UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

AUTOCOMPLETAR

Aluno(s): Lucas Felix e Lucas Lagoa
Curso: Ciência da Computação
Professor: Leonardo Rocha

1.INTRODUÇÃO

A técnica de paralelização de algoritmos é um recurso que realiza a divisão de tarefas que normalmente seu algoritmo faria em uma thread para múltiplas threads, visando diminuir o tempo de ociosidade dos mesmos, fazendo-o deixar de ser um algoritmo sequencial.

2.PROBLEMA PROPOSTO

O problema proposto por esse trabalho seria o de paralelizar o trabalho prático anterior a este, que no caso foi o de simular o autocompletar de um navegador de buscas, fazendo com que múltiplas threads executem o problema juntamente, nos fazendo utilizar práticas de paralelização de programas visando otimizar o programa proposto posteriormente.

3.SOLUÇÕES IMPLEMENTADAS

Foram apresentadas duas soluções para o problema proposto, sendo que uma é um algoritmo de força bruta paralelo e a outra e um algoritmo heurístico exato também implementado para a utilização de threads. Nesta seção será explicado como cada um dos programas fora implementado, contudo com um foco maior na parte paralelizada do algoritmo e suas diferenças com o algoritmo serial.

3.1-FORÇA BRUTA

O algoritmo de força bruta basicamente compara todos os caracteres a fim de encontrar padrões. Entretanto há um interesse maior com a relação apenas aos prefixos das palavras, a parte inicial da palavra que está presente no dicionário deve ser igual ao prefixo digitado, batendo todos os caracteres até o tamanho da palavra pesquisada pelo usuário.

Um exemplo seria pesquisar a palavra “bat”, possíveis retornos seriam Batman, batata, bate papo, entre outros que tem um começo igual à bat.

3.1.2-IMPLEMENTAÇÃO

Para começo temos a leitura dos parâmetros do programa, logo ai já temos uma diferença, pois o programa serial não necessitava da passagem do parâmetro que é o número de threads que serão utilizadas, já no algoritmo paralelo, é oferecido ao usuário a opção de escolher quantas threads o mesmo deseja que sejam executadas.

Apesar de termos como opção a escolha do número de threads saiba que as mesmas são limitadas e variam de quantidade de acordo com a arquitetura de cada computador, contudo mesmo colocando o número de threads maior que o número das mesmas no computador, o sistema operacional consegue manejar tal problema com bastante precisão utilizando o escalonador de processos, porém, este assunto não será tão explorado, pois o trabalho é mais focado na parte paralela do autocompletar.

Depois de feita a passagem dos parâmetros o programa faz a leitura do arquivo, armazenando o dicionário onde estão as palavras que serão comparadas.

Após a leitura do arquivo é feita uma série de inspeções que para garantir que o usuário não erre ao passar os parâmetros, por exemplo, caso ele peça que retorne o número de palavras igual a zero ou que o número de palavras que ele peça que retorne seja maior que o número de palavras do dicionário. Caso não entre em nenhuma dessas condições o programa agora faz maior parte do trabalho.

Primeiramente temos uma função que verifica se todas as letras que o usuário digitou são minúsculas, caso não seja, a função deixa as mesmas minúsculas para que haja um padrão nas palavras digitadas pelo usuário.

Nesta parte, antes de entrarmos na explicação dela, temos que deixar claro que a parte que foi paralelizada no trabalho, foi a parte que tinha uma complexidade maior e a qual demorava mais em ambos algoritmos, que é a parte da busca das palavras. Como a palavra digitada pelo usuário deve comparar no mínimo um caractere com cada palavra do dicionário, tal processo exigia uma demanda de tempo muito grande, com isso, utilizando as threads podemos fazer uma subdivisão de trabalhos, colocando cada thread para operar com parte do arquivo. E ainda antes de explicar o algoritmo já paralelizado temos que definir como foram passados para as threads qual parte cada uma iria executar para que não houvesse uma redundância no trabalho realizado pelas threads. Então para começar devemos lembrar que essa parte não está presente no programa seqüencial, e tal modificação é essencial para um bom funcionamento do algoritmo paralelo.

