RELATÓRIO: ESTRATÉGIAS DIVIDIR E CONQUISTAR COM MULTIPROCESSAMENTO EM PYTHON

Ryan Augusto Brandão Salles - 221008436
Víctor Moreira Almeida - 221008481

Dezembro de 2024

1 Introdução

Essa seção visa trazer alguns assuntos introdutórios, mais precisamente, situar o leitor sobre o que se trata esse documento, qual o trabalho que busca ser realizado e qual exatamente seria sua motivação, além de demais especificidades de caráter estritamente introdutório ao teor do que será tratado nesse relatório.

1.1 O que é esse documento

Esse documento é um relatório que sumariza resultados obtidos durante a implementação e testagem de algoritmos de Divisão & Conquista paralelizados para a disciplina de Projeto de Algoritmos durante o semestre 2024.2, bem como detalhes sobre como exatamente esses algoritmos foram implementados.
Mais especificamente, os resultados obtidos para a paralelização dos algoritmo mergeSort. Detalhes sobre esse algoritmo podem ser encontrados na seção 3.
Ademais, os algoritmos foram escolhidos por sua facilidade de implementação e paralelização, mantendo o foco na possibilidade de paralelização como prova de conceito, não necessariamente um código pronto para produção.

1.2 O que não é esse documento

Esse documento não busca como objetivo ser uma pesquisa formal sobre paralelização de algoritmos de Divisão & Conquista tampouco uma implementação completa, apenas um breve experimento que a dupla julgou ser interessante para seus colegas e avaliadores.
No mais, esse documento certamente não é uma prova de que absolutamente todos os algoritmos Dividir & Conquistar necessariamente se beneficiam de uma estratégia de multiprocessamento, certamente exigindo um estudo muito mais

aprofundado do que o aqui realizado.

1.3 Justificativa I: possibilidade de melhor desempenho e suas consequências

Um melhor desempenho em qualquer um desses algoritmos pode ser crucial para problemas estupidamente grandes. Melhor desempenho pode ser a diferença entre aguardar um ano para a execução de um algoritmo ou 20 anos.

1.4 Justificativa II: interesse acadêmico e lúdico

Algoritmos acadêmicos são classicamente implementados utilizando um core e uma thread. Nossa dupla gostaria de ver as possibilidades de melhor performance de um algoritmo ao utilizar o poder completo do hardware em nossas mãos.
Mais especificamente, algoritmos de tipo Dividir & Conquistar, segundo o artigo no tópico da wikipédia, são geralmente trivialmente paralelizáveis, o que levanta a questão de exatamente quanta performance poderia ser ganha caso fossem de fato paralelizáveis, adicionando capacidade de realmente utilizar todo o poder do hardware moderno disponíveis em nossas mãos.
No mais, simplesmente parece algo divertido de se fazer e se ver rodando e isso deveria ser justificativa o suficiente, caso as **outras justificativas** não existissem.

1.5 Escolha de linguagem

A linguagem escolhida para essa implementação foi a Python, por duas razões simples:

- simplicidade de uso
- disponibilidade de utensílios

Onde por "simplicidade de uso"queremos dizer "Nós não precisamos pensar muito em como algo precisa ser feito ou pesquisar sintaxe"e, por "disponibilidade de utensílios"queremos dizer "não é necessário escrever algo tão básico quanto uma lista encadeada ou um método pop, a linguagem já possui isso implementada na biblioteca padrão".
No mais, Python, para o nosso relatório, funciona bem ao possuir uma implementação de criação de processos portável, algo que dificultaria em muito o uso de linguagem C, considerando que os membros utilizam tanto os sistemas Windows e Linux.
Utilizando Python, podemos testar a performance dos algoritmos em uma multitude de hardware sem a necessidade de reinplementar um código para um determinado sistema operacional que esse hardware está rodando no momento.

2 Metodologia

Para esse experimento, a metodologia utilizada é o seguinte procedimento:

1. implementação do algoritmo base (single thread, single process)
2. avaliação da performance base
3. modificação do algoritmo para paralelização
4. avaliação da performance paralelizada
5. comparação de performances e determinação do speedup

Esses itens servirão de subtópicos para a metodologia e serão expandidos a seguir.
Demais tópicos além dos enumerados serão adicionados conforme necessidade.

2.1 Implementação do algoritmo base

A implementação do algoritmo base consiste em implementar o algoritmo em python inicialmente para rodar em apenas um processo e uma thread, utilizando somente um dos cores disponíveis da cpu, ou seja, para um determinado algoritmo de complexidade assintótica

$$O(n \log(n)) \tag{1}$$

espera-se que, para um determinado input de tamanho n e dada uma determinada cpu de capaz de executar k instruções por segundo, o algoritmo demore cerca de

$$\frac{n}{k} \log(\frac{n}{k}) \tag{2}$$

segundos para executar o dado algoritmo.
O propósito dessa etapa é obter o algoritmo que levará à próxima etapa, assunto do nosso próximo tópico.

2.2 Avaliação da performance base

A avaliação da performance base se dará por rodar o algoritmo obtido na implementação base e observar como a curva de crescimento do tempo de execução se comporta para inputs cada vez maiores. Isso serve 2 propósitos simples:

- Averiguar se a implementação feita foi o mais próxima possível de uma complexidade ótima (cuja negação implica na necessidade de corrigir o algoritmo);

e, certamente,

- Obter a performance que desejamos melhorar com o uso de estratégias de paralelização.

