Explorando o Potencial e Limitações do ChatGPT para Paralelização de Códigos:
Uma Análise Empírica de Desempenho e Eficiência

Emerson Santos Barbosa

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa - PB

2023

Emerson Santos Barbosa

Explorando o Potencial e Limitações do ChatGPT para Paralelização de Códigos

Uma Análise Empírica de Desempenho e Eficiência

Monografia apresentada ao curso Ciência da Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Thaís Gaudencio do Rêgo

Novembro de 2023

Catalogação na publicação
Seção de Catalogação e Classificação

B238e Barbosa, Emerson Santos.
Explorando o potencial e limitacões do chatGPT para paralelização de códigos: uma análise empírica de desempenho e eficiência / Emerson Santos Barbosa. - João Pessoa, 2023.
84 f.

Orientação: Thaís Gaudencio do Rêgo.
TCC (Graduação) - UFPB/CI.

1. ChatGPT. 2. Paralelização de códigos. 3. Inteligência artificial em programação. 4. Análise empírica. 5. Desempenho de software. I. Rêgo, Thaís Gaudencio do. II. Título.

UFPB/CI CDU 004.8

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado ***Explorando o Potencial e Limitações do ChatGPT para Paralelização de Códigos*** de autoria de Emerson Santos Barbosa, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Thaís Gaudencio do Rêgo
Universidade Federal da Paraíba - UFPB

Prof. Dr. Yuri De Almeida Malheiros Barbosa
Universidade Federal da Paraíba - UFPB

Prof. Dr. Marcelo Iury de Sousa Oliveira
Universidade Federal da Paraíba - UFPB

João Pessoa, 16 de novembro de 2023

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

**** A epígrafe é opcional ****

****A dedicatória é opcional****

AGRADECIMENTOS

Minha vida nunca foi difícil, claro que temos momentos em que as coisas apertam um pouco, mas sorte a minha que sempre tive pessoas ao meu lado que esforçaram-se ao máximo para que tudo fluísse a meu favor e jamais me cobraram nada por isso. Se não fosse por essas pessoas, certamente eu não conseguiria chegar até aqui.

Gostaria de agradecer primeiramente a Deus por tudo que tem feito em minha vida e por me permitir atingir mais esse objetivo.

Agradeço imensamente a minha vó e meu avô, dona Maria do Socorro e Sr Edinaldo, que além de me criarem, no início da minha graduação, quando eu morava sozinho perto do centro de informática, à 60km da minha cidade, todo final de semana me levava comida, lençóis limpos, e frutas. Apesar das dificuldades jamais me deixaram faltar nada. Se não fosse por eles, nada do que sou hoje seria possível, jamais esquecerei disso.

Agradeço a minha mãe por confiar em mim, por me incentivar e sempre estar ao meu lado independente das circunstâncias.

Agradeço a minha noiva que está comigo desde o inicio, que me deu forças quando precisei, que foi meu porto seguro quando já não tinha mais chão, hoje começamos a colher os frutos de nossos esforços.

Agradeço apenas por essas palavras ao meu pai, que já se foi, e que sempre me apoiou naquilo que eu queria seguir.

Agradeço aos amigos que conheci nessa jornada acadêmica, que me tirou altas risadas e aliviaram momentos tensos, e que compartilharam bastante conhecimentos.

Por fim, e não menos importante, agradeço a minha orientadora Profª Thaís Gaudencio do Rêgo, excelente profissional, pessoa compreensiva e humana, obrigado pelos ensinamentos, paciência e dicas valiosas.

RESUMO

A paralelização é uma estratégia bem conhecida para otimizar algoritmos, mas sua implementação eficaz é repleta de desafios. Neste contexto, o ChatGPT, um modelo de linguagem treinado em vastos conjuntos de dados, é investigado como uma ferramenta potencial para a programação de alto desempenho. Este trabalho se dedica a explorar as capacidades e limitações do ChatGPT na paralelização de códigos, uma área importante para melhorar o desempenho computacional em face da crescente demanda por eficiência. Através de um estudo de caso, experimentos práticos e avaliações quantitativas, buscamos não apenas destacar os benefícios e desafios da integração do ChatGPT na paralelização, mas também estabelecer uma metodologia para pesquisas futuras na área. Os experimentos foram conduzidos focando em dois problemas. Para cada problema, foram realizados testes comparativos entre implementações sequenciais e paralelas, avaliando o desempenho e a eficácia das soluções geradas pelo ChatGPT. Os resultados indicam que, enquanto o ChatGPT pode gerar códigos paralelos eficientes e otimizados, a qualidade da paralelização depende significativamente da complexidade do problema e das instruções fornecidas ao modelo. Em casos onde as instruções eram claras e específicas, observou-se uma melhoria no desempenho. Por outro lado, instruções vagas ou genéricas resultaram em soluções menos otimizadas. Concluímos que o ChatGPT tem um potencial significativo como ferramenta de auxílio à paralelização de códigos, mas é imperativo que os usuários forneçam instruções precisas e detalhadas e tenham prévio conhecimento dos seus objetivos para aproveitar ao máximo suas capacidades. Este estudo serve como um ponto de partida para futuras investigações, visando aprimorar a integração entre inteligência artificial e programação de alto desempenho.

Palavras-chave: ChatGPT, Paralelização de Códigos, Inteligência Artificial em Programação, Análise Empírica, Desempenho de Software.

ABSTRACT

Parallelization is a well-known strategy for optimizing algorithms, yet its effective implementation is fraught with challenges. In this context, ChatGPT, a language model trained on vast datasets, is investigated as a potential tool for high-performance programming. This work is dedicated to exploring the capabilities and limitations of ChatGPT in code parallelization, a critical area for enhancing computational performance in the face of increasing demands for efficiency. Through a case study, practical experiments, and quantitative evaluations, we aim not only to highlight the benefits and challenges of integrating ChatGPT into parallelization but also to establish a methodology for future research in the field. The experiments were conducted focusing on two problems. For each problem, comparative tests between sequential and parallel implementations were carried out, assessing the performance and efficacy of the solutions generated by ChatGPT. The results indicate that while ChatGPT can generate efficient and optimized parallel codes, the quality of parallelization significantly depends on the complexity of the problem and the instructions provided to the model. Where instructions were clear and specific, an improvement in performance was observed. Conversely, vague or generic instructions resulted in less optimized solutions. We conclude that ChatGPT holds significant potential as a tool to aid in code parallelization, but it is imperative that users provide precise and detailed instructions and have prior knowledge of their objectives to fully leverage its capabilities. This study serves as a starting point for future investigations, aiming to enhance the integration between artificial intelligence and high-performance programming.

Keywords: ChatGPT, Code Parallelization, Artificial Intelligence in Programming, Empirical Analysis, Software Performance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

- **CHATGPT** – *Chat Generative Pre-Trained Transformer* (Transformador Gerativo Pré-Treinado para Chat)
- **GPU** – *Graphics Processing Unit* (Unidade de Processamento Gráfico)
- **GIL** – *Global Interpreter Lock* (Bloqueio Global do Interpretador)
- **IA** – Inteligência Artificial
- **LSTMs** – *Long Short-Term Memory* (Memória de Curto e Longo Prazo)

Conteúdo

1 INTRODUÇÃO

A demanda por desempenho e eficiência tem crescido de forma exponencial à medida que as aplicações tornam-se mais complexas e os conjuntos de dados expandem em tamanho e escopo. Desde os primórdios da computação, a paralelização tem sido vista como uma solução promissora para esses desafios. A história da computação paralela evoluiu significativamente, especialmente com o advento de arquiteturas multinúcleo e sistemas distribuídos, tornando-se uma abordagem essencial para otimizar o desempenho.

No entanto, a paralelização manual de códigos é uma tarefa intrincada. Ela exige um profundo conhecimento sobre a arquitetura do sistema, além de enfrentar desafios como garantir a corretude do código e lidar com problemas de concorrência. A paralelização eficaz pode levar a melhorias significativas no desempenho e na velocidade de execução de algoritmos, desde aplicativos até renderização e geração de imagens, resultando em uma experiência do usuário mais responsiva e tempos de processamento reduzidos.

Nesse panorama, o ChatGPT (*Chat Generative Pre-trained Transformer*) emerge como uma inovação revolucionária. Originalmente desenvolvido para tarefas de processamento de linguagem natural, o ChatGPT tem mostrado sua versatilidade em diversas áreas, desde a geração de texto até aplicações em otimização de código. Sua capacidade de capturar padrões complexos e gerar saídas coerentes o posiciona como uma ferramenta promissora para a paralelização automática de códigos.

O principal objetivo deste trabalho é avaliar a eficácia do ChatGPT na identificação de oportunidades de paralelismo em códigos existentes e na subsequente geração de códigos paralelos otimizados. Utilizaremos uma combinação de análise estática e dinâmica para avaliar o desempenho do ChatGPT nesta tarefa.

À medida que a demanda por desempenho continua a crescer, encontrar métodos automáticos e eficientes de paralelização torna-se não apenas desejável, mas essencial. Ao combinar a inteligência artificial com técnicas de programação de alto desempenho, buscamos abrir novas perspectivas para enfrentar desafios computacionais complexos.

Através da análise de estudo de caso, experimentos práticos e avaliações quantitativas,

esclareceremos os benefícios e desafios de incorporar o ChatGPT no desenvolvimento de sistemas paralelos. No restante deste trabalho, abordaremos em detalhes a teoria por trás do ChatGPT, exploraremos casos de uso específicos de sua aplicação na paralelização de códigos e analisaremos os resultados obtidos. Além disso, discutiremos as implicações mais amplas dessa abordagem e as perspectivas futuras que ela pode oferecer para a área de computação de alto desempenho.

1.1 Premissas e Hipóteses

1.1.1 Premissas

Complexidade da Paralelização Manual: A paralelização manual de códigos é uma tarefa complexa, que exige conhecimento especializado e uma compreensão profunda da arquitetura do sistema.

Capacidade do ChatGPT: ChatGPT tem demonstrado habilidades avançadas em compreender e gerar linguagem humana, bem como em identificar e replicar padrões complexos em textos. Pacheco (2011)

Necessidade de Otimização: Existe uma demanda crescente por métodos mais eficientes e automatizados de paralelização, devido ao aumento constante na necessidade de desempenho computacional. Tian et al (2023)

1.1.2 Hipóteses

H1 O ChatGPT pode ser treinado ou adaptado para identificar oportunidades de paralelismo em códigos existentes, reduzindo a necessidade de intervenção manual e expertise especializada.

H2 Utilizando o ChatGPT, é possível gerar códigos paralelos otimizados que são comparáveis, em termos de desempenho e eficiência, aos códigos paralelizados manualmente por especialistas.

H3 A integração do ChatGPT no processo de desenvolvimento pode resultar em uma redução significativa no tempo necessário para otimizar e paralelizar códigos, acelerando o ciclo de desenvolvimento de software.

1.2 Objetivos específicos

- **Analisar a Capacidade do ChatGPT:** Estudar a habilidade do ChatGPT em compreender linguagens de programação e identificar estruturas de código que possam ser otimizadas através da paralelização.
- **Identificar Oportunidades de Paralelismo:** Utilizar o ChatGPT para reconhecer segmentos de código que possam ser candidatos à paralelização, avaliando a precisão e eficácia dessas identificações.
- **Experimentar com Paralelização Assistida:** Conduzir experimentos onde o Chat GPT auxilie no processo de paralelização, fornecendo sugestões ou modificações.
- **Avaliar o Desempenho:** Comparar códigos paralelizados com a assistência do ChatGPT contra códigos paralelizados manualmente, medindo métricas como tempo de execução, eficiência e escalabilidade.
- **Documentar Limitações e Desafios:** Registrar quaisquer desafios, limitações ou obstáculos encontrados ao utilizar o ChatGPT para assistência na paralelização, fornecendo insights para futuras pesquisas e desenvolvimentos.

1.3 Estrutura da monografia

O restante desta monografia está organizado da seguinte forma:

- **Capítulo 2 - Conceitos Gerais e Revisão da Literatura:** Este capítulo apresenta uma revisão abrangente dos trabalhos relacionados à paralelização de códigos e à aplicação de modelos de linguagem, como o ChatGPT, em tarefas de programação.

- **Capítulo 3 - Metodologia:** Descreve a metodologia de pesquisa adotada, incluindo a seleção de amostras, métodos de coleta de dados e técnicas de análise.
- **Capítulo 4 - Análise dos Resultados:** Apresenta e discute os resultados obtidos, avaliando o desempenho do ChatGPT em comparação com métodos tradicionais de paralelização.
- **Capítulo 5 - Conclusão e Trabalhos Futuros:** Resume as principais descobertas do estudo, discute suas implicações e sugere direções para pesquisas futuras.
- **Referências:** Lista todas as fontes citadas ao longo da monografia.
- **Apêndices:** Contém material suplementar, como códigos-fonte, tabelas adicionais e outros recursos que complementam o estudo.

2 CONCEITOS GERAIS E REVISÃO DA LITERATURA

Neste capítulo, estabelecemos o contexto e a relevância do uso de modelos de linguagem, com foco especial no ChatGPT, na área de programação paralela. Discutiremos os conceitos fundamentais, as aplicações práticas e as pesquisas relacionadas, proporcionando uma base sólida para os capítulos subsequentes.

2.1 ChatGPT

O ChatGPT é um modelo de linguagem avançado desenvolvido pela OpenAI, baseado na arquitetura de transformadores (do inglês *transformers*). Ele é treinado em um vasto conjunto de dados textuais, aprendendo a gerar respostas em linguagem natural que são contextualmente relevantes e gramaticalmente corretas OpenAI (2023). O modelo tem sido amplamente utilizado em uma variedade de aplicações, desde assistentes virtuais até ferramentas de desenvolvimento de software e será detalhado na seção seguinte (Biswas, 2023).