Para que cada thread saiba qual parte será responsável primeiramente fazemos a divisão do número de palavras pelo número de threads, para sabermos quantas palavras cada thread irá receber e somamos o resto para a última thread. Ou seja, a última thread sempre irá trabalhar mais que as demais threads.

Depois de feita a divisão de quantidade, resta saber qual parte cada thread irá receber, e para isso utiliza-se a seguinte fórmula:

$$Inicio=(div*i)$$

$$Final=(div*i)+inicio-1$$

Apenas a última thread que tem a segunda equação diferente, sendo ela:

$$Final=(div*i)+inicio+resto-1$$

Assim, cada thread irá saber qual será seu início e fim, e serão disparadas com uma pequena variação de tempo, que é quase imperceptível.

Neste momento, começa a função de busca que foi paralelizada, onde antigamente apenas uma parte do processador trabalhava, agora temos várias threads dividindo o

trabalho para que haja um melhor desempenho do algoritmo. A busca paralela se dá basicamente da seguinte forma:

Uma thread compara a palavra digitada pelo usuário com uma certa quantidade de palavras do dicionário, enquanto outras threads comparam com as outras partes. Como exemplo para melhor explicar o funcionamento do algoritmo, tomemos como base o seguinte dicionário:

Palavra 0:	Batman
Palavra 1:	Célula
Palavra 2:	Falha
Palavra 3:	Batata

Tabela 1: A tabela mostra basicamente um exemplo de dicionário

Considerando que a palavra digitada pelo usuário seja “bat” e que o número de threads seja igual a dois, então uma thread pegaria as posições 0 e 1 do dicionário e a segunda thread pegaria as posições 2 e 3. Ficando da seguinte forma.

Primeira thread:	Batman	Célula
Segunda thread:	Falha	Batata

Tabela 2: Mostra basicamente qual parte do dicionário cada thread irá pegar

Depois disso é feita a pesquisa, a mesma é realizada da seguinte forma:

Compara-se do primeiro até o último caractere do tamanho do padrão, ou seja, se o padrão possui 3 caracteres, como é nosso caso, ele fará três comparações, se o padrão possui 4 caracteres, fará quatro comparações e assim por diante.

Primeira thread:							Segunda thread						
Ordem de comparação:	1º	2º	3º				Ordem de comparação:	1º	2º	3º			
Palavra Digitada:	B	A	T				Palavra digitada:	B	A	T			
Palavra 0:	B	A	T	M	A	N	Palavra 2:	F	A	L	H	A	
Resultado:	C	C	C				Resultado:	E	C	E			
Palavra 1:	C	É	L	U	L	A	Palavra 3:	B	A	T	A	T	A
Resultado	E	E	E				Resultado:	C	C	C			

Tabela 3: Mostra basicamente como cada thread irá trabalhar, sendo “c” caracteres que estão corretos com o padrão e “e” os caracteres que estão incorretos

As pesquisas são feitas de forma independente, assim, logo depois de feita a seleção das palavras qualificadas, que possuem o prefixo que o padrão digitado pelo usuário, as

mesmas devem ser ordenadas para ser impressas as k primeiras possíveis pesquisas para o usuário.

Para ordenação foi utilizado o famoso algoritmo de ordenação bubblesort, que basicamente verifica os elementos adjacentes a ele trocando os mesmos de posição quando estiverem fora de ordem. Após ordenado, o algoritmo apenas faz a impressão das k primeiras palavras com prefixo ao padrão já ordenado.

3.2-HEURÍSTICO

O segundo algoritmo implementado é uma heurística exata baseada no algoritmo de processamento de caracteres KMP (Knuth-Morris-Pratt). O algoritmo diferente do força bruta, faz comparações de trás pra frente (assim como o KMP), evitando comparações desnecessárias quando não acha casamento de caracteres. Para esse algoritmo foi paralelizada também a busca, que é a parte que exige mais processamento, e ao paralelizar podemos ver uma melhora significativa no desempenho do algoritmo.

