ORDENAÇÃO NUMÉRICA UTILIZANDO THREADS COM ALGORITMO MERGE SORT

João Vitor Pogiolli do Lago - 158049

Limeira, Novembro de 2019

Conteúdo

Limeira, Novembro de 2019

1. Introdução

Atualmente a necessidade de desenvolver softwares com capacidade de processamento multi-thread torna-se inevitável. Devido ao avanço dos recursos computacionais atuais, saber desenvolver e projetar corretamente um software para aproveitar tais recursos é de fundamental.

2. Objetivo

Este trabalho tem o objetivo de criar um algoritmo para ordenação de um vetor inteiro de *N* elementos, obtidos através da leitura de *M* arquivos. A ordenação, por sua vez, ocorrerá de forma paralela utilizando threads, com isso, o algoritmo deverá ser capaz de distribuir os *N* elementos a serem ordenados preferencialmente igualmente entre as threads. Para a ordenação proposta, realizaremos os testes para 2, 4, 8 e 16 threads de ordenação.Todo o código fonte, arquivos de entrada e saída, arquivo de teste
podem ser encontrados no repositório GIT do grupo:

https://github.com/Poggioli/neuron

3. Software

O software proposto consiste na criação de um número *T* de threads de execução para a ordenação, sendo o valor *T* determinado pelo usuário. Com base no número de threads, nos arquivos informados pelo usuário e no total de elementos lidos dos arquivos, o software irá distribuir para cada thread um número *X* de elementos, tentando sempre distribuir um número igualitário de elementos para todas as threads do sistema.

Para a criação de threads foi utilizada a biblioteca ‘pthread’, a qual implementa e controla todas as threads do software. Para controle das informações passadas para as threads, utiliza-se uma estrutura pré definida no inicio do código, contendo as informações necessárias para a ordenação por cada thread.

3.1. Cálculo do Número de Elementos para cada Thread

Para o software proposto, o cálculo do número de elementos para cada thread para cada uma das *T* threads é feito através de duas expressões matemáticas para determinar o intervalo inferior e superior no vetor de números a serem ordenados.

$$posicaoInferior = ID * (N / T)$$

$$posicaoSuperior = (ID + 1) * (N / T) - 1$$

onde:

ID ID da thread. Valores entre [0, T].
T Número total de threads
N Número de elementos para ordenação

Em uma rodada de ordenação onde o usuário tenha entrado com um número de threads igual a 4 e um número total de 1000 elementos para ordenação, temos os seguintes valores para as posições inferiores e superiores de cada thread.

Tabela 1 - Valores das posições inferiores e superiores para cada thread

Thread	ID	posicaoInferior	posicaoSupeior
1	0	0	249
2	1	250	499
3	2	500	749
4	3	750	999

3.2. Definições TIME_T e CLOCK_T

Para o cálculo do tempo de uso do processador pelo algoritmo proposto e o tempo total de execução, foram utilizadas as definições clock_t e time_t respectivamente. Ambas são instanciadas no começo do algoritmo (linhas 72 e 73) e tem sua definição de tempo inicial um comando antes das threads serem criadas, e tem sua finalização (linhas 121 e 122) no primeiro comando após todas as threads terem executado suas operações, como vemos na Figura 1 a seguir.

Figura 1

4. Testes e Validação Multi-Threads

Para validação do código desenvolvido, utilizamos como plataforma de teste um Macbook Pro 2012 com as seguintes configurações da Figura 2:

Figura 2

Para uma validação correta de desempenho foi rodado o algoritmo para todos os casos de 2, 4, 8 e 16 threads, utilizando 5 arquivos de entrada com 10.000 (dez mil) números aleatórios cada de 1 até 100.000 (cem mil). A seguir temos exemplos da linha de execução:

(1) ./mergesortMultiThread 2 file1.txt file2.txt file3.txt file4.txt file5.txt output.txt

(2) ./mergesortMultiThread 4 file1.txt file2.txt file3.txt file4.txt file5.txt output.txt

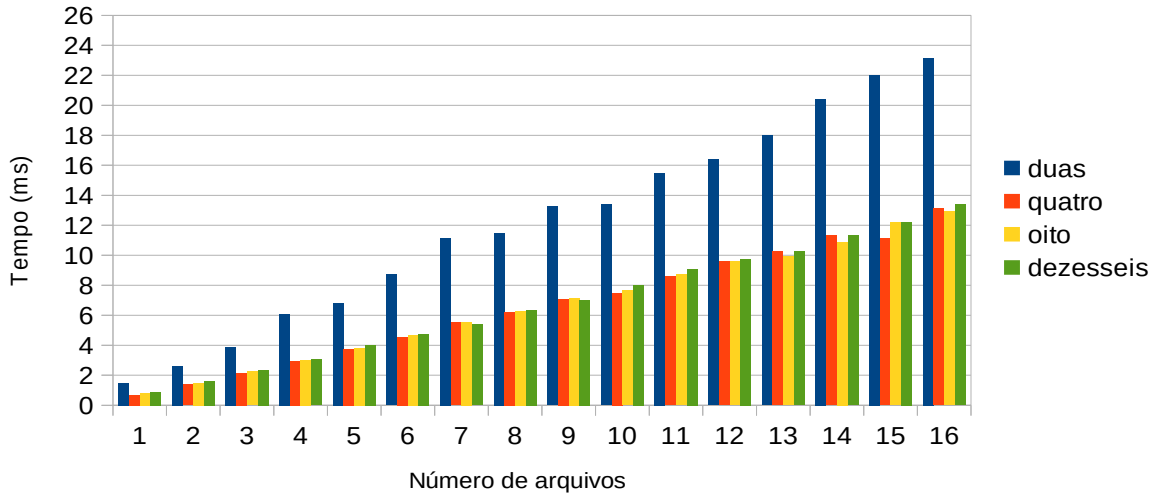
(3) ./mergesortMultiThread 8 file1.txt file2.txt file3.txt file4.txt file5.txt output.txt