2.1.1 Arquitetura e Treinamento

O ChatGPT é construído sobre a arquitetura de transformadores, uma estrutura neural que permite a captura de contextos complexos em sequências de dados, como textos. O modelo foi apresentado inicialmente em 2017 e usam mecanismos de atenção para pesar a importância relativa de diferentes palavras em uma frase, permitindo uma compreensão mais rica do contexto (Vaswani et al., 2017).

A Figura 1 apresenta uma representação visual da arquitetura do transformador, conforme apresentado por Vaswani et al. (2017).

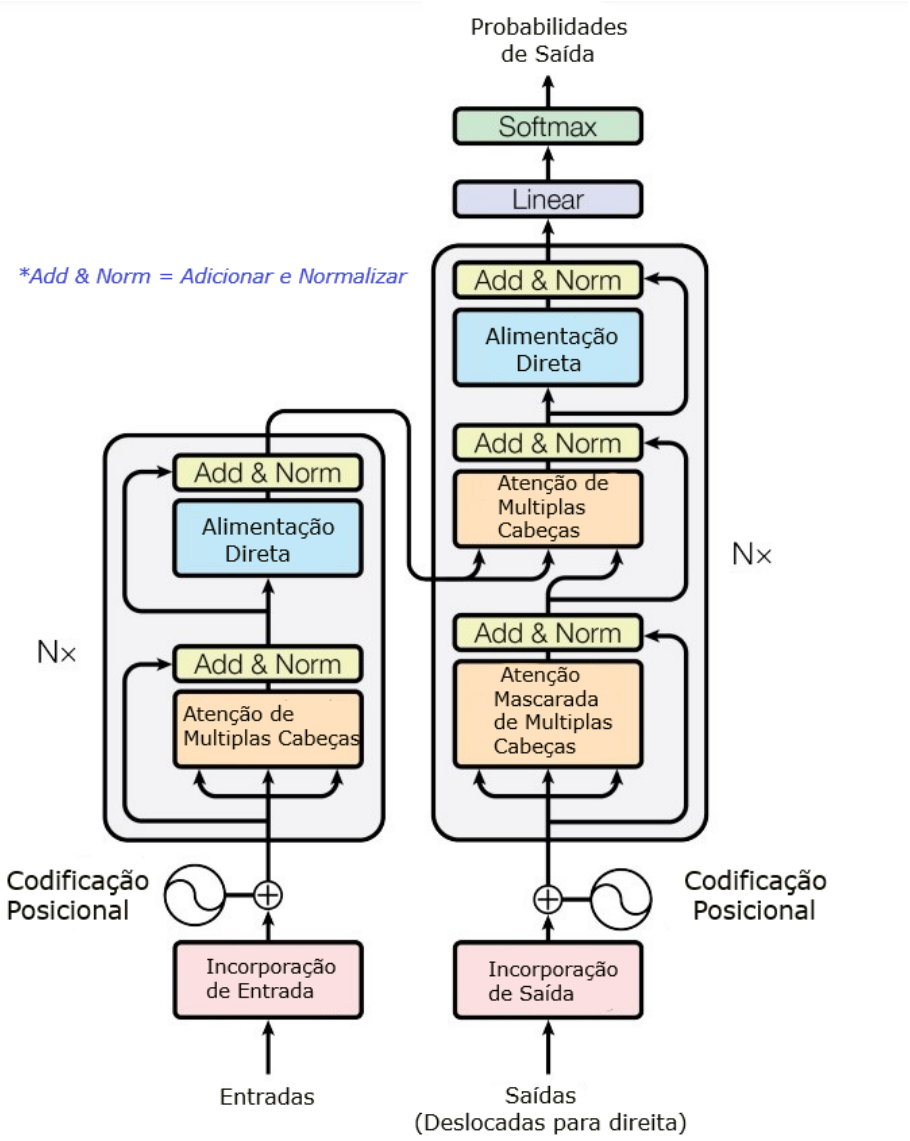

Figura 1: *Transformer* - Arquitetura do modelo. Fonte: Vaswani et al. (2017)

Os componentes da Figura 1 atuam da seguinte forma:

1. **Entradas (*Inputs*)**: Representam as sequências de texto que são submetidas ao modelo, seja para processamento, análise ou tradução.

2. **Codificação Posicional (*Positional Encoding*)**: Ao contrário de modelos recorrentes como as *Long Short-Term Memory* (LSTMs), que processam sequências levando em conta a ordem natural dos dados, os *Transformers* não têm uma noção intrínseca de ordem sequencial. Para resolver isso, uma codificação posicional é introduzida, atribuindo uma marca única de posição a cada *token*. Tokens são as unidades básicas de processamento no modelo, representando palavras ou subpartes de palavras que são analisadas pelo algoritmo, permitindo que o modelo mantenha a sequencialidade e compreenda a ordem dos termos na sequência de entrada.

3. **Atenção de Múltiplas Cabeças (*Multi-Head Attention*)**: Trata-se do mecanismo fundamental dos *transformers*. Ele permite que o modelo focalize em diferentes partes da entrada simultaneamente, identificando vários tipos de relacionamentos e contextos.

4. **Atenção Mascarada de Múltiplas Cabeças (*Masked Multi-Head Attention*)**: Funciona de maneira similar à Multi-Head Attention (Atenção de múltiplas Cabeças), mas é empregado principalmente no decodificador para assegurar que a predição de um *token* específico dependa somente dos *tokens* anteriores na sequência.

5. **Adicionar & Normalizar (*Add & Norm*)**: Estas são camadas residuais e de normalização que auxiliam na estabilidade e eficácia do treinamento do modelo.

6. ***Feed Forward***): Entre as camadas de atenção, encontram-se redes neurais *feed-forward* tradicionais, que ajudam no processamento dos dados.

7. **Nx**: Esta notação indica que a estrutura interna (como a combinação de *Add&Norm*, *Multi-Head Attention* e *Feed Forward*) é replicada por N vezes, onde N é o número de camadas do *Transformer*.

8. **Saídas (*Outputs*)**: Após todo o processamento, o modelo fornece uma série de saídas. Em aplicações de tradução, por exemplo, estas seriam sequências de texto traduzido.

O treinamento do modelo ocorre em duas etapas principais: pré-treinamento e ajuste fino (em inglês, *fine tuning*.

1. **Pré-treinamento**: Nesta fase, o modelo é treinado em grandes quantidades de texto. O objetivo é permitir que o modelo aprenda a estrutura da linguagem, a gramática, uma ampla variedade de informações factuais e, até certo ponto, habilidades de raciocínio. Este treinamento ajuda o modelo a entender o contexto e formar uma base sólida de conhecimento geral. et al. (2020).

2. **Ajuste Fino (Fine-tuning)**: Depois que o modelo é pré-treinado, ele passa pelo processo de ajuste fino, que o refina para tarefas específicas. O ajuste fino é realizado em um conjunto de dados mais específico, muitas vezes com a supervisão direta de humanos. et al. (2020). Durante o ajuste fino, exemplos de diálogo entre humanos e o modelo são usados para treinar o modelo a responder de forma mais precisa e relevante. Esse processo permite que o modelo seja adaptado para aplicações específicas e responda de maneira alinhada com as expectativas do usuário. OpenAI (2019).

2.1.2 Aplicações em Programação

O ChatGPT, desenvolvido pela OpenAI (OpenAI, 2023), é um modelo de linguagem avançado que tem sido aplicado com sucesso em diversas áreas da programação. Sua capacidade de compreender e gerar linguagem natural o torna particularmente útil para tarefas como geração de código, depuração e otimização de algoritmos. Além disso, o ChatGPT foi treinado em uma variedade de linguagens de programação, o que o torna uma ferramenta versátil para programadores de diferentes especialidades (Chen et al, 2021).

Um dos exemplos notáveis da aplicação do ChatGPT na programação é o GitHub Copilot, uma ferramenta de assistência de codificação alimentada pelo modelo GPT-3.5 da OpenAI, que ajuda os desenvolvedores a escrever código mais rapidamente, sugerindo linhas inteiras ou blocos de código (Chen et al., 2021). O *Copilot* pode gerar código para uma ampla gama de linguagens e *frameworks*, tornando-o uma ferramenta poderosa para desenvolvedores que trabalham em projetos complexos e multifacetados.

Outra aplicação do ChatGPT é na depuração de código, onde o modelo pode sugerir correções para erros comuns, ajudando os programadores a economizar tempo e reduzir a frustração durante o desenvolvimento de software (Surameery e Shakor, 2023). Por exemplo, o ChatGPT pode analisar mensagens de erro e fornecer sugestões de correção baseadas no contexto do código e na descrição do problema fornecida pelo usuário.

Além disso, o ChatGPT tem sido utilizado na otimização de algoritmos, onde pode sugerir melhorias na eficiência do código ou na estrutura de dados. Isso é particularmente útil em cenários onde o desempenho é crítico, como no processamento de grandes conjuntos de dados, que é o caso do presente estudo, ou na implementação de sistemas em tempo real (Surameery e Shakor, 2023).

Em resumo, o ChatGPT oferece um vasto potencial para melhorar a produtividade e a eficiência na programação, fornecendo assistência em tempo real que pode acelerar o desenvolvimento de software e melhorar a qualidade do código produzido, inclusive em tarefas de paralelização.

2.2 Paralelização

A paralelização é uma técnica em computação de alto desempenho, permitindo que tarefas sejam executadas simultaneamente, aproveitando múltiplos núcleos de processador ou CPUs. Isso resulta em uma execução mais rápida e eficiente de programas e algoritmos. Existem diversos tipos de paralelismo, incluindo paralelismo de dados, onde os dados são divididos e processados simultaneamente, e paralelismo de tarefas, onde diferentes tarefas são executadas em paralelo (Pacheco, 2011). A escolha do tipo de paralelismo, explicado a seguir, e a estratégia de implementação dependem da natureza do problema e do algoritmo em questão.

2.2.1 Multiprocessamento

O multiprocessamento refere-se à capacidade de um sistema de processar mais de uma tarefa ao mesmo tempo, utilizando vários processadores ou núcleos em uma única

máquina. Esse processo é viabilizado pelo uso de múltiplos processadores, ou núcleos, em uma única máquina (Pacheco, 2011). Dentro desse contexto, uma *thread* é definida como a menor sequência de instruções programadas que pode ser gerenciada de forma independente por um agendador no sistema operacional (Pacheco, 2011). Em sistemas equipados com multiprocessamento, as *threads* são particularmente úteis, pois podem ser executadas em diferentes processadores, ou núcleos, quase simultaneamente, permitindo a execução paralela e, portanto, uma melhora significativa no desempenho (Pacheco, 2011).

As *threads* gerenciam independentemente um agendador no sistema operacional. Em sistemas equipados com múltiplos núcleos de CPUs, diversas *threads* podem ser executadas simultaneamente, cada uma em seu respectivo núcleo, como mostra a Figura 2. Isso se traduz em um aumento significativo na eficiência e velocidade da execução, especialmente quando tratamos de tarefas computacionais complexas e intensivas (Pacheco, 2011).

A eficiência trazida pelo multiprocessamento é muito importante para a realização efetiva de multitarefas, detalhada na Seção 2.2.2. A capacidade de um sistema operacional de alternar entre múltiplas tarefas é grandemente aprimorada quando cada tarefa pode ser executada em um núcleo separado, reduzindo a necessidade de alternância frequente e permitindo uma verdadeira execução paralela.

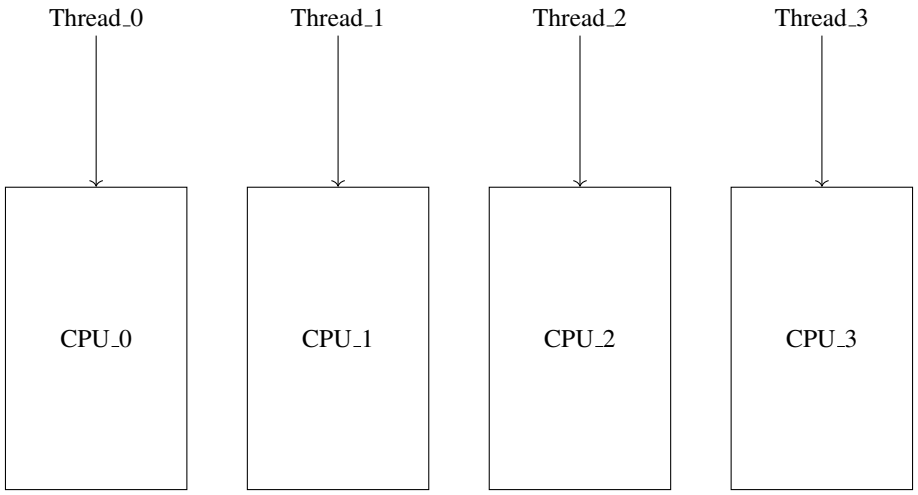

Figura 2: Representação simplificada de um sistema com multiprocessamento. Fonte: Autor (2023)

2.2.2 Multitarefas

Multitarefas é a capacidade de um sistema operacional em gerenciar e alternar entre várias tarefas rapidamente. Isso é realizado por meio da gestão de processos e suas respectivas *threads*. Na realidade, o processador pode alternar entre *threads* de diferentes processos de forma tão veloz, que cria a ilusão de execução paralela para o usuário. Pacheco (2011). A Figura 3 apresenta uma ilustração dessa alternância.

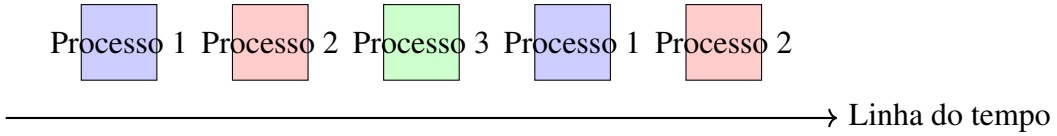

Figura 3: Representação da alternância de tarefas em sistemas multitarefas com três processos. Fonte: Autor (2023)

A Seção 2.2.3 a seguir demonstra os critérios de avaliação utilizados para quantificar e comparar aspectos como a melhoria de desempenho e a eficiência no uso de recursos computacionais. São importantes para entender a eficácia da multitarefa em sistemas operacionais modernos.