3.2.1-IMPLEMENTAÇÃO

Como já dito o algoritmo heurístico é baseado no KMP e seu começo é basicamente igual ao força bruta, fazendo a função min, calculando a quantidade de valores que cada thread irá receber e já partindo para a busca

Depois do começo basicamente vamos pra função de busca, que ao contrário da busca no força bruta tem sua comparação de trás para frente, e caso ache um não casamento de caractere, o mesmo para de fazer as comparações. O quadro abaixo mostra basicamente como o algoritmo heurístico irá trabalhar:

Primeira thread:							Segunda thread:					
Ordem de comparação:	3º	2º	1º				Ordem de comparação:	3º	2º	1º		
Palavra pesquisada:	P	A	R				Palavra pesquisada:	P	A	R		
Primeira palavra:	P	A	R	A	D	A	Primeira palavra:	P	A	R	I	S
Resultado:	C	C	C				Resultado:	C	C	C		
Segunda palavra:	P	A	U	L	O		Segunda palavra:	T	A	N	T	O
Resultado:	N.C.	N.C.	E				Resultado:	N.C	N.C.	E		

Tabela 4: Essa tabela mostra basicamente o funcionamento da pesquisa no algoritmo heurístico

O algoritmo quando encontrar um caractere que não bate com o caractere naquela posição no padrão não efetuará mais buscas.

Depois de feitas as buscas, lembrando mais uma vez que cada thread irá agir de forma independente, são salvas posições de memória de cada uma das palavras que possuem os caracteres iguais ao do prefixo. Após termos as posições das palavras que desejamos, ordenamos seus pesos, utilizando o bubblesort, assim como no força bruta e são impressas as k primeiras palavras.

4.ANÁLISE DE COMPLEXIDADE

Na análise de complexidade será discutida a complexidade dos algoritmos e se ele mudou do algoritmo serial para o paralelo e caso tenha mudado como mudou.

4.1-FORÇA BRUTA

Primeiramente, o algoritmo de força bruta faz a leitura do arquivo, a leitura do mesmo ocorre em duas estruturas de repetição aninhadas até o final do arquivo, sendo que estas estruturas variam de zero até o número total de palavras, logo a complexidade da leitura é O(número de palavras*número de palavras).

Depois de feita a leitura, temos utiliza-se a função strlen (que retorna o tamanho da palavra, tal função é encontrada na biblioteca string.h) que retorna o tamanho da string, a complexidade da função é O(tamanho), sendo tamanho a quantidade de caracteres presentes na palavra digitada pelo usuário. Também temos a função min, responsável por deixar a palavra digitada pelo usuário minúsculas, a função tem basicamente uma estrutura de repetição com uma condicional aninhada, o loop varia de zero até o tamanho da palavra digitada, sendo assim a complexidade é O(tamanho da palavra digitada*2).

Após sabermos o tamanho da palavra digitada pelo usuário, temos que dividir os valores que cada thread irá receber para isso é utilizado uma estrutura de repetição e dentro do mesmo temos uma série de condicionais que verificam qual será o inicio e qual será o fim de cada thread, como este varia de zero até número de threads a complexidade desta função será O(número de threads*2).

Após feito isso são disparadas as threads que irão trabalhar sobre a função de busca, a complexidade da função de busca é dada por duas estruturas de repetição aninhadas, sendo que uma percorre de zero até o número de palavras e a segunda percorre de zero até o tamanho do padrão digitado pelo usuário, dentro dos loops possui-se duas condicionais, uma que compara os caracteres e outro que verifica que se o contador é do tamanho da palavra digitada, caso seja temos um terceiro loop que vai de 0 até tamanho, sendo este tamanho o número da maior palavra. Logo a complexidade final desta função é: O(número de palavras/número de threads*tamanho do padrão*2*2*tamanho), sendo assim a complexidade final O(4*número de palavras*tamanho do padrão*tamanho).