(4) ./mergesortMultiThread 16 file1.txt file2.txt file3.txt file4.txt file5.txt output.txt

Para a validação final dos testes, rodamos uma bateria de 3 testes do algoritmo, variando o número de threads entre 2, 4, 8 ou 16. Lembramos que o tempo medido é o tempo de uso do processador. Para 2 threads devemos dividir o tempo por 2 para encontrar o tempo real de execução, e para 4, 8 e 16 threads devemos dividir o tempo por 4, visto que esse é o numero máximo de threads de execução do nosso processador, os tempos estão normalizados em milissegundos. O gráfico a seguir diz respeito à primeira bateria de testes.

Arquivos	Bateria de teste 1				Bateria de teste 2				Bateria de teste 3			
	Threads				Threads				Threads			
	2	4	8	16	2	4	8	16	2	4	8	16
1	1,482	0,688	0,81	0,861	1,378	0,688	0,792	0,869	1,487	0,739	0,8	0,904
2	2,634	1,39	1,455	1,62	2,844	1,497	1,453	1,671	2,934	1,396	1,561	1,598
3	3,868	2,156	2,251	2,368	4,247	2,097	2,293	2,323	4,476	2,182	2,274	2,4
4	6,109	2,947	3,018	3,055	5,974	3,048	2,903	3,12	5,621	2,966	2,891	3,163
5	6,798	3,738	3,814	4,009	7,347	3,613	3,835	3,873	7,524	3,764	3,871	3,814
6	8,715	4,565	4,679	4,762	9,007	4,657	4,569	4,725	8,643	4,281	4,579	4,725
7	11,16	5,513	5,553	5,376	9,567	5,318	5,478	5,387	9,513	5,186	5,403	5,513
8	11,466	6,234	6,281	6,333	12,146	6,318	6,162	6,293	10,455	6,154	6,409	6,25
9	13,249	7,082	7,139	7,002	13,662	6,973	7,032	7,404	12,137	7,164	6,937	7,177
10	13,383	7,504	7,665	8,038	14,984	7,941	7,746	8,052	13,889	7,934	7,635	7,966
11	15,468	8,595	8,712	9,093	14,783	8,679	8,868	8,653	15,45	8,844	8,632	8,91
12	16,399	9,583	9,593	9,77	19,294	9,115	9,438	9,551	19,145	9,305	9,333	9,638
13	17,982	10,261	9,914	10,247	20,254	9,898	10,109	10,281	20,681	9,866	10,555	10,237
14	20,436	11,312	10,892	11,309	22,038	11,418	11,447	11,087	22,708	11,414	11,107	11,325
15	22,03	11,123	12,182	12,213	24,673	11,634	12,054	11,966	22,541	11,893	12,181	12,378
16	23,162	13,121	12,932	13,388	23,714	12,53	12,808	13,321	24,352	13,315	13,351	13,187

Gráfico de tempo



Limeira, Novembro de 2019

5. Vídeo

https://www.youtube.com/watch?v=HdiIFFd058Y

6. Conclusão

Ao analizarmos os gráficos de duração dos testes, vemos claramente que o algoritmo se comporta mais rápido com 4 threads para execução, o que é exatamente igual ao número de threads disponíveis no processador. Quando rodamos o teste para 8 ou 16 threads, elas se comportam semelhante ao teste de 4 threads, porém com um pouco mais de tempo devido ao gerenciamento de todas as threads.

Ao analizarmos a duração do algoritmo para 2 threads de execução, vemos que o tempo é aproximadamente 100% maior do que o teste com 4 threads quando colocamos um grande volume de dados. Isso confronta exatamente com o resultado esperado, pois ao utilizarmos todos os núcleos de processamento do processador a aplicação se torna mais eficiente.

Portanto, quando desenvolvemos um sistema, seja ele específico ou não, devemos levar em conta uma análise do hardware no qual o sistema será executado. Não é de interesse desenvolver sistemas multi-thread complexos para hardware onde existem poucos núcleos de execução, como processadores single core ou dual core. Entretanto, é extremamente válido a criação de mecanismos multi-thread quando lidamos com máquinas de alto poder computacional, como servidores.