2.2.3 Critérios de Avaliação

Nesta seção, apresentamos os principais critérios de avaliação utilizados na área e nesse estudo, bem como os métodos correspondentes para cada critério. A Tabela 1 a seguir detalha essas informações.

Critério	**Método de Avaliação**
Melhoria de Desempenho	Medição e comparação do tempo de execução entre as soluções propostas e as de referência, considerando diferentes tamanhos e complexidades de dados de entrada.
Validação do Código	Comparação dos resultados produzidos pelo código paralelizado e pelo código sequencial para assegurar a corretude da implementação paralela.
Repetições e Consistência	Execução de múltiplas rodadas de testes e utilização da média dos tempos de execução para minimizar o impacto de outliers e garantir a consistência dos resultados.
Análise de Dados de Entrada	Avaliação do comportamento do código com diferentes conjuntos de dados de entrada, considerando tanto dados aleatórios quanto casos de teste específicos.
Análise Estatística	Aplicação de técnicas estatísticas, como testes de significância, para avaliar se as diferenças de desempenho são estatisticamente significativas.
Escalabilidade	Avaliação do desempenho do código à medida que o tamanho do problema aumenta, observando se os ganhos de desempenho são mantidos e como eles se comparam com a versão sequencial.
Overhead de Paralelização	Análise do overhead introduzido pela paralelização, considerando aspectos como a criação de threads/processos e a comunicação entre eles.
Continua na próxima página	

Tabela 1 – Continuação da página anterior

Critério	Método de Avaliação
Manipulação de Memória	Utilização de ferramentas e métricas específicas para monitorar o uso de memória durante a execução, avaliando a eficiência na alocação e desalocação de recursos.
Eficiência no Uso de Recursos Computacionais	Monitoramento da utilização dos núcleos durante a execução, analisando o balanceamento de carga e a paralelização efetiva das tarefas.
Facilidade de Compreensão	Revisão de código por pares e aplicação de métricas de qualidade de código relacionadas à legibilidade e manutenibilidade.
Compreensão do Problema	Análise qualitativa da correspondência entre os requisitos do problema e as funcionalidades implementadas na solução proposta.

Tabela 1: Critérios de Avaliação e Métodos Correspondentes

2.3 Trabalhos Relacionados

Durante o processo de busca de trabalhos para embasamento deste estudo, o objetivo foi identificar materiais que explorassem de maneira o uso de modelos de linguagem, em especial o ChatGPT, em contextos de programação, com um foco especial na paralelização de códigos e otimização de desempenho, temas centrais deste trabalho.

Utilizamos palavras-chave como ”ChatGPT”, ”programação paralela”, ”otimização de código”e ”assistência em programação”em bases de dados acadêmicas como IEEE Xplore, Google Scholar e PubMed. O objetivo era identificar estudos que explorassem o uso de modelos de linguagem, especialmente o ChatGPT, em contextos de programação, com um interesse particular em trabalhos que abordassem a paralelização de códigos ou otimização de desempenho. Os trabalhos analisados foram majoritariamente extraídos de repositórios acadêmicos e suas referências também serviram de inspiração para o presente

estudo.

2.3.1 O papel do ChatGPT na programação

Um dos trabalhos mais relevantes para o contexto deste estudo é o artigo "Role of ChatGPT in Computer Programming" de Biswas (2023). Este trabalho investiga as capacidades do ChatGPT em várias tarefas de programação, desde correção e conclusão de código, até geração de documentos e desenvolvimento de *chatbots*.

Biswas et al. (2023) demonstram que o ChatGPT pode ser uma ferramenta versátil no ambiente de desenvolvimento de software. O estudo aborda como o ChatGPT pode ser integrado em ambientes de programação para melhorar a produtividade do desenvolvedor. Eles também fornecem exemplos práticos que ilustram as capacidades do ChatGPT em diferentes tarefas de programação, incluindo correção de erros de sintaxe, otimização e refatoração de código, e até mesmo a geração de documentação.

O artigo conclui que o uso de ChatGPT pode melhorar a satisfação geral com os serviços de suporte e ajudar as organizações a construir uma reputação de expertise e confiabilidade. Embora o foco do artigo não seja especificamente na paralelização de códigos, suas descobertas fornecem uma base sólida para explorar o potencial do ChatGPT em tarefas mais especializadas de programação, como a paralelização.

Este trabalho serve como um ponto de partida importante para o nosso estudo, pois estabelece as capacidades gerais do ChatGPT em programação. Ele também destaca a necessidade de mais pesquisas para explorar o potencial do ChatGPT em tarefas de programação mais especializadas, uma lacuna que este estudo visa preencher.

2.3.2 Desmascarando o gigante: Avaliação de ChatGPT em Algoritmos e Estruturas de Dados

O estudo conduzido por Arefin et al. (2023b) oferece uma avaliação abrangente das capacidades de codificação do ChatGPT, focando em problemas relacionados a algoritmos e estruturas de dados em Python. Este trabalho é particularmente relevante para o nosso

estudo, pois estabelece tanto as capacidades quanto as limitações do ChatGPT em tarefas de codificação, que são fundamentais para a computação paralela.

O artigo também faz várias contribuições importantes, incluindo a avaliação da qualidade do código gerado pelo ChatGPT e um exame do potencial de memorização de dados de treinamento pelo modelo. Essas contribuições podem fornecer *insights* para a nossa avaliação da eficácia da ferramenta na paralelização de códigos. A metodologia empregada pelos autores para avaliar o ChatGPT pode servir como um guia para como podemos abordar a avaliação da eficácia do ChatGPT em tarefas de programação paralela, especialmente em termos de qualidade de código e tratamento de erros.

A abordagem metodológica de Arefin et al. (2023) é particularmente instrutiva para o presente estudo, proporcionando um caminho a ser explorado em nossa própria investigação, especialmente no que tange à qualidade de código e gestão de erros. Este foco é também uma preocupação central no trabalho de Tian et al. (2023), que examina estas questões em um contexto mais amplo e será discutido a seguir.

2.3.3 A Eficiência e Limitações do ChatGPT como Assistente de Programação

O artigo "Is ChatGPT the Ultimate Programming Assistant - How far is it?" de Tian et al (2023) oferece uma avaliação empírica abrangente das capacidades do ChatGPT em tarefas de programação. Eles utilizam várias métricas, incluindo correção de código, complexidade de tempo e taxa de reparo, para avaliar o desempenho do ChatGPT. Este trabalho é relevante para o nosso estudo, pois utiliza uma base métrica que podemos adaptar para avaliar o desempenho do ChatGPT em tarefas de paralelização de códigos.

As limitações apontadas pelo artigo, como a dificuldade do ChatGPT em generalizar para problemas novos e não vistos, servem como um alerta para o nosso estudo. Isso ressalta a necessidade de testes rigorosos e validação ao aplicar o ChatGPT em cenários de programação paralela. Além disso, as métricas utilizadas no artigo podem ser adaptadas para avaliar especificamente a eficácia do ChatGPT em paralelizar códigos, fornecendo assim uma abordagem metodológica robusta para o nosso estudo.

2.3.4 Quem Responde Melhor? Uma Análise Aprofundada das Respostas do ChatGPT e Stack Overflow para Perguntas de Engenharia de Software

O estudo realizado por Kabir et al. (2023a). conduziu uma análise aprofundada das respostas do ChatGPT para 517 perguntas do Stack Overflow relacionadas à engenharia de software. Este trabalho é de significativa importância para o nosso estudo, pois proporciona um quadro para avaliar a qualidade das respostas do ChatGPT, que pode ser adaptado para avaliar o desempenho do ChatGPT em tarefas de paralelização de código.

O artigo também revela que 52% das respostas do ChatGPT são incorretas e 77% são verbosas. Esses achados servem como um alerta para o nosso próprio projeto, enfatizando a necessidade de testes e validação rigorosos. Além disso, os resultados do estudo do usuário podem oferecer *insights* sobre por que os desenvolvedores podem preferir usar o ChatGPT, apesar de suas limitações, o que poderia ser relevante ao discutir a adoção do usuário da nossa ferramenta de paralelização.

Em resumo, os trabalhos apresentados nesta seção oferecem uma ampla visão sobre as capacidades e limitações do ChatGPT em diferentes contextos de programação. Eles fundamentam o nosso estudo, destacando tanto as potencialidades quanto os desafios associados ao uso do ChatGPT em programação. O presente estudo se diferencia dos trabalhos anteriores ao focar especificamente na aplicação do ChatGPT para a paralelização de códigos, como destacado na Tabela 2.

Nossa abordagem, baseada em experimentos práticos e avaliação quantitativa, visa fornecer uma análise objetiva das capacidades do ChatGPT em otimizar algoritmos através da paralelização. Ao preencher esta lacuna, esperamos contribuir para o entendimento de como modelos de linguagem podem ser efetivamente utilizados para melhorar o desempenho de algoritmos, proporcionando uma ferramenta para desenvolvedores e pesquisadores interessados em explorar o potencial da Inteligência Artificial em programação de alto desempenho.

Critério	**Foco Principal**	**Metodologia**	**Principais Contribuições**	**Limitações Identificadas**	**Relevância para o Presente Estudo**
Biswas et al. (2023)	Capacidades gerais do ChatGPT em programação	Análise qualitativa e exemplos práticos	Demonstração da versatilidade do ChatGPT em programação	Não foca especificamente em paralelização de códigos	Estabelece as capacidades gerais do ChatGPT em programação
Arefin et al. (2023)	Avaliação do ChatGPT em algoritmos e estruturas de dados	Avaliação da qualidade do código e exame do potencial de memorização	Avaliação das capacidades de codificação do ChatGPT	Não aborda diretamente a paralelização de códigos	Fornece insights sobre a qualidade do código gerado pelo ChatGPT
Tian (2023)	Eficiência e limitações do ChatGPT como assistente de programação	Avaliação empírica utilizando métricas específicas	Avaliação empírica das capacidades do ChatGPT em programação	Dificuldade em generalizar para problemas novos e não vistos	Oferece uma abordagem metodológica para avaliação do ChatGPT em programação
Presente Estudo	Aplicação do ChatGPT para paralelização de códigos	Experimentos práticos e avaliação quantitativa	Metodologia para avaliação do ChatGPT em paralelização de códigos	Necessidade de instruções claras e específicas para otimização	Base para exploração do ChatGPT em paralelização de códigos

Tabela 2: Comparação entre os trabalhos relacionados e o presente estudo

3 METODOLOGIA

Neste capítulo será delineada a metodologia utilizada para avaliar a capacidade do ChatGPT(Versão 3.5) em paralelizar códigos. Serão abordados problemas de diferentes níveis, com diferentes paradigmas inicialmente na linguagem Python. Caso o ChatGPT falhe na geração de código em Python para um determinado problema, foi proposta a solução na linguagem C/C++. A base teórica foi extraída essencialmente do livro "Uma Introdução à Programação Paralela"(Pacheco, 2011), assim como a extração de problemas concretos com soluções bem definidas, a fim de serem comparadas as de nossos testes.

3.1 Avaliação Experimental e Definição dos Problemas

Para a construção de soluções para os problemas extraídos foram definidas duas abordagens:

- **Solução Genérica a partir de Descrição Simplificada**: É apresentada ao ChatGPT uma descrição simplificada do problema e solicitamos uma solução sem especificar o contexto ou a linguagem de programação. O objetivo é avaliar a capacidade do ChatGPT de compreender e resolver o problema de forma genérica, sem restrições ou diretrizes adicionais.
- **Otimização com Especificações Avançadas**: Além da descrição do problema, são fornecidas ao ChatGPT informações detalhadas sobre API, bibliotecas, linguagem e hardware (como CPU e GPU). Isso nos permite avaliar a capacidade do ChatGPT de otimizar e paralelizar códigos em cenários mais restritos e específicos, maximizando a utilização dos recursos disponíveis.

3.2 Problemas Selecionados

Para avaliar as soluções propostas pelo ChatGPT, optamos por focar em dois problemas clássicos e amplamente reconhecidos no campo da programação: a multiplicação

de matriz por vetor e a ordenação de listas. Esses problemas foram escolhidos devido à sua relevância em diversas aplicações práticas e sua presença constante tanto em contextos educacionais, quanto profissionais.

Ambos os problemas, multiplicação de matriz por vetor e ordenação de listas, têm suas peculiaridades no que diz respeito à demanda de processamento.

A multiplicação de matriz por vetor, uma operação central na álgebra linear, implica na multiplicação de cada elemento da matriz por um elemento do vetor. Esta operação tem complexidade temporal $O(n^2)$ para matrizes quadradas de tamanho n.

A ordenação de listas, por sua vez, possui demanda de processamento que varia conforme o algoritmo empregado. Enquanto o *Bubble Sort* tem complexidade $O(n^2)$, algoritmos mais otimizados como o *Merge Sort* e *Quick Sort* possuem complexidade $O(n \log n)$ (Yang et al. (2011)). A ordenação também exige comparações e trocas, o que pode ser, dependendo da natureza dos dados, desafiador. Nas Seções 3.3 e 3.4, detalharemos o funcionamento de cada problema e como o ChatGPT aborda soluções, além de discutir os desafios e fatores que influenciam a paralelização e eficiência.

3.3 Multiplicação de Matriz por Vetor

A multiplicação de uma matriz por um vetor é uma operação fundamental em álgebra linear e tem aplicações em diversas áreas da ciência da computação, como gráficos computacionais, solução de sistemas lineares e otimização. Dada uma matriz A de dimensões $m \times n$ e um vetor v de dimensão n, o objetivo é calcular um novo vetor w de dimensão m, onde cada elemento w_i é calculado como o produto escalar da i-ésima linha de A com v.