Depois de feita a pesquisa e retornada às palavras da pesquisa, as mesmas passam pela função vetor maiores, que percorre de zero até número de palavras fazendo com que um

vetor receba o peso das palavras, e este vetor logo é ordenado pelo algoritmo bubblesort, sendo seu tempo no pior caso O(número de palavras selecionadas ^{2}), sendo assim a complexidade final da função é O(número de palavras selecionadas ^{2}+número de palavras).

Para finalizar temos a função imprime, que possui duas estruturas de repetição aninhadas, sendo que uma percorre de zero até a quantidade de palavras que serão impressas, e a segunda estrutura condicional que vai de zero até o número de palavras, e dentro dos mesmo ainda há uma condicional que verifica se o vetor com os pesos é igual ao peso da palavra selecionada, caso seja imprime-se a palavra, logo a complexidade final da função é O(quantidade de palavras que serão impressas*número de palavras*2).

Logo a complexidade final do algoritmo é O(max(O(quantidade de palavras que serão impressas*número de palavras*2)+O(número de palavras+número de palavras selecionadas ^{2})+O(4*número de palavras/número de threads*tamanho do padrão*tamanho)+O(número de threads*2)+O(tamanho da palavra)+O(número de palavras*número de palavras)+O(tamanho da palavra digitada*2)), sendo assim a complexidade final do algoritmo é O(número de palavras*número de palavras).

4.2-HEURÍSTICO

O algoritmo heurístico tem um começo basicamente igual ao do força bruta, possuindo uma leitura com complexidade O(número de palavras*número de palavras). Sua única diferença no começo está depois da leitura onde se preenche um vetor com o tamanho do número de palavras com -1, tal vetor será usado para armazenar a posição das palavras selecionadas, sendo assim sua complexidade é O(número de palavras). Depois disso o algoritmo heurístico tem seu resto, até a função de pesquisa igual ao algoritmo de força bruta, utilizando a função strlen, que possui complexidade O(tamanho da palavra), depois a função min que possui complexidade igual O(2*tamanho da palavra), seguindo para os cálculos de posição da thread que possuem a complexidade igual a O(número de palavras*2).

Depois de feito este começo praticamente igual ao força bruta temos a função de busca, que possui duas estruturas de repetição aninhadas sendo que uma vai de zero até número de palavras/número de threads, e a segunda que vai do tamanho da palavra do usuário até zero, decrementando, e ainda dentro do mesmo possui-se duas condicionais, uma responsável por verificar se o caractere da palavra é igual ao caractere do dicionário e outro que verifica se o contador é igual ao tamanho da palavra digitada pelo usuário. Caso seja, um vetor de palavras selecionadas (aquele que recebeu -1 no inicio) agora recebe as posições de memória das palavras que foram selecionadas. Logo a complexidade final da função é O(número de palavras/número de threads*tamanho da palavra do usuário*4).

Após feita seleção das palavras temos a função ordena que possui uma estrutura de repetição e uma estrutura condicional, sendo que a estrutura de repetição varia de zero

até número de palavras, e a estrutura condicional é responsável por armazenar no vetor peso, o peso das palavras que foi selecionada. No algoritmo serial, tal função tinha complexidade igual a O(2*número de palavras retornadas na busca), já aqui tem complexidade igual O(2*número de palavras), ou seja, apenas no pior caso, onde todas as palavras do dicionário serão retornadas, a complexidade das duas funções é igual, e assim a complexidade do algoritmo heurístico paralelizado é maior que a complexidade do mesmo algoritmo não paralelizado.

Por fim, as palavras são ordenadas utilizando o algoritmo de ordenação bubblesort que tem complexidade no pior caso de O(número de palavras selecionadas ^{2}), e por fim temos a impressão das palavras, a função imprime possui duas estruturas de repetição, sendo que uma varia de zero até o valor de palavras que serão impressas e a outra varia de zero até o número de palavras, possuindo dentro dos mesmos duas estruturas condicionais, que verificam a palavra que será impressa. Logo a complexidade final da função é O(número de palavras que serão impressas*número de palavras*4).