2.3 Modificação do algoritmo para paralelização

Como buscamos avaliar quanto é possível extrair de performance ao utilizar todos os cores de uma cpu moderna, temos de modificar o algoritmo base para podermos seguir com nossa avaliação.
Os algoritmos serão implementados utilizando a biblioteca "threading"da Python, exceto caso haja diminuição de performance em comparação com a biblioteca "multiprocessing". Nesse caso, a biblioteca multiprocessing será utilizada.
Segundo a documentação da biblioteca "threading"do Python, na implementação da Python em C, a biblioteca threading não realmente permite o multiprocessamento devido a um lock no interpretador. Portanto, para fins da próxima etapa, será utilizado apenas o tipo de paralelização que apresentar speedup em comparação com a performance base observada.

2.4 Avaliação da performance paralelizada

Após a paralelização do algoritmo, mediremos sua performance. A expectativa inicial é que, dado uma determinada etapa completamente paralelizável do algoritmo, digamos, por exemplo, durante um mergeSort, uma etapa de divisão ou conquista, seja possível tornar a equação (2) em algo como:

$$\frac{n}{k\zeta}\log(\frac{n}{k}) \tag{3}$$

Onde ζ indica o número de cores que estamos usando. Idealmente, todos os cores disponíveis serão utilizados, algo que depende de configuração do sistema operacional para correto gerenciamento dos processos e, muito infelizmente, está além do nosso alcance sem uma quantidade de trabalho que destruiria o ponto desse relatório.
É esperado que a performance teorizada para cada algoritmo nunca chegue perto de seu auge teorico por fatores como consumo de processamento por parte do sistema operacional, já que é inviável que esses algoritmos sejam rodados em espaço protegido.
No mais, essa avaliação será feita com base no mesmo conjunto de dados da avaliação base a fim de evitar anomalias na comparação de performance.

2.5 Comparação de performance e determinação do speedup

Após a obtenção dos dados experimentais das etapas 1 a 4, a quinta etapa consiste em comparar quanto do algoritmo foi realmente paralelizado e, logo, quanta performance conseguimos "espremer"do nosso hardware.
Essa etapa será realizada após a implementação de ambos os algoritmos e, teoricamente, pode, inclusive, ser automatizada para a possibilidade de testarmos mais rapidamente em outros hardwares, sejam eles de propriedade da dupla ou

de voluntários.

2.6 Conjunto de dados para testagem

Utilizaremos a biblioteca time para obter o tempo de execução do algoritmo e um vetor de inteiros gerado durante a execução que utilizará a biblioteca random. Os testes serão os seguintes:

1. 10 números gerados com seed 10
2. 100 números gerados com seed 100
3. 1000 números gerados com seed 1000

E assim por diante, aumentando progressivamente o conjunto de dados e a seed por um fator de 10. Caso o algoritmo demore mais que 1 minuto para executar, ele será considerado muito lento e sua execução será terminada. Portanto, 1 minuto de espera é a condição para time limit exceeded (TLE).

3 Algoritmos

Nesse tópico, serão brevemente explicados os algoritmos implementados, seu funcionamento, objetivo e estratégia de paralelização.

3.1 MergeSort

MergeSort é um algoritmo de ordenação de vetores desenvolvido inicialmente por Von Neumann, um fato peculiar e divertido. No mais, a ideia é razoavelmente simples de ser compreendida: ordene partes menores, mescle as partes ordenadas, lucre.
Esse algoritmo é, em primeira mão, um excelente candidato para qualquer um dos tipos de paralelização que buscamos utilizar, já que aparenta causa um estado de espera ou lock enquanto precisa aguardar as chamadas recursivas "filhas"terminarem, o que dificulta determinar exatamente se o processo é cpu-bound ou IO-bound, portanto, necessitando ambas as implementações e testagens.

3.1.1 Versão single-thread

É um algoritmo razoavelmente simples de ser implementado e tudo que possivelmente poderiamos dizer de interessante sobre ele já é dito na disciplina de EDA2.

3.2 Versão multithreaded

A ideia de paralelização para esse algoritmo é extremamente natural e utilizamos a versão trivial para observar um possível speedup (ou slowdown devido ao tempo de demora de criação das threads).
Tal ideia consiste em criar uma thread nova para cada chamada recursiva até um número máximo de threads. Após esse máximo, o algoritmo passa a rodar a versão normal do mergeSort.
Teoricamente, é possível deixar que o algoritmo utilize o maior número de threads necessário para ordenar o vetor, mas isso não só demora muito mais tempo como também consome uma quantidade extrema de espaço.

4 Resultados

Os resultados obtidos aqui presentes foram obtidos utilizando uma cpu Intel(R) Core(TM) i5-12500H para o algoritmo paramerge_bcl0c_threadversion.py, disponível na pasta src \mergesort.
Essa seção será dividida nas versões single e multithread.
Tempos acíma de 30 segundos serão considerados TLE. Os valores utilizados estão em segundos.

4.1 Singlethread

100	1k	10k	100k
0.0002	0.0024	0.0807	7.449

Enquanto o tempo de rodagem para 100k não demorou para o usuário precisamente 7.449 segundos, isso pode ser explicado por trocas de contexto do sistema.

4.2 Multithread I - 4 Threads

100	1k	10k	100k
0.0024	0.0043	0.0901	8.9766

Enquanto a velocidade é inicialmente comparável, pode-se observar que criar as threads para eventualmente iniciar a mesma rodagem do algoritmo mergeSort possui um custo que não exatamente se paga.

5 Conclusão

A implementação desenvolvida utilizando threads resultou em um desempenho reduzido no código, caracterizando um slowdown. Especula-se que a paralelização por GPU ou a reescrita do código com uma abordagem mais eficiente

para operações com threads poderia proporcionar melhorias na execução do algoritmo. No entanto, os testes realizados até o momento indicam que a paralelização, na configuração atual, tem contribuído para a diminuição da performance.