Analisando o Desempenho da Paralelização no Algoritmo de Ordenação *Mergesort In-place*

Adriano M. Garcia, Márcia C. Cera, Sérgio L. S. Mergen

Universidade Federal do Pampa (UNIPAMPA) – Campus Alegrete
Av. Tiarajú, 810 – CEP 97546-550 – Alegrete – RS – Brasil

adriano1mg@gmail.com,{marciacera, sergiomergen}@unipampa.edu.br

Resumo. *Algoritmos de ordenação são amplamente utilizados e estão presentes em várias aplicações. Em nosso trabalho, realizamos a paralelização de uma implementação in-place do algoritmo Mergesort com o objetivo de reduzir seu tempo de execução. A linguagem de programação utilizada é Java e utilizamos threads para a paralelização. Os objetivos iniciais foram atingidos, com redução de até 74% no tempo de execução.*

1. Introdução

Algoritmos de ordenação são algoritmos que implementam operações capazes de ordenar sequências de dados recebidas como entrada [CORMEN *et al*. 2002]. Existe uma série de aplicações que necessitam de dados ordenados. O resultado das pesquisas em sites de busca, por exemplo, são mostrados aos usuários ordenados de acordo com algum parâmetro, tais como importância ou interesses.

Há diversos tipos de algoritmos de ordenação, sendo que cada um utiliza métodos de ordenação diferentes. O desempenho deles varia em função da entrada de dados, que pode diferir em relação ao tamanho, tipo de dados, organização etc. Um desses algoritmos é o *Mergesort*, que possui um bom desempenho trabalhando com entradas grandes [SZWARCFITER e MARKENZON, 1994]. Trata-se de um algoritmo recursivo que implementa o método de divisão e conquista para ordenar. Sua complexidade é $\Theta(n \log n)$ para todos os casos [CORMEN *et al*. 2002].

Embora existam diversos métodos de ordenação, há um problema que a maioria deles enfrenta ao trabalhar com grandes volumes de dados: o consumo de memória. Para resolver esse problema, contamos com os algoritmos *in-place*, que utilizam uma quantia constante de memória durante sua execução. Nosso objetivo neste trabalho é diminuir o tempo de execução do *Mergesort* em uma versão *in-place.*

Para atingir esse objetivo, exploramos a técnica de paralelização. Essa técnica permite que vários trechos de um algoritmo executem ao mesmo tempo, em paralelo. Ela se torna mais eficaz ao trabalharmos com a tecnologia multinúcleo, a mesma que dispomos na maior parte das máquinas atuais, onde podemos dividir os trechos paralelos em relação à quantia de núcleos de um processador.

Neste artigo apresentaremos nosso trabalho de paralelização e seu desenvolvimento na Seção 2. A organização dos testes e o ambiente de execução serão mostrados na Seção 3. Na seção 4 mostraremos os resultados obtidos e finalizaremos com nossas conclusões na sessão 5.

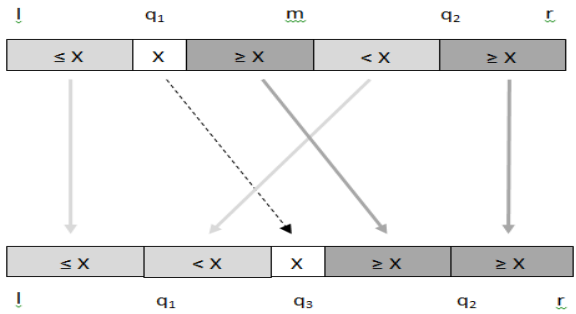

Figura 1: Funcionamento *swapping block* da mescla *in-place* [DUVANENKO 2012].

2. Paralelização do *Mergesort In-place*

O algoritmo de *Mergesort* é dividido em duas etapas. A primeira divide o vetor de entrada em trechos menores subsequentes, enquanto a segunda mescla esses trechos. Na etapa de mescla, assume-se que