Portanto a complexidade final do algoritmo é do algoritmo é O(max(O(número de palavras que serão impressas*número de palavras*4)+O(número de palavras selecionadas ^{2})+O(2*número de palavras)+O(2*número de palavras retornadas na busca)+O((número de palavras/número de threads)*tamanho de palavras do usuário*4)+O(tamanho da palavra)+O(2*tamanho da palavra)+O(número de palavras)+O(número de palavras*2)+O(número de palavras ^{2})), ou simplesmente O(número de palavras ^{2}).

5.ANÁLISE DE RESULTADOS E TESTES

Na análise de resultados e testes serão comparados o algoritmo heurístico e força bruta, tanto seriais como também os paralelos, mostrando as vantagens dos paralelos em relação aos algoritmos seriais, e como mesmo paralelizado o algoritmo de força bruta consegue ser infinitamente pior quanto comparado ao algoritmo heurístico, apesar de uma pequena melhora.

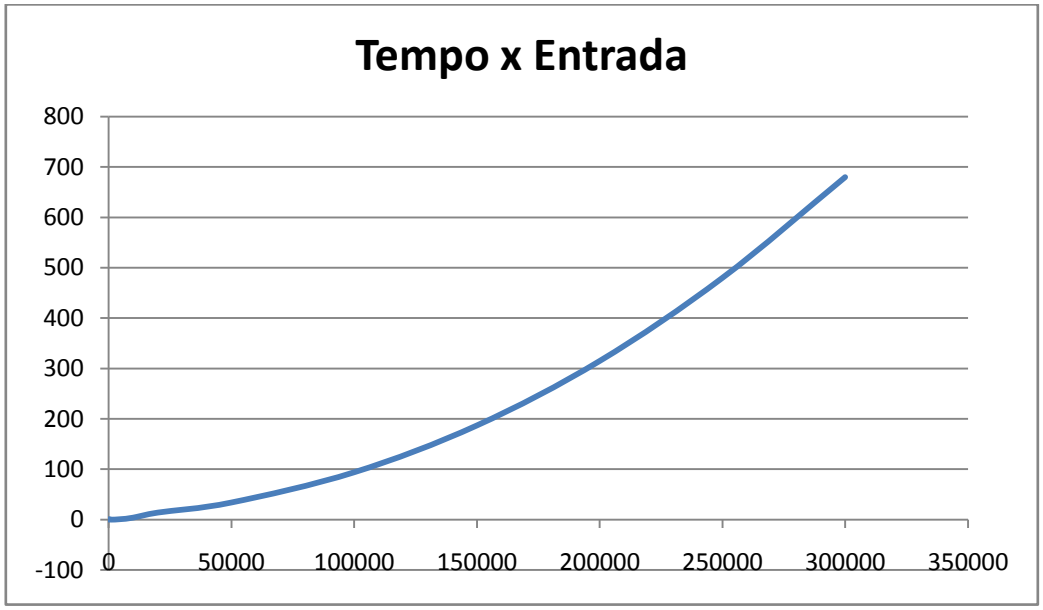
	Utilizando uma tabela ordenada com prefixo=1 e retorno=3, para os algoritmos com paralelos número de threads = 2			
Número de palavras:	Heurístico (Paralelo)	Heurístico (Serial)	Força Bruta (Paralelo)	Força Bruta (Serial)
10	0 m/s	0 m/s	0 m/s	0 m/s
20	0 m/s	0 m/s	0 m/s	0 m/s
50	0 m/s	0 m/s	0 m/s	0 m/s
100	0 m/s	0 m/s	0 m/s	0 m/s
500	1 m/s	0 m/s	2 m/s	2 m/s
1000	0 m/s	0 m/s	6 m/s	6 m/s
5000	1 m/s	3 m/s	63 m/s	52 m/s
10000	4 m/s	10 m/s	210 m/s	217 m/s
20000	14 m/s	11 m/s	880 m/s	856 m/s
50000	34 m/s	29 m/s	5617 m/s	5513 m/s
100000	94 m/s	85 m/s	22050 m/s	22315 m/s
150000	187 m/s	158 m/s	49103 m/s	49494 m/s
200000	315 m/s	263 m/s	88754 m/s	82719 m/s
250000	480 m/s	392 m/s	135823 m/s	195946 m/s
300000	680 m/s	561 m/s	199385 m/s	226906 m/s