Matematicamente, o elemento w_i é dado por:

$$w_i = \sum_{j=1}^{n} A_{ij} \times v_j \tag{1}$$

A natureza da operação permite que cada elemento w_i seja calculado indepen-

dentemente, tornando-a uma candidata ideal para paralelização. No entanto, a eficiência da paralelização pode ser afetada por vários fatores, como a organização dos dados na memória e a sobrecarga de comunicação entre *threads* ou processos.

Veremos posteriormente as implementações realizadas pelo ChatGPT utilizando as variações de *prompts*(entradas) propostas na Seção 3.1, além da implementação sequencial, desenvolvida manualmente, para realização dos testes comparativos.

3.3.1 Implementação sequencial - Matriz x Vetor:

Para nossa avaliação, utilizaremos uma matriz A e um vetor v com valores aleatórios. A solução sequencial servirá como referência, e compararemos seu desempenho com a solução paralela proposta pelo ChatGPT, observando não apenas a correção do resultado, mas também a eficiência e a escalabilidade da implementação paralela. O código utilizado para a implementação sequencial é apresentado na Figura 4, na linguagem Python.

```python
import numpy as np

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    for i in range(M):
        for j in range(N):
            w[i] += A[i, j] * v[j]
    return w
```

Figura 4: Algoritmo de multiplicação de matriz por vetor em Python.

A operação de multiplicação de matriz por vetor, como apresentada, é uma operação de complexidade $O(m \times n)$, onde m é o número de linhas da matriz e n é o número de colunas (que também corresponde ao tamanho do vetor). Em nossa implementação, temos

um *loop* aninhado, que percorre cada linha da matriz e, para cada linha, percorre cada elemento do vetor, realizando a multiplicação e acumulando o resultado.

O código apresentado define duas constantes, M e N, que representam as dimensões da matriz. A função `matrix_vector_mult` recebe uma matriz A e um vetor v como parâmetros. Dentro da função, inicializamos um vetor w com zeros, que armazenará o resultado da multiplicação. Os dois *loops* aninhados são responsáveis por calcular o produto escalar de cada linha da matriz com o vetor.

Uma característica relevante dessa operação é que o cálculo de cada elemento do vetor resultante w é independente dos outros. Isso significa que, teoricamente, poderíamos calcular cada, ou um conjunto, w_i em paralelo, sem interferência ou necessidade de comunicação entre as tarefas paralelas. Esta é uma propriedade desejável quando pensamos em paralelização, pois sugere que podemos obter ganhos significativos de desempenho, ao dividir o trabalho entre múltiplos núcleos ou processadores.

3.4 Ordenação por transposição Ímpar-Par

O algoritmo de ordenação por transposição ímpar-par representa uma abordagem paralelizável para a ordenação de listas, inspirando-se na estrutura básica do *Bubble Sort*, mas introduzindo melhorias significativas em termos de eficiência em ambientes paralelos.

O mesmo opera comparando e trocando elementos adjacentes em uma lista. A principal distinção reside na forma como essas operações são realizadas: o processo é dividido em duas fases distintas - uma para índices pares e outra para índices ímpares. Durante a fase par, todos os pares de elementos em posições pares são comparados e trocados, se necessário. Analogamente, a fase ímpar lida com elementos em posições ímpares. Essas duas fases são repetidas até que a lista esteja completamente ordenada. A seguir na Figura 5 temos o pseudocódigo para implementação do Algoritmo de Ordenação Ímpar-Par.

A lista armazena um conjunto de números inteiros que serão arranjados na ordem crescente. Durante a fase par, cada elemento par `arr[i-1]` é comparado ao elemento a sua direita arr[i], caso estejam fora de ordem (`arr[i-1] > arr[i]`), a troca é execu-

```
    n = length(arr)
    for phase from 0 to n-1 do
        if phase is even then
            for i from 1 to n step 2 do
                if arr[i-1] > arr[i] then
                    swap(arr[i-1], arr[i])
        else
            for i from 1 to n-1 step 2 do
                if arr[i] > arr[i+1] then
                    swap(arr[i], arr[i+1])
    return arr
```

Figura 5: Algoritmo de Ordenação por Transposição Ímpar-Par

tada. Já na fase ímpar, o elemento ímpar é comparado com o elemento à sua direita, caso estejam fora de ordem, é executada a troca. Após `N` fases, a lista estará ordenada.

Por exemplo, supondo que queremos realizar o rearranjo em ordem crescente da lista `arr = [3,5,2,4,1]`.

- **Fase 0 (Par)**: Começaríamos analisando as posições pares com seus sucessores:

```
[0, 3] > [1, 5] ?  False → [3,5,2,4,1]
[2, 2] > [3, 4] ?  False → [3,5,2,4,1]
```

- **Fase 1 (Ímpar)**: Testaríamos elementos na posição ímpar com seus sucessores:

```
[1, 5] > [2, 2] ?  True → [3,2,5,4,1]
[3, 4] > [4, 1] ?  True → [3,2,5,1,4]
```

e assim sucessivamente até a fase 4, onde teríamos a lista completamente ordenada `arr = [1,2,3,4,5]`, demonstrado na figura 6 a seguir.

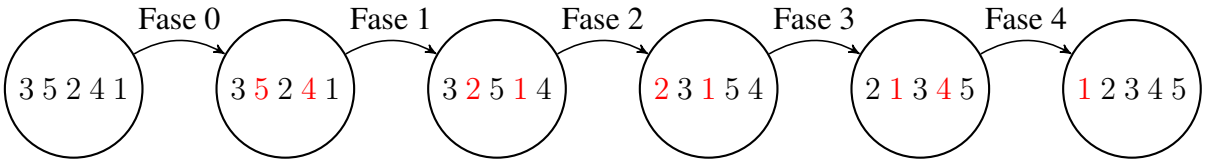

Figura 6: Simulação do algoritmo de ordenação par-ímpar, elementos destacados em vermelho serão comparados aos seus sucessores

A simplicidade do algoritmo de ordenação par-ímpar, juntamente com sua estrutura única, oferece oportunidades para paralelismo, o que pode ser particularmente benéfico em sistemas com múltiplos processadores ou núcleos.

A complexidade do algoritmo, em sua forma básica, é de $O(n^2)$, semelhante ao *Bubble Sort*, devido à necessidade de realizar várias passagens pela lista até que ela esteja completamente ordenada. No entanto, a natureza do algoritmo permite que as comparações e trocas dentro de cada fase (Par ou Ímpar) sejam realizadas simultaneamente, o que pode reduzir significativamente o tempo de execução em um ambiente paralelo.

Durante a fase par, todos os elementos em posições pares podem ser comparados e trocados (se necessário) ao mesmo tempo, e o mesmo se aplica aos elementos em posições ímpares durante a fase ímpar. Isso significa que, com recursos paralelos suficientes, cada fase pode ser completada em tempo constante.

Na Seção 4.2, discutiremos a implementação paralela do algoritmo de ordenação Ímpar-Par com abordagem simplificada e detalhada, inicialmente sem a apresentação do algoritmo sequencial.

3.5 Configuração do Ambiente de Testes

Os experimentos e testes foram conduzidos em um ambiente computacional específico para garantir a reprodutibilidade e a precisão dos resultados. A configuração do hardware e do software utilizados é detalhada a seguir:

- **Hardware:**

  - Sistema Operacional: Windows 10
  - *Desktop* com 16GB de memória **RAM**.
  - Processador *AMD Ryzen 5 2600*, composto por 6 núcleos físicos e 12 *threads* lógicas.

- **Software:**
  - Linguagem de Programação: Python, *versão 3.11.4*.
  - Editor de Código: Visual Studio Code, *versão 1.83*.

- **Bibliotecas**
  - concurrent.futures
  - Tensorflow, v2.14.

Todos os códigos-fonte, bem como os dados utilizados, estão disponíveis em Emerssu. (2023). Paralell-chatgpt-solutions [Repositório de código]. GitHub. `https://github.com/Emerssu/paralell-chatgpt-solutions` para garantir a possibilidade de validação dos experimentos realizados.

3.6 Procedimento Experimental

Os procedimentos experimentais foram estruturados visando avaliar a eficácia e eficiência dos códigos gerados pelo ChatGPT, sob diferentes cenários e configurações. A metodologia adotada é descrita a seguir:

1. **Especificação do Problema:** Os problemas foram especificados e submetidos ao ChatGPT, incluindo parâmetros e diretrizes específicas para a implementação da solução. Exemplos de diretrizes incluem a paralelização utilizando *multiprocessing*, emprego de *threads* com concorrência, OpenMP para implementações em C, entre outros.

2. **Validação do Código:** Antes da execução dos testes, os códigos gerados foram validados comparando os resultados produzidos pelo código paralelizado e pelo código sequencial para assegurar a corretude da implementação paralela.

3. **Execução dos Códigos:** Os códigos gerados foram executados variando o tamanho da entrada (por exemplo, o tamanho da matriz e do vetor em operações de multiplicação de matriz por vetor). Cada teste foi repetido 5 vezes, e os resultados foram registrados para análise posterior. As execuções ocorreram apenas com o *Visual Studio Code*, e as tarefas executando em segundo plano do sistema operacional.

4. **Análise de Dados de Entrada:** Os dados de entrada utilizados nos testes foram variados para explorar diferentes cenários e complexidades, incluindo dados totalmente aleatórios e casos de teste específicos.

5. **Avaliação de Bibliotecas e Limitações:** Foi realizada uma análise das bibliotecas nos códigos gerados, avaliando se as soluções propostas contornam eficientemente as limitações intrínsecas do sistema e da linguagem de programação utilizada.

6. **Análise Estatística dos Resultados:** Os resultados foram analisados utilizando técnicas estatísticas para avaliar se as diferenças de desempenho são estatisticamente significativas.

7. **Análise de Resultados:** Os resultados foram analisados considerando não apenas a eficiência temporal, mas também a adequação e eficácia das soluções propostas, em relação aos problemas especificados.

3.7 Critérios de avaliação utilizados

Para poder avaliar a qualidade das soluções propostas pelo modelo, utilizamos alguns dos critérios descritos na Tabela 1, dentre eles estão

- Melhoria de desempenho.
- Validação do código.

- Repetições e Consistência.
- Análise de dados de entrada.
- Compreensão do problema.

No próximo capítulo serão mostradas os códigos geradas, e também as análises de desempenho das soluções sugeridas pelo ChatGPT para os problemas Matriz x Vetor 3.3 e Ordenação Ímpar-Par 3.4, em comparação com com os códigos sequenciais, e também com as soluções disponibilizadas por Pacheco (2011) para o mesmo problema, desenvolvidas na linguagem C. Saberemos se o ChatGPT é capaz de melhorar os códigos inicialmente desenvolvidos em Python, e também se as soluções paralelizadas conseguem se aproximar do desempenho da linguagem de baixo nível.

[1]Códigos fonte e materiais adicionais para o livro *'An Introduction to Parallel Programming'* estão disponíveis no site oficial da Elsevier (Acessado em: 12 de agosto de 2023).

4 ANÁLISE DOS RESULTADOS

Nesta seção, faremos a análise dos resultados obtidos a partir das soluções propostas para os problemas abordados nas Seções 3.3 e 3.4. O objetivo principal é avaliar o desempenho, a eficiência e a precisão de cada código gerado, explorando suas vantagens e limitações.

4.1 Implementações propostas pelo ChatGPT - Matriz x Vetor

Com base na natureza paralelizável da multiplicação de matriz por vetor, e considerando as características e limitações da linguagem Python, é interessante explorar como um modelo avançado de linguagem natural, como o ChatGPT, abordaria essa tarefa. Estamos particularmente interessados em entender as estratégias de paralelização adotadas, a eficiência da solução proposta e como ela se compara à nossa implementação sequencial. A seguir, na Implementação 1 e 2, apresentaremos as soluções paralelizadas propostas pelo ChatGPT e discutiremos seus méritos e possíveis áreas de melhoria.

4.1.1 Implementação paralela com *prompt* simplificado

Abaixo na Implementação 1, solicitamos um algoritmo paralelizado ao ChatGPT apresentando como entrada um algoritmo sequencial e um comando simples, sem restrições ou orientações adicionais.

Prompt simplificado com código exemplo

Paralelize esse código

```python
import numpy as np

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    for i in range(M):
        for j in range(N):
            w[i] += A[i, j] * v[j]
    return w
```

Resposta ChatGPT

Para paralelizar o código que realiza a multiplicação de uma matriz por um vetor, você pode usar a biblioteca NumPy em conjunto com a biblioteca multiprocessing para dividir o trabalho entre vários processos. Aqui está uma versão paralelizada do seu código:

```python

import numpy as np
import multiprocessing

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult_row(i, A, v, result):
    M, N = A.shape
    result[i] = np.dot(A[i], v)

def parallel_matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
```

```
15
16          num_cores = multiprocessing.cpu_count()
17          pool = multiprocessing.Pool(processes=num_cores)
18
19          result = pool.starmap(matrix_vector_mult_row, [(i, A, v, w) for i in range(M)])
20          pool.close()
21          pool.join()
22
23          return w
24
25      # Exemplo de uso
26      A = np.random.rand(M, N)
27      v = np.random.rand(N)
28      result = parallel_matrix_vector_mult(A, v)
29
30
```

Implementação 1: Implementação Paralelizada em Python Utilizando Multiprocessing

4.1.2 Implementação paralelizada com *prompt* detalhado

Na Implementação 2 e 3, fornecemos ao ChatGPT um comando mais específico, detalhando a abordagem de paralelização desejada. Em vez de uma simples solicitação, orientamos o modelo sobre quais bibliotecas módulos e métodos utilizar, buscando uma solução mais otimizada e adaptada às nossas necessidades.

