo* em relação ao *GPT-4o* pode ser explicado pela diferença na aplicação ds cláusula *collapse*. Utilizada pelo *GPT-4o*, combina múltiplos laços aninhados em um único, visando melhorar a distribuição do paralelismo [Lorenzon and Beck Filho 2019]. No entanto, em aplicações de *Stencil* com grande número de iterações, como as analisadas, essa abordagem introduziu um sobrecusto significativo no gerenciamento de paralelismo, aumentando a latência e diminuindo a eficiência devido à complexidade do acesso à memória. Por outro lado, o *GPT-3.5 turbo* evitou essa cláusula, resultando em códigos paralelos mais simples e com menor sobrecarga de sincronização. Sem o sobrecusto associado ao *collapse*, o *GPT-3.5 turbo* foi capaz de distribuir o trabalho de forma mais eficiente, levando a um desempenho superior em diversas aplicações.

5. Conclusões e trabalhos futuros

A geração automática de código paralelo, especialmente por meio de modelos de linguagem como os da família GPT, representa um avanço significativo no campo da computação paralela, oferecendo um potencial para otimizar o desenvolvimento de aplicações complexas que utilizam Computação *Stencil*. Através do uso de oito aplicações e de três arquiteturas multicore com diferentes números de núcleos computacionais, a análise realizada neste trabalho demonstra que, embora ambas as versões do GPT tenham produzido códigos paralelos que superaram os desempenhos das versões sequenciais na maioria das aplicações, o *GPT-4o* não apresentou um desempenho superior geral em comparação com o *GPT-3.5 turbo*, apesar de sua capacidade de gerar código mais genérico. Além disso, a dificuldade em lidar com dependências de dados revela uma limitação significativa que ainda precisa ser superada por esses modelos, indicando que melhorias na identificação e tratamento dessas dependências são necessárias para a eficácia completa da automação do código paralelo. De um modo geral, mostrou-se que o *GPT-3.5 turbo* foi capaz de gerar códigos paralelos com desempenho 15% superior aos do *GPT-4o*, quando considerando a média geral das aplicações e arquiteturas utilizadas. Como trabalhos futuros, pretendemos explorar a integração de técnicas adicionais de aprendizado profundo para melhorar o reconhecimento de padrões de dependência de dados, além de investigar abordagens híbridas que combinem a geração automática de código com otimizações manuais para atingir um desempenho ideal.

Agradecimentos

Os autores gostariam de agradecer o apoio financeiro das instituições de fomento CNPq, Capes 01 e FAPERGS.

Referências

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Buscemi, A. (2023). A comparative study of code generation using chatgpt 3.5 across 10 programming languages.

Cattaneo, R., Natale, G., Sicignano, C., Sciuto, D., and Santambrogio, M. D. (2015). On how to accelerate iterative stencil loops: a scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–26.

Cesare, V., Colonnelli, I., and Aldinucci, M. (2020). Practical parallelization of scientific applications. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 376–384. IEEE.

Eijkhout, V. (2010). *Introduction to high performance scientific computing*. Lulu. com.

Godoy, W., Valero-Lara, P., Teranishi, K., Balaprakash, P., and Vetter, J. (2023). Evaluation of openai codex for hpc parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ICPP-W 2023. ACM.

Kalender, M. E., Mergenci, C., and Ozturk, O. (2014). Autopar: An automatic parallelization tool for recursive calls. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 159–165. IEEE.

Lorenzon, A. F. and Beck Filho, A. C. S. (2019). *Parallel computing hits the power wall: principles, challenges, and a survey of solutions*. Springer Nature.

Marques, S. M. V., Serpa, M. S., Muñoz, A. N., Rossi, F. D., Luizelli, M. C., Navaux, P. O., Beck, A. C. S., and Lorenzon, A. F. (2022). Optimizing the edp of openmp applications via concurrency throttling and frequency boosting. *Journal of Systems Architecture*, 123:102379.

Navaux, P. O. A., Lorenzon, A. F., and da Silva Serpa, M. (2023). Challenges in high-performance computing. *Journal of the Brazilian Computer Society*, 29(1):51–62.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Schmidt, B., Gonzalez-Martinez, J., Hundt, C., and Schlarb, M. (2017). *Parallel programming: concepts and practice*. Morgan Kaufmann.

Sterling, T., Brodowicz, M., and Anderson, M. (2017). *High performance computing: modern systems and practices*. Morgan Kaufmann.

Valero-Lara, P., Huante, A., Lail, M. A., Godoy, W. F., Teranishi, K., Balaprakash, P., and Vetter, J. S. (2023). Comparing llama-2 and gpt-3 llms for hpc kernels generation.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.