Tabela 5: A tabela 5 mostra basicamente o tempo gasto pelos algoritmos paralelos e seriais para uma mesma base de entrada

Podemos perceber que o algoritmo heurístico paralelo possui um tempo igual e às vezes até inferior ao algoritmo serial, o que nos faz pensar se o algoritmo paralelo realmente é melhor, mas vendo o força bruta, vemos uma melhora significativa no último caso.

Como dito acima, o algoritmo heurístico paralelo possui um desempenho igual e às vezes até pior que o mesmo algoritmo serial, isso se dá pelo aumento da complexidade do algoritmo, a função ordena que antes possui os valores certos da quantidade de palavras e armazenava elas em posições adjacentes, agora armazena as palavras pelas posições, ou seja, na hora de ordenar deve-se procurar em todas as posições do dicionário, o que faz com que o algoritmo tenha um desempenho inferior. Caso tivéssemos considerado apenas a pesquisa, o algoritmo paralelo conseguiria ter um desempenho muito superior ao mesmo algoritmo serial, contudo precisamos das

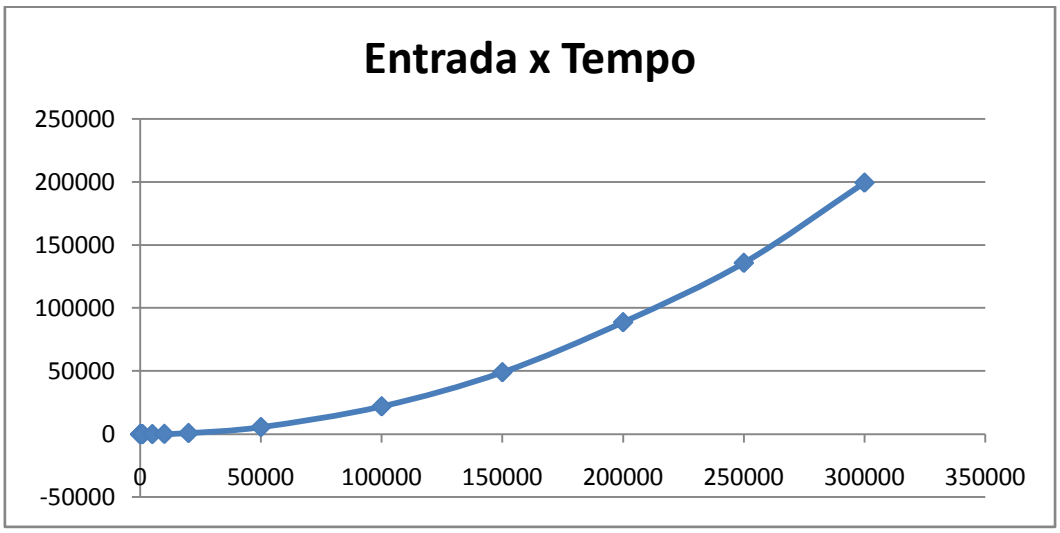
palavras já ordenadas, o que faz com que o algoritmo paralelo não tenha um desempenho melhor que o serial.



1ª tabela: Essa tabela define basicamente o comportamento do algoritmo heurístico

Podemos ver pela tabela acima que o comportamento do algoritmo heurístico é basicamente dado pela quantidade de entrada, quanto maior a mesma, maior será o tempo gasto.