Prompt com especificações detalhadas e código exemplo

Paralelize esse código utilizando bibliotecas otimizadas e de paralelização disponíveis no python 3 dividindo as as tarefas igualmente entre processos e *threads*

```
import numpy as np

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    for i in range(M):
        for j in range(N):
            w[i] += A[i, j] * v[j]
    return w
```

Resposta ChatGPT

Para paralelizar eficientemente o código usando bibliotecas otimizadas e de paralelização disponíveis no Python 3, você pode aproveitar as bibliotecas NumPy e a função numpy.dot para as operações matriciais e a biblioteca concurrent.futures para a paralelização de threads. Aqui está o código paralelizado, distribuindo igualmente as tarefas entre processos e threads:

```

import numpy as np
import concurrent.futures

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz
NUM_THREADS = 6  # Número de threads a serem usadas

def matrix_vector_mult_row(start, end, A, v, result):
```

```
        M, N = A.shape
        for i in range(start, end):
            result[i] = np.dot(A[i], v)

    def parallel_matrix_vector_mult(A, v):
        M, N = A.shape
        w = np.zeros(M)
        thread_list = []
        chunk_size = M // NUM_THREADS

        with concurrent.futures.ThreadPoolExecutor(max_workers=NUM_THREADS) as executor:
            futures = []
            for i in range(NUM_THREADS):
                start = i * chunk_size
                end = (i + 1) * chunk_size if i < NUM_THREADS - 1 else M
                futures.append(executor.submit(matrix_vector_mult_row, start, end, A, v,
    w))

            concurrent.futures.wait(futures)

        return w

    # Exemplo de uso
    A = np.random.rand(M, N)
    v = np.random.rand(N)
    result = parallel_matrix_vector_mult(A, v)
```

Implementação 2: Implementação em Python usando o módulo concurrent.futures

Resposta ChatGPT

Neste código, usamos a biblioteca concurrent.futures.ThreadPoolExecutor para criar um pool de threads e distribuir as tarefas de multiplicação de matriz por vetor entre as threads. O número de threads é definido como 6 para corresponder aos núcleos físicos em sua máquina. Certifique-se de ajustar NUM_THREADS conforme necessário com base em sua máquina e no tamanho dos seus dados. A função np.dot é usada para calcular o produto escalar entre a linha da matriz e o vetor, o que é mais eficiente do que o loop manual.
Este código aproveita as otimizações internas do NumPy e a paralelização de threads fornecida pela biblioteca concurrent.futures para obter um desempenho eficiente.

Solicitamos também a implementação utilizando o módulo *threading* do Python, com objetivo de saber se o ChatGPT seria capaz de gerar uma solução eficiente, mesmo com o obstáculo trazido pelo Global Interpreter Lock (GIL) em relação ao paralelismo de *threads*.

Prompt com especificações avançadas e código exemplo

Paralelize esse código utilizando utilizando a biblioteca threading do python, distribuindo igualmente os processos entre os núcleos físicos, considerando que minha máquina possui 6 núcleos físicos e 6 lógicos

```
import numpy as np

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    for i in range(M):
        for j in range(N):
            w[i] += A[i, j] * v[j]
    return w
```

Resposta ChatGPT

Para paralelizar o código usando a biblioteca threading e distribuir igualmente os processos entre os núcleos físicos, você pode usar um pool de threads. No entanto, tenha em mente que o GIL (Global Interpreter Lock) do Python pode limitar o desempenho do paralelismo de threads em determinadas situações. No entanto, este exemplo ainda dividirá o trabalho entre as threads:

```
1
2 import numpy as np
3 import threading
4
5 M = 1000  # Número de linhas da matriz
6 N = 1000  # Número de colunas da matriz
7 NUM_THREADS = 6  # Número de threads a serem usadas
8
9 def matrix_vector_mult_row(start, end, A, v, result):
```

```
    M, N = A.shape
    for i in range(start, end):
        for j in range(N):
            result[i] += A[i, j] * v[j]

def parallel_matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    thread_list = []
    chunk_size = M // NUM_THREADS

    for i in range(NUM_THREADS):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i < NUM_THREADS - 1 else M
        thread = threading.Thread(target=matrix_vector_mult_row, args=(start, end, A, v,
    w))
        thread_list.append(thread)
        thread.start()

    for thread in thread_list:
        thread.join()

    return w

# Exemplo de uso
A = np.random.rand(M, N)
v = np.random.rand(N)
result = parallel_matrix_vector_mult(A, v)
```

Implementação 3: Implementação em Python usando o módulo threading.

Prompt com especificações avançadas e código exemplo

Paralelize esse código utilizando utilizando bibliotecas otimizadas e de paralelização disponíveis no python 3, dividindo as as tarefas igualmente entre processos e threads e utilizando o máximo da GPU disponível em minha máquina, utilize de operadores e funções otimizadas disponíveis na linguagem e busque o melhor gerenciamento de memória possível

```
import numpy as np

M = 1000  # Número de linhas da matriz
N = 1000  # Número de colunas da matriz

def matrix_vector_mult(A, v):
    M, N = A.shape
    w = np.zeros(M)
    for i in range(M):
        for j in range(N):
            w[i] += A[i, j] * v[j]
    return w
```

Resposta ChatGPT

Para aproveitar ao máximo a GPU disponível em sua máquina, você pode usar a biblioteca TensorFlow, que é altamente otimizada para cálculos em GPUs e oferece uma maneira eficiente de realizar a multiplicação de matrizes por vetores. Você também pode usar a biblioteca concurrent.futures para paralelizar o processo entre threads. Certifique-se de que você tenha o TensorFlow instalado em seu ambiente. Aqui está o código paralelizado que utiliza a GPU e distribui igualmente as tarefas entre processos e threads:

```
1
2   import numpy as np
3   import concurrent.futures
```

```
    import tensorflow as tf

    M = 1000  # Número de linhas da matriz
    N = 1000  # Número de colunas da matriz
    NUM_THREADS = 6  # Número de threads a serem usadas

    def matrix_vector_mult(A, v):
        # Utilize TensorFlow para fazer a multiplicação da matriz por vetor
        w = tf.linalg.matvec(A, v)
        return w

    def parallel_matrix_vector_mult(A, v):
        M, N = A.shape
        w = np.zeros(M)
        thread_list = []
        chunk_size = M // NUM_THREADS

        with concurrent.futures.ThreadPoolExecutor(max_workers=NUM_THREADS) as executor:
            futures = []
            for i in range(NUM_THREADS):
                start = i * chunk_size
                end = (i + 1) * chunk_size if i < NUM_THREADS - 1 else M
                futures.append(executor.submit(matrix_vector_mult, A[start:end], v))

            concurrent.futures.wait(futures)

            for i, future in enumerate(futures):
                start = i * chunk_size
                end = (i + 1) * chunk_size if i < NUM_THREADS - 1 else M
                w[start:end] = future.result()

        return w

    # Exemplo de uso
    A = np.random.rand(M, N)
    v = np.random.rand(N)
    A = tf.constant(A, dtype=tf.float32)
    v = tf.constant(v, dtype=tf.float32)

    result = parallel_matrix_vector_mult(A, v)
```

Implementação 4: Implementação em Python usando o módulo concurrent.futures e a biblioteca tensorflow.

4.2 Implementações propostas pelo ChatGPT - Algoritmo Ímpar-Par

Abordamos esse problema de forma diferente com o ChatGPT. Foi apresentada a descrição do problema e como ele deveria ser implementado sem apresentar um código exemplo, buscando validar as hipóteses **H2** e **H3**. A seguir veremos as soluções propostas pelo ChatGPT para resolução paralelizada do problema de Ordenação Ímpar-Par de acordo com as respectivas entradas, e avaliaremos posteriormente seus acertos e possíveis pontos de melhorias.

4.2.1 Solução proposta com *prompt* simplificado

A princípio, sem apresentar um código sequencial exemplo, o ChatGPT propôs a solução descrita na Implementação 9 e 5, uma delas com erro (Apêndice A) ao copiar e executar.

Prompt simplificado

Construa um algoritmo de Ordenação por transposição ímpar-par paralelo

Resposta ChatGPT

A ordenação par-ímpar é um algoritmo de ordenação que classifica os elementos de um array, separando os números pares dos ímpares e classificando-os separadamente. Para torná-lo paralelo, você pode usar programação paralela ou concorrente para classificar as duas metades (pares e ímpares) do array simultaneamente.
Aqui está um exemplo de um algoritmo de ordenação par-ímpar paralelo em Python, usando o módulo threading para executar as classificações paralelamente:

```
1
2  import threading
3
4  def odd_even_sort(arr):
5      n = len(arr)
6      is_sorted = False
7
8      while not is_sorted:
```

```
        is_sorted = True
        threads = []

        # Fase de classificação paralela para números pares
        for i in range(0, n - 1, 2):
            if i + 1 < n and arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                is_sorted = False
            threads.append(threading.Thread(target=sort_even, args=(arr, i, i + 2, n)))

        # Fase de classificação paralela para números ímpares
        for i in range(1, n - 1, 2):
            if i + 1 < n and arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                is_sorted = False
            threads.append(threading.Thread(target=sort_odd, args=(arr, i, i + 2, n))

        # Iniciar todas as threads
        for thread in threads:
            thread.start()

        # Aguardar todas as threads terminarem
        for thread in threads:
            thread.join()

def sort_even(arr, start, end, n):
    for i in range(start, end, 2):
        if i + 2 < n and arr[i] > arr[i + 2]:
            arr[i], arr[i + 2] = arr[i + 2], arr[i]

def sort_odd(arr, start, end, n):
    for i in range(start, end, 2):
        if i + 2 < n and arr[i] > arr[i + 2]:
            arr[i], arr[i + 2] = arr[i + 2], arr[i]

if __name__ == "__main__":
    arr = [3, 6, 1, 8, 4, 9, 2, 5, 7]
    print("Array não ordenado:", arr)

    odd_even_sort(arr)

    print("Array ordenado:", arr)
```

Implementação 5: Ordenação par-ímpar em python usando o módulo threading.

Com essa entrada o ChatGPT retornou uma implementação que demonstrou o erro `list index out of range` ao ser executada, embora consiga ordenar corretamente (de forma completamente diferente do esperado), isso ocasiona interrupção imediata do programa e até perda de dados em alguns casos. A implementação disponibilizada com erro encontra-se na seção de apêndices. Contudo, o ChatGPT também é capaz de identificar e corrigir erros no código. A implementação acima é a versão corrigida após novo *prompt*.

Corrige o erro: IndexError: list index out of range

Após correção o código funciona corretamente ordenando a lista na ordem crescente, mesmo que de forma bastante ineficiente.

4.2.2 Solução proposta com *prompt* detalhado

A Implementação 6 demonstra o código funcional mas incorreto, o que pode ter ocorrido pela entrada não especificar detalhadamente como a solução deveria funcionar.

Prompt com especificações mais detalhada

Construa um algoritmo de ordenação par-ímpar paralelo em python utilizando bibliotecas otimizadas, dividindo os cálculos entre os núcleos do processador com o objetivo de obter melhor desempenho. Busque contornar as limitações da linguagem e utilize todos os recursos disponíveis para isso

Resposta ChatGPT

A ordenação par-ímpar é um algoritmo de ordenação que separa os elementos em duas categorias: pares e ímpares, e, em seguida, classifica separadamente os elementos pares e ímpares. No final, os elementos pares e ímpares são mesclados para obter a lista ordenada. Vou mostrar como você pode implementar a ordenação par-ímpar em Python, aproveitando os recursos de multithreading para dividir o trabalho entre os núcleos do processador.

Para tirar o máximo proveito do processamento paralelo, usaremos a biblioteca concurrent.futures para executar a classificação de pares e ímpares em paralelo. Esta biblioteca é uma parte da biblioteca padrão do Python e permite que você crie *pools* de *threads* para executar tarefas em paralelo.

Aqui está o código para a ordenação par-ímpar paralela em Python:

```
1
2  import concurrent.futures
3
4  def parallel_sort(arr):
5      # Divide o array em elementos pares e ímpares
6      even = [x for x in arr if x % 2 == 0]
7      odd = [x for x in arr if x % 2 != 0]
8
9      # Função para classificar uma lista
10     def sort_list(lst):
11         return sorted(lst)
12
13     with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
14         # Classifique a lista de elementos pares em uma thread
15         even_sorted = executor.submit(sort_list, even)
16
17         # Classifique a lista de elementos ímpares em outra thread
18         odd_sorted = executor.submit(sort_list, odd)
19
20         # Espere a conclusão das duas tarefas
21         even_sorted = even_sorted.result()
22         odd_sorted = odd_sorted.result()
23
24     # Combine as listas ordenadas de elementos pares e ímpares
25     sorted_arr = []
26
27     i, j = 0, 0
```

```
28
29      while i < len(even_sorted) and j < len(odd_sorted):
30          if even_sorted[i] < odd_sorted[j]:
31              sorted_arr.append(even_sorted[i])
32              i += 1
33          else:
34              sorted_arr.append(odd_sorted[j])
35              j += 1
36
37      sorted_arr.extend(even_sorted[i:])
38      sorted_arr.extend(odd_sorted[j:])
39
40      return sorted_arr
41
42  # Exemplo de uso
43  if __name__ == "__main__":
44      unsorted_array = [9, 7, 6, 5, 1, 3, 8, 2, 4]
45      sorted_array = parallel_sort(unsorted_array)
46      print("Array ordenado:", sorted_array)
```

Implementação 6: Ordenação por transposição Ímpar-Par em python usando o módulo concurrent.futures.

Mais uma vez o ChatGPT propôs uma solução completamente diferente do esperado para o algoritmo em relação ao seu funcionamento. Neste caso, ao invés de ordenar baseado nos índices da lista como descrito na Seção 3.4, são construídas duas sublistas, uma com valores pares e outra com os ímpares, extraídos da lista principal. Essas sublistas são então ordenadas em paralelo usando *threads*, e os resultados são combinados para formar a lista ordenada final. Acontece que na primeira resposta, a IA entregou um código que ordenava as sublistas mas não ordenava a lista final, ou seja, teríamos apenas uma lista com valores pares ordenados concatenada à outra lista com valores ímpares ordenados. Supondo L uma lista de inteiros:

Lista Original:	`L = [18, 15, 12, 2, 5, 3]`
Lista de Pares Ordenada:	`L_par = [2, 12, 18]`
Lista de Ímpares Ordenada:	`L_ímpar = [3, 5, 15]`
Lista Final (Pares + Ímpares):	`L_final = [2, 12, 18, 3, 5, 15]`

A lista final deveria ser `L_final_correto = [2, 3, 5, 12, 15, 18]`, o que foi alcançado após reportar o erro do código para o ChatGPT.