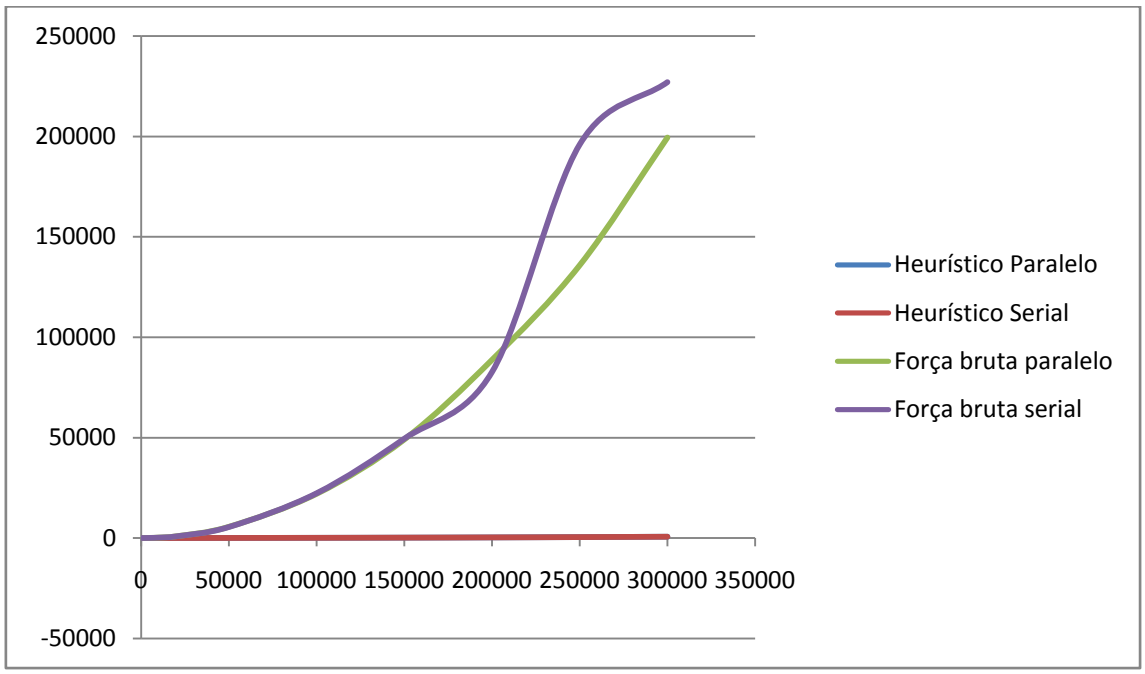
O algoritmo força bruta como vimos acima é um algoritmo que teve o desempenho melhorado graças ao paralelismo, o gráfico abaixo demonstra basicamente isso:



2º gráfico: Define basicamente o comportamento do algoritmo força bruta

Podemos perceber que mesmo paralelizado o algoritmo força bruta, que teve o desempenho aumentado, consegue ter ainda um desempenho muito inferior ao

algoritmo heurístico e mesmo apresentando um resultado tão correto quanto a heurística baseada no KMP, o algoritmo força bruta não é recomendado para este caso.



3º gráfico: O gráfico acima mostra o comportamento dos quatro algoritmos sobrepostos

O gráfico acima mostra basicamente o comportamento dos quatro algoritmos, podemos perceber que não vemos o comportamento do algoritmo heurístico paralelo, pois o mesmo é basicamente igual ao serial. Observando o gráfico também podemos ver que o algoritmo força bruta paralelo possui um desempenho bem melhor que ao serial, entretanto ainda não é um algoritmo com um tempo aceitável.

Neste momento será apresentado o speedup dos algoritmos que é o fator de aceleração, que é a medida de quantas vezes mais rápido o algoritmo paralelo é em relação ao algoritmo serial.A equação que define o speedup é basicamente a seguinte:

SpeedUp=Tempo de execução na versão seqüencial / Tempo de execução na versão paralela

Calculando o speedup para o algoritmo de força bruta com base de entrada igual a 300 mil palavras e para o algoritmo paralelo a utilização de duas threads:

SpeedUp=226906/199385= 1,1380

Vemos que o algoritmo de força bruta paralelo é 1,13 vezes mais rápido que o mesmo algoritmo serial, isso pode ser visto através de gráficos e tabelas, o algoritmo paralelo é conseguiu ser bem superior quando comparado ao serial.