Entretando, mesmo após dezenas de tentativas na linguagem Python, inclusive descrevendo o problema e passando seu pseudocódigo, o ChatGPT não conseguiu fornecer uma solução correta e eficiente para o problema da Ordenação por Transposição Ímpar-Par, respeitando as regras do algoritmo.

4.2.3 Solução proposta com *prompt* detalhado e especificação da linguagem

Passamos então a explorar o modelo para gerar uma solução na linguagem C utilizando exatamente a mesma entrada utilizada para a linguagem Python, com detalhes do funcionamento do problema, expostas nas Implementação 7 e 8 (Implementação com OpenMP). Esse *prompt* foi também utilizado para gerar soluções na linguagem Python, mas o ChatGPT continuou na ideia das Implementações 5 e 6.

Prompt com especificações detalhadas

Construa um algoritmo de ordenação por transposição ímpar-par paralelo em C utilizando bibliotecas otimizadas, dividindo os cálculos entre os núcleos do processador com o objetivo de obter melhor desempenho. Busque contornar as limitações da linguagem e utilize todos os recursos necessários disponíveis para isso.

O algoritmo funciona da seguinte forma: Supondo que queremos realizar o rearranjo em ordem crescente da lista `arr = [3,5,2,4,1]`.

Fase 0: Começaríamos analisando as posições pares com seus sucessores:
`[posicao, valor]`
`[0, 3]` > `[1, 5]` ? False →`[3,5,2,4,1]`
`[2, 2]` > `[3, 4]` ? False →`[3,5,2,4,1]`

Fase 1: Testaríamos elementos na posição ímpar com seus sucessores:

`[1, 5]` > `[2, 2]` ? True →`[3,2,5,4,1]`
`[3, 4]` > `[4, 1]` ? True →`[3,2,5,1,4]`

e assim sucessivamente até a fase 4 onde teríamos a lista completamente ordenada `arr = [1,2,3,4,5]`

Esse é o pseudocódigo do problema:

```
n = length(arr)
for phase from 0 to n-1 do
    if phase is even then
        for i from 1 to n step 2 do
            if arr[i-1] > arr[i] then
                swap(arr[i-1], arr[i])
    else
        for i from 1 to n-1 step 2 do
            if arr[i] > arr[i+1] then
                swap(arr[i], arr[i+1])
return arr
```

Resposta ChatGPT

Aqui está uma implementação em C do algoritmo de ordenação por transposição ímpar-par, otimizada para aproveitar o paralelismo com *threads*. Para esta implementação, usaremos a biblioteca pthread para criar *threads* e dividir o trabalho entre elas:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 4  // Ajuste o número de threads conforme necessário

int arr[] = {3, 5, 2, 4, 1};
int n = 5;

void compare_and_swap(int start, int step) {
    for (int i = start; i < n - 1; i += step) {
        if (arr[i] > arr[i + 1]) {
            int temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }
}

void* parallel_sort(void* arg) {
    int thread_id = *((int*)arg);
    for (int phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
            for (int i = 1; i < n; i += 2) {
                compare_and_swap(i - 1, 2);
            }
        } else {
            for (int i = 1; i < n - 1; i += 2) {
                compare_and_swap(i, 2);
            }
        }
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
```

```c

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        if (pthread_create(&threads[i], NULL, parallel_sort, &thread_ids[i]) != 0) {
            perror("Error creating thread");
            exit(1);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("Error joining thread");
            exit(1);
        }
    }

    printf("Lista ordenada: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Implementação 7: Implementação algoritmo de ordenação por transposição ímpar-par usando pthreads

Resposta ChatGPT

Neste código, criamos várias *threads* que executam a função *parallel_sort*, que realiza a ordenação por transposição ímpar-par. Cada *thread* opera em uma parte da matriz e, portanto, as tarefas de ordenação são distribuídas entre as *threads*, o que pode melhorar o desempenho em sistemas multi-core. Certifique-se de ajustar o valor de NUM_THREADS de acordo com o número de núcleos de processamento disponíveis em seu sistema para obter o melhor desempenho possível.

Prompt com especificações mais específicas

Utilize também o OpenMP

Resposta ChatGPT

Aqui está um exemplo de um algoritmo de ordenação por transposição ímpar-par paralelo em C utilizando OpenMP:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void oddEvenSort(int arr[], int n) {
    for (int phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
            #pragma omp parallel for
            for (int i = 1; i < n; i += 2) {
                if (arr[i - 1] > arr[i]) {
                    int temp = arr[i - 1];
                    arr[i - 1] = arr[i];
                    arr[i] = temp;
                }
            }
        } else {
            #pragma omp parallel for
            for (int i = 1; i < n - 1; i += 2) {
                if (arr[i] > arr[i + 1]) {
                    int temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
                }
            }
        }
    }
}

int main() {
    int arr[] = {3, 5, 2, 4, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

```
37      printf("\n");
38
39      oddEvenSort(arr, n);
40
41      printf("Sorted array: ");
42      for (int i = 0; i < n; i++) {
43          printf("%d ", arr[i]);
44      }
45      printf("\n");
46
47      return 0;
48  }
```

Implementação 8: Implementação algoritmo de ordenação por transposição ímpar-par usando OpenMP

4.3 Validação das saídas

Antes de analisar o desempenho e as técnicas utilizadas pelo ChatGPT na proposta de solução para os problemas, é importante validar a corretude dos resultados. Esta validação garante que as soluções propostas não apenas otimizam o tempo de execução, mas também produzem resultados corretos e confiáveis.

Para cada problema abordado, foi definido um caso teste específico para comprovar a veracidade e corretude dos resultados. Estes casos de teste são para garantir que as soluções sejam, no mínimo, matematicamente e logicamente corretas.

Para o problema apresentado na Figura 4, onde discutimos a multiplicação de matriz por um vetor, utilizamos um caso de teste para validar a solução. A matriz A e o vetor B utilizados no teste estão descritos na Figura 7, e o vetor C é o resultado esperado para todos os códigos descritos na Seção 3.3.

```
# Exemplo de Matriz e Vetor
A_example = np.array([
    [2, 4, 6, 8, 10],
    [1, 3, 5, 7, 9],
    [10, 20, 30, 40, 50],
    [5, 10, 15, 20, 25],
    [2, 3, 4, 5, 6]
])

v_example = np.array([1, 2, 3, 4, 5])

# Resultado esperado: [110, 95, 550, 275, 70]
```

Figura 7: Caso de teste para a multiplicação matriz x vetor em Python.

Já para o problema apresentado na Figura 5, onde demonstramos o algoritmo de ordenação Ímpar-Par, o caso de teste para validar a solução foi a lista [3,5,2,4,1]. A lista utilizada no teste está descrita na Figura 8, bem como o resultado esperado para todos os códigos descritos na Seção 3.3.

```
    listas = [
        [3,5,2,4,1],
        [3,5],
        [3],
        []
    ]

    resultados:
        [3,5,2,4,1] -> [1,2,3,4,5]
        [3,5] -> [3,5]
        [3] -> [3]
        [] -> []
```

Figura 8: Casos de teste para ordenação por transposição ímpar-par

4.4 Análise dos Códigos Gerados

Primeiramente, as implementações propostas apresentaram uma melhoria significativa no desempenho quando comparadas às suas contrapartes sequenciais. Uma grande parcela desse avanço pode ser creditada ao uso de bibliotecas otimizadas, como `Numpy` e `TensorFlow`. Essa estratégia não só tornou o código mais simples e legível, mas também contribuiu para a sua manutenção, já que essas bibliotecas são constantemente aprimoradas pela comunidade de desenvolvedores.

Quanto ao problema de ordenação, o modelo encontrou algumas dificuldades. Ele não conseguiu gerar uma solução em Python que atendesse completamente às necessidades do algoritmo. No entanto, ele foi capaz de criar uma solução em C visualmente intuitiva e que se mostrou bastante eficiente.

4.4.1 Legibilidade e Manutenibilidade

Para os dois problemas, os códigos gerados mostraram-se claros e bem estruturados, seguindo boas práticas de programação. A documentação inserida facilita a compreensão das funções e da lógica implementada, contribuindo para a manutenibilidade. As variáveis e funções nos códigos gerados foram nomeadas de maneira clara e descritiva. Por exemplo, a função `matrix_vector_mult` deixa explícito que seu propósito é realizar a multiplicação de uma matriz por um vetor. Além disso, o uso de `A` para representar a matriz e `v` para o vetor segue convenções matemáticas comuns, facilitando a compreensão para indivíduos com formação em matemática ou ciências exatas.

Vale ressaltar que, embora o código esteja em inglês, o ChatGPT tem a capacidade de gerar ou traduzir códigos para diversos idiomas.

4.4.2 Eficiência do Código

- **Matriz x Vetor**
  A implementação paralela demonstrou ser eficiente, fazendo uso dos recursos computacionais disponíveis. A divisão do trabalho entre núcleos *threads* foi realizada

de maneira equitativa, maximizando a utilização dos núcleos do processador e reduzindo o tempo de execução, em comparação com a implementação sequencial. Entretando, algumas implementações obtiveram bons resultados mais por utilizarem bibliotecas otimizadas do que por consequência do paralelismo criado.

- **Ordenação por transposição ímpar-par**
  A implementação do algoritmo de ordenação por transposição ímpar-par, apesar de não ter sido completamente desenvolvida em Python, mostrou-se promissora em termos de eficiência, quando implementada em C. A estrutura do algoritmo permite a execução paralela de transposições, o que pode levar a uma redução significativa no tempo de execução para *arrays* de grande tamanho. No entanto, vale destacar que no gerenciamento dinâmico de memória ao usar como entrada *arrays* muito grande, geralmente os códigos inciais disponibilizados pelo ChatGPT não tratam esse problema. No entanto, isso pode ser tratado com prévio conhecimento do programador e o auxílio da IA.

É importante notar que a eficiência dos códigos também dependem da implementação específica e das características do *hardware* em que é executado. Portanto, testes e ajustes adicionais podem ser necessários para otimizar o desempenho em diferentes cenários e garantir que o algoritmo seja executado de maneira eficiente em todas as situações.

4.4.3 Escalabilidade

- **Matriz x Vetor**
  Contrariamente ao que se poderia esperar, o aumento no número de *threads* não resultou necessariamente em uma melhoria no desempenho. Em alguns casos, observou-se que configurar o número de *threads* para um (NUM_THREADS = 1) proporcionou resultados melhores do que usar um número maior de *threads* (NUM_THREADS = 6).

  Isso pode ser atribuído a vários fatores, incluindo a sobrecarga associada à criação e gerenciamento de múltiplas *threads*, bem como a natureza das operações de álgebra

linear realizadas. Operações de álgebra linear são intensivas em CPU e podem se beneficiar de execução em paralelo, mas a sobrecarga de sincronização e comunicação entre *threads* pode anular os ganhos de desempenho em alguns casos, especialmente quando o tamanho da matriz não é suficientemente grande para justificar o paralelismo.

Além disso, é importante considerar o *Global Interpreter Lock* (GIL) do Python, que pode limitar a execução paralela em *threads*. No entanto, as bibliotecas utilizadas, como `Numpy` e `TensorFlow`, são implementadas de maneira a contornar o GIL para operações específicas, permitindo que elas tirem proveito de múltiplos núcleos e processadores.

Portanto, a escalabilidade dos códigos paralelizados depende fortemente do contexto específico em que são executados, incluindo o tamanho dos dados e a natureza das operações realizadas. Isso destaca a importância de realizar uma análise de desempenho cuidadosa e ajustar o número de *threads*, de acordo com as características específicas do problema e do *hardware* disponível, por isso é de grande importância o programador ter uma base do problema ao qual está enfrentando.

- **Ordenação por transposição ímpar-par**
  Este algoritmo, por sua natureza, possui um potencial considerável para execução paralela, uma vez que várias transposições podem ser realizadas simultaneamente.

  Durante os testes, observou-se que o aumento no número de *threads* tendeu a resultar em uma melhoria no tempo de execução, especialmente para *arrays* de grande tamanho. Isso se deve ao fato de que o algoritmo consegue dividir eficientemente o trabalho entre as *threads*, minimizando a sobrecarga de comunicação e sincronização.

  Em suma, o algoritmo de ordenação por transposição ímpar-par demonstra uma boa escalabilidade, especialmente para *arrays* de grande tamanho, mas é importante estar atento ao ponto de saturação para garantir que os recursos computacionais sejam utilizados de maneira eficiente.

4.5 Comparação de Desempenho

A avaliação do desempenho das diferentes abordagens de implementação é crucial para entender as vantagens e desvantagens de cada uma delas. Nesta seção, apresentamos gráficos que ilustram os tempos de execução para cada solução dos problemas descritos nas Seções 3.3 e 3.4, incluindo os resultados para uma implementação em C utilizando OpenMP.

4.5.1 Desempenho Matriz x Vetor

As Figuras 9, 10 e 11, a seguir, ilustram os tempos de execução para cada solução do problema descrito na Seção 3.3.

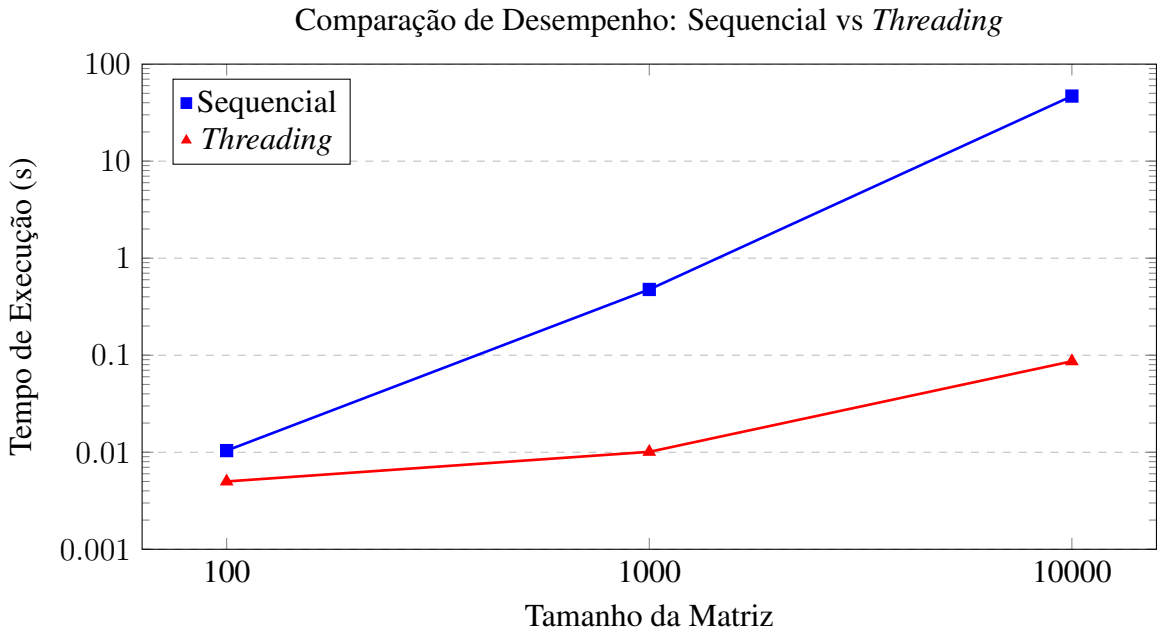

Figura 9: Comparação de desempenho entre as abordagens sequencial e threading.

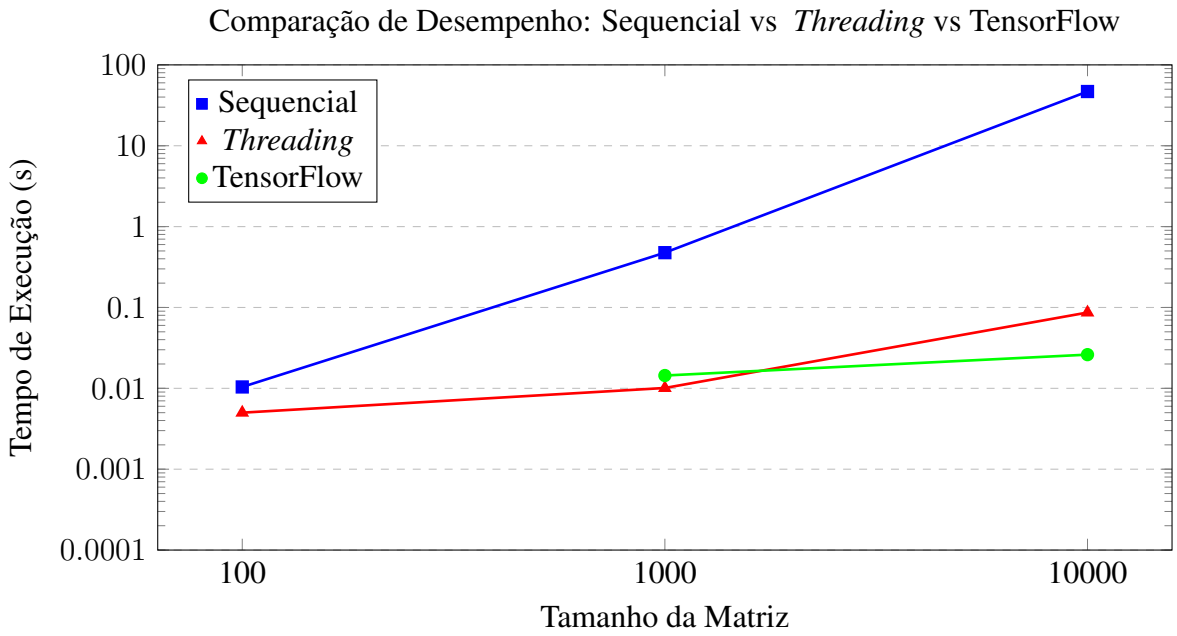

Figura 10: Comparação de desempenho entre as abordagens sequencial, *threading* e TensorFlow.

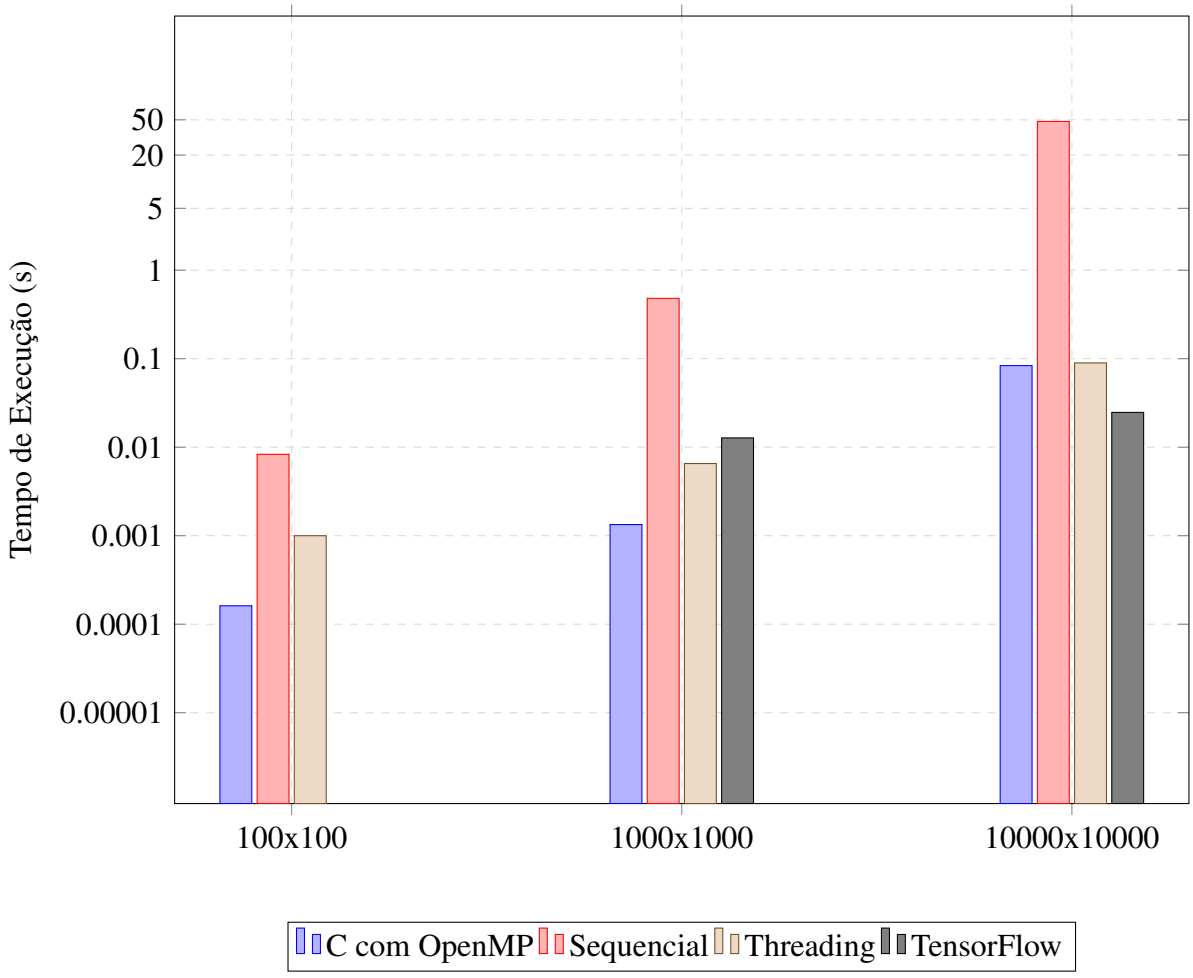

Figura 11: Comparação de desempenho entre as abordagens C com OpenMP, Sequencial, Threading e TensorFlow.

Os gráficos apresentados nas Figuras 9, 10 e 11 ilustram o tempo de execução para diferentes tamanhos de matriz nas abordagens sequencial, *threading* e TensorFlow, em Python, e C com OpenMP.

A implementação em Python com *threading*, gerada pelo ChatGPT, apesar de ser significativamente mais rápida que a implementação sequencial, como vemos na Figura 9, ela não alcança o desempenho da implementação em C com OpenMP, especialmente para matrizes de grande porte. É importante ressaltar também que, embora o módulo *threa-*

ding seja utilizado, não ocorre um paralelismo real de CPU devido ao Global Interpreter Lock (GIL) do Python. Este mecanismo impede a execução simultânea de múltiplos threads de bytecode Python, limitando a capacidade de paralelismo verdadeiro em operações intensivas de CPU.

No entanto, a implementação com *threading* pode ser mais eficiente em cenários específicos, especialmente quando envolve operações de I/O ou chamadas a bibliotecas externas que podem liberar o GIL. Além disso, a gestão de memória compartilhada entre threads é geralmente mais eficiente em termos de recursos computacionais do que a criação e comunicação entre processos separados, como no caso do *multiprocessing*. Portanto, a eficiência observada na implementação com *threading* advém mais da gestão eficiente de recursos e da divisão de tarefas do que do paralelismo de CPU real.

Contudo, vale destacar que os códigos gerados pelo ChatGPT são legíveis e fáceis de manter, o que é importante, especialmente considerando a complexidade inerente à programação paralela. A clareza do código facilita a compreensão, a depuração e a manutenção, aspectos cruciais em desenvolvimento de software. Isso valida a Hipótese **H3**, sugerindo que a integração do ChatGPT no processo de desenvolvimento pode acelerar o ciclo de desenvolvimento de software.

A implementação utilizando *TensorFlow* uma biblioteca otimizada para operações de álgebra linear e execução em GPU, apresenta um desempenho maior que os demais, Figura 10. Ela supera todas as outras abordagens em todos os tamanhos de matriz testados, ressaltando o potencial do uso de bibliotecas especializadas e otimizadas em Python. Isso também valida a Hipótese **H1**, pois mostra que o ChatGPT pode ser adaptado para identificar oportunidades de paralelismo e gerar códigos otimizados com a ajuda de bibliotecas especializadas.

A implementação em C com OpenMP destaca o poder do paralelismo em nível de *thread* e a eficiência alcançável com uma linguagem de baixo nível, Figura 11. Embora ofereça um desempenho superior, exige conhecimento aprofundado de programação paralela e otimizações específicas de linguagem de baixo nível.

Em resumo, os resultados evidenciam a capacidade do ChatGPT em sugerir algo-

ritmos em Python que melhoram significativamente o desempenho em relação à execução sequencial, sendo ao mesmo tempo legíveis e fáceis de manter. Apesar da lacuna de desempenho em comparação com implementações em linguagens de baixo nível, as soluções propostas pela IA conseguem se destacar em termos de eficiência e praticidade, validando as Hipóteses **H1** e **H3**.

4.5.2 Desempenho Ordenação por Transposição Ímpar-Par

A Tabela 3 a seguir ilustra os tempos de execução para cada solução do problema de ordenação de listas por transposição Ímpar-Par (Seção 3.4).

Tamanho do Array	PTHREAD	OpenMP ChatGPT	OpenMP Pacheco (2011)
1000	0,385s	0,013s	0,005s
3000	9,669s	0,023s	0,013s
5000	44,578s	0,042s	0,026s
10000	366,139	0,097s	0,077s

Tabela 3: Tempos médios de execução para diferentes tamanhos de array e métodos de paralelização

A Figura 12 mostra que o tempo de execução para a implementação usando *Pthreads*, disponível no repositório Emerssu. (2023). Paralell-chatgpt-solutions, aumenta exponencialmente com o tamanho do *array*, o que é um indicativo de escalabilidade ruim. Isso pode ser atribuído à sobrecarga de gerenciamento de *threads* e a mal divisão de tarefas . Cada *thread* executa a ordenação completa, o que pode levar a redundâncias e ineficiências, especialmente para grandes tamanhos de *array*. A implementação não parece tirar vantagem completa do paralelismo, resultando em um desempenho inferior.

O algoritmo sequencial em Python, disponível nos apêndices deste trabalho, mostra um aumento linear no tempo de execução à medida que o tamanho do *array* aumenta, como esperado para a solução sequencial. A simplicidade do código e a ausência de sobrecarga de gerenciamento de *threads* contribuem para seu bom desempenho em *arrays* menores.