Calculando agora o speedup para o algoritmo heurístico com base de entrada igual a 300 mil palavras e para o algoritmo paralelo utilizando duas threads:

SpeedUp=561/680=0,825

Vemos que o algoritmo paralelo neste caso não é nem mesmo mais rápido, como dito mais acima isso se dá pelo aumento da complexidade do algoritmo, que faz com que o tenha-se um desempenho menor. Mas não apenas isso, o overhead gerado pelo processador ao se fazer o paralelismo não compensa o custo de se paralelizar o mesmo, além disso a função mais indica para ser paralelizada, a que mais demandava tempo, era a função de ordenação, contudo ao fazermos um estudo, para paralelizar a mesma iria precisarmos de blocos já ordenados para fazer a ordenação paralela de forma correta, pois ao tentarmos paralelizar a mesma, não conseguimos tais blocos, o que fez com que optássemos por paralelizar a busca.

Antes de finalizar a seção é bom deixar claro que mesmo com a utilização de mais threads os algoritmos, apesar de se fazer uma busca muito mais rápida, têm que esperar as mesmas acabarem de trabalhar para depois fazer a ordenação, o que acaba deixando o algoritmo com um desempenho inferior ao esperado. Uma solução proposta para o problema seria paralelizar o resto do algoritmo, contudo iríamos travar na ordenação, pois a mesma depende de todos os dados para deixar os mesmo ordenados de modo que os mesmos possam ser mostrados ao usuário.

6.INTERFACE GŔAFICA

Visando um sistema de integração eficiente que se pareceria com um “auto-complete” de um buscador, a interface gráfica foi implementada em HTML e PHP. No arquivo PHP, contém a integração do HTML com o código, sendo que ela recebe exatamente como é executado no terminal, e assim a cada palavra digitada na interface gráfica, o algoritmo compila e assim mostra as palavras abaixo. Vale a pena destacar que para a interface integrar com o algoritmo, foi necessária uma mudança no código, e essa mudança foi para simular uma lista de um buscador, a cada palavra que foi selecionada, é colocado um caracter especial depois dela, e assim quando a interface detectar esse caracter especial, ela quebraria a linha fazendo com queas palavras saiam em formato de lista. E assim, para visualizar o algoritmo integrado com a interface gráfica, deve seguir uma série de passos:

https://www.apachefriends.org/pt_br/download.html

Devemos primeiramente instalar o XAMPP que pode ser encontrado no endereço abaixo.

E deve-se escolher de acordo com a versão do Linux no computador a ser executado oscódigos. Logo após de realizado o download, deve-se instalar o arquivo baixado. (Lembre-se de colocar as permissões de administrador, e de executar em Propriedades>Permissões>executar).

Logo após a instalação do programa, deve-se primeiramente rodar o programa com o seguinte comando no terminal:

sudo /opt/lampp/lampp start

E assim, o programa estará executando. Para finalizar, deve-se colocar as pastas com asimplementações no seguinte caminho:

/opt/lampp/htdocs

Nesse caminho o restante a se fazer é executar o programa dentro desse caminho e paravisualizar ele deve-se digitar no navegador:

localhost/"Nome da pasta que contém o programa"

Exemplo: localhost/Heuristico

E assim é possível usar a interface gráfica.

7.CONCLUSÃO

A implementação do trabalho ocorreu em partes sem muitos problemas, porém alguns tópicos merecem atenção à serem debatidos. Na tentativa de paralelizar o algoritmo, houve um aumento na complexidade do mesmo, e esse aumento da complexidade gerou um mínimo aumento no tempo, porém esse tempo não é tão diferenciável da implementação anterior. Outro tópico que merece destaque a ser debatido, é em relação à parte gráfica, que da antiga implementação para essa, não houve alteração da mesma.

8.REFERÊNCIAS BIBLIOGRÁFICAS

- ROCHA, Leonardo C. D. Slides da unidade curricular de Algoritmos e Estruturas de Dados III, 2016.