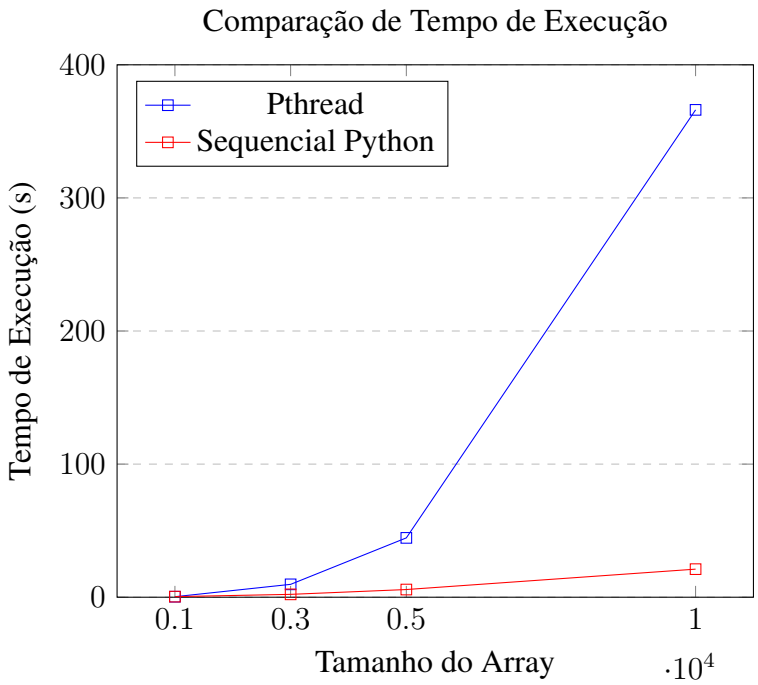

Figura 12: Comparação de Tempo de Execução entre Pthread e Algoritmo Sequencial em Python

A implementação do algoritmo de ordenação por transposição Ímpar-Par, utilizando OpenMP, feita com base nas instruções do ChatGPT, mostra um bom desempenho, com tempos de execução aumentando levemente com o tamanho do *array*. O uso eficiente das diretivas OpenMP e a paralelização eficaz contribuem para esse desempenho. Isso valida a Hipótese **H2**, mostrando que o ChatGPT pode gerar códigos paralelos otimizados, embora ainda haja espaço para melhorias.

A implementação disponível no livro Pacheco (2011) também mostra um aumento regular no tempo de execução, que é consistentemente mais lento do que a implementação do ChatGPT para tamanhos de *array* maiores no intervalo testado. Isso pode ser devido a diferenças nas estratégias de paralelização ou na configuração do ambiente de execução. De qualquer forma, é um resultado bastante interessante por parte do ChatGPT, pois atinge nível de competitividade com implementações feitas por especialistas e bastante conhecidas no âmbito da programação paralela. A seguir podemos ver o gráfico da Figura 13

onde esboça o desempenho superior do algoritmo desenvolvido pela IA em relação ao disponibilizado no livro Pacheco (2011).

O ganho percentual de tempo de execução também pode ser consultado na Tabela 4.

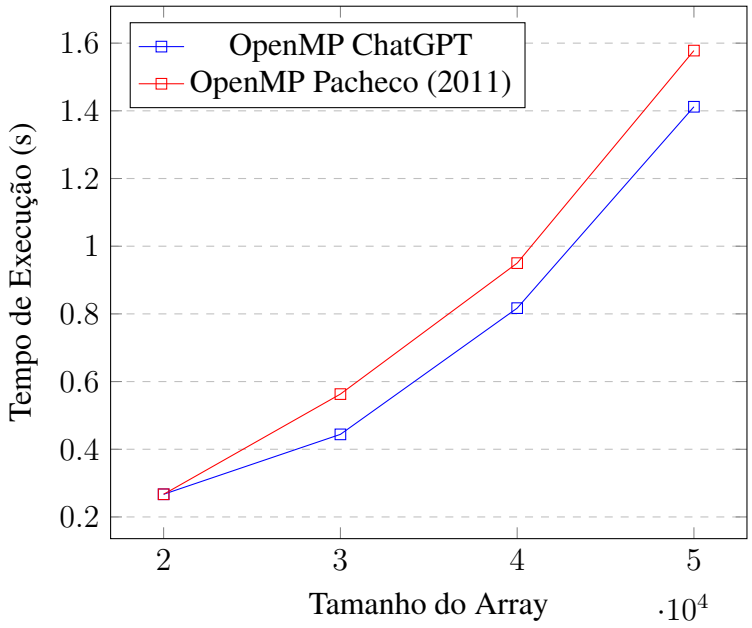

Figura 13: Comparação dos tempos de execução do algoritmo desenvolvido com OpenMP pelo ChatGPT em relação ao Algoritmo desenvolvido com OpenMP disponível no Pacheco (2011)

Tamanho do Array	Melhoria Percentual
20,000	0,00%
30,000	21,14%
40,000	14,00%
50,000	10,52%

Tabela 4: Melhoria percentual do algoritmo desenvolvido com OpenMP pelo ChatGPT em relação ao Algoritmo desenvolvido com OpenMP disponível no Pacheco (2011)

Com base na análise dos códigos e dos gráficos, podemos concluir que o uso efi-

ciente do paralelismo e gestão é muito importante para obter um bom desempenho em algoritmos de ordenação.

A implementação com *Pthreads*, apesar de oferecer um certo controle sobre a criação e gerenciamento de *threads*, revelou-se menos eficiente em termos de escalabilidade e desempenho, principalmente devido à redundância nas operações de ordenação e à falta de um esquema eficaz de balanceamento de carga. Por outro lado, as implementações utilizando OpenMP demonstraram uma capacidade superior de escalar com o aumento do tamanho do *array*, graças à abstração de alto nível proporcionada pelo OpenMP e à sua capacidade de distribuir automaticamente o trabalho entre as *threads* disponíveis.

5 CONCLUSÕES E TRABALHOS FUTUROS

O objetivo principal deste estudo foi explorar as capacidades e limitações do ChatGPT na paralelização de códigos, uma área crítica para melhorar o desempenho computacional em diversos campos da ciência e engenharia. Através de uma abordagem envolvendo estudos de caso, experimentos práticos e avaliações quantitativas, conseguimos não apenas destacar os benefícios e desafios da integração do ChatGPT na paralelização, mas também estabelecer uma base para pesquisas futuras na área.

Os resultados obtidos indicam que o ChatGPT pode, de fato, ser uma ferramenta valiosa para a programação de alto desempenho, gerando códigos paralelos eficientes e otimizados em situações onde as instruções fornecidas são claras e específicas, podendo chegar em pelo menos 20% de melhoria. No entanto, também ficou evidente que a qualidade da paralelização depende significativamente da complexidade do problema em questão e da precisão das instruções fornecidas ao modelo. Em casos de instruções vagas ou genéricas, as soluções geradas tendem a ser menos otimizadas, ressaltando a necessidade de um conhecimento prévio sólido por parte dos usuários para tirar o máximo proveito das capacidades do ChatGPT.

Durante o desenvolvimento deste trabalho, algumas limitações e desafios se tornaram aparentes. A dependência de instruções precisas e a dificuldade do modelo em lidar com problemas de alta complexidade sem orientação detalhada são pontos que merecem atenção especial. Além disso, a avaliação da eficácia do ChatGPT em paralelizar códigos mostrou-se uma tarefa não trivial, exigindo uma abordagem e um conjunto de métricas bem definidas.

5.1 Trabalhos Futuros

Diante das descobertas e desafios apresentados neste estudo, oportunidades para trabalhos futuros se desenham. Uma direção seria aprofundar a investigação sobre as estratégias de instrução que podem ser empregadas para otimizar a eficácia do ChatGPT em tarefas de paralelização, buscando desenvolver diretrizes claras e métodos que possam ser facilmente replicados por outros pesquisadores e desenvolvedores.

Outra área de interesse seria explorar a integração do ChatGPT com outras ferramentas e tecnologias de programação paralela, visando criar um ecossistema mais robusto e eficiente para o desenvolvimento de software de alto desempenho. Isso poderia incluir a criação de plugins ou extensões que facilitassem a interação entre o ChatGPT e ambientes de desenvolvimento integrados (IDEs), tal como funciona o GitHub Copilot, por exemplo.

Além disso, seria interessante conduzir estudos comparativos entre o ChatGPT e outras ferramentas ou métodos de paralelização, a fim de estabelecer *benchmarks* claros e identificar áreas específicas onde o ChatGPT se destaca ou necessita de melhorias.

Por fim, a realização de estudos de caso em aplicações do mundo real, envolvendo problemas complexos e cenários de uso diversificados, ajudaria a validar as descobertas deste trabalho e a expandir nosso entendimento sobre as capacidades e limitações do ChatGPT na prática.

Este trabalho serve como um ponto de partida para futuras investigações, abrindo caminho para uma exploração mais aprofundada e abrangente do potencial do ChatGPT e outras ferramentas baseadas em Inteligência Artificial na programação de alto desempenho.

REFERÊNCIAS

Som Biswas. Role of chatgpt in computer programming. *Mesopotamian Journal of Computer Science*, 2023. doi: 10.58496/MJCSC/2023/002. URL https://doi.org/10.58496/MJCSC/2023/002.

concurrent.futures. *concurrent.futures — Launching parallel tasks*. https://docs.python.org/3/library/concurrent.futures.html. Acessado em: 18 de outubro de 2023.

Elsevier. An Introduction to Parallel Programming - Companion Site, Acessado em: 12 de agosto de 2023. URL https://booksite.elsevier.com/9780123742605/?ISBN=9780123742605.

Brown. et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Haoye Tian et al. Is chatgpt the ultimate programming assistant - how far is it? *arXiv preprint arXiv:2304.11938*, 2023. doi: 10.48550/arXiv.2304.11938. URL https://doi.org/10.48550/arXiv.2304.11938.

Samia Kabir et al. Who answers it better? an in-depth analysis of chatgpt and stack overflow answers to software engineering questions. *arXiv preprint arXiv:2308.02312*, 2023a. doi: 10.48550/arXiv.2308.02312. URL https://doi.org/10.48550/arXiv.2308.02312.

Sayed Erfan Arefin et al. Unmasking the giant: A comprehensive evaluation of chatgpt's proficiency in coding algorithms and data structures. *arXiv preprint arXiv:2307.05360*, 2023b. doi: 10.48550/arXiv.2307.05360. URL https://doi.org/10.48550/arXiv.2307.05360.

GitHub Copilot. GitHub Copilot: Your AI pair programmer. URL https://github.com/features/copilot.

Global Interpreter Lock. Global interpreter lock. https://wiki.python.org/moin/GlobalInterpreterLock. [Acessado em: 03 de Novembro de 2023].

OpenAI. Better language models and their implications. https://openai.com/research/better-language-models, 2019.

OpenAI. Openai - introducing chatgpt, 2023. URL https://openai.com/blog/chatgpt.

Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 9780123742605.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. URL https://doi.org/10.48550/arXiv.1706.03762.

You Yang, Ping Yu, and Yan Gan. Experimental study on the five sort algorithms. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1314–1317, 2011. doi: 10.1109/MACE.2011.5987184.

ANEXOS E APÊNDICES

APÊNDICE A

```
1
2  import threading
3
4  def odd_even_sort(arr):
5      n = len(arr)
6      is_sorted = False
7
8      while not is_sorted:
9          is_sorted = True
10         threads = []
11
12         # Fase de classificação paralela para números pares
13         for i in range(0, n - 1, 2):
14             if arr[i] > arr[i + 1]:
15                 arr[i], arr[i + 1] = arr[i + 1], arr[i]
16                 is_sorted = False
17             threads.append(threading.Thread(target=sort_even, args=(arr, i, i + 2)))
18
19         # Fase de classificação paralela para números ímpares
20         for i in range(1, n - 1, 2):
21             if arr[i] > arr[i + 1]:
22                 arr[i], arr[i + 1] = arr[i + 1], arr[i]
23                 is_sorted = False
24             threads.append(threading.Thread(target=sort_odd, args=(arr, i, i + 2)))
25
26         # Iniciar todas as threads
27         for thread in threads:
28             thread.start()
29
30         # Aguardar todas as threads terminarem
31         for thread in threads:
32             thread.join()
33
34 def sort_even(arr, start, end):
35     for i in range(start, end, 2):
36         if arr[i] > arr[i + 2]:
37             arr[i], arr[i + 2] = arr[i + 2], arr[i]
38
39 def sort_odd(arr, start, end):
40     for i in range(start, end, 2):
41         if arr[i] > arr[i + 2]:
42             arr[i], arr[i + 2] = arr[i + 2], arr[i]
43
44 if __name__ == "__main__":
```

```
    arr = [3, 6, 1, 8, 4, 9, 2, 5, 7]
    print("Array não ordenado:", arr)

    odd_even_sort(arr)

    print("Array ordenado:", arr)
```

Implementação 9: Implementação algoritmo par-ímpar em python com erro

APÊNDICE B

```python
import concurrent.futures

def parallel_sort(arr):
    # Divide o array em elementos pares e ímpares
    even = [x for x in arr if x % 2 == 0]
    odd = [x for x in arr if x % 2 != 0]

    # Função para classificar uma lista
    def sort_list(lst):
        return sorted(lst)

    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        # Classifique a lista de elementos pares em uma thread
        even_sorted = executor.submit(sort_list, even)

        # Classifique a lista de elementos ímpares em outra thread
        odd_sorted = executor.submit(sort_list, odd)

        # Espere a conclusão das duas tarefas
        even_sorted = even_sorted.result()
        odd_sorted = odd_sorted.result()

    # Combine as listas ordenadas de elementos pares e ímpares
    sorted_arr = even_sorted + odd_sorted

    return sorted_arr

# Exemplo de uso
if __name__ == "__main__":
    unsorted_array = [9, 7, 6, 5, 1, 3, 8, 2, 4]
    sorted_array = parallel_sort(unsorted_array)
    print("Array ordenado:", sorted_array)
```

Implementação 10: Implementação algoritmo par-ímpar em python com lógica incorreta

APÊNDICE C

```python
from numpy import random

def odd_even_transposition_sort(arr):
    n = len(arr)
    for phase in range(n):
        if phase % 2 == 0:
            for i in range(1, n, 2):
                if arr[i - 1] > arr[i]:
                    arr[i - 1], arr[i] = arr[i], arr[i - 1]
        else:
            for i in range(1, n - 1, 2):
                if arr[i] > arr[i + 1]:
                    arr[i], arr[i + 1] = arr[i + 1], arr[i]
    return arr

# Exemplo de uso
lista = random.randint(0, 100, 10000) # [3, 5, 2, 4, 1]
print("Lista original:", lista)
lista_ordenada = odd_even_transposition_sort(lista)
print("Lista ordenada:", lista_ordenada)
```

Implementação 11: Implementação sequencial algoritmo Ímpar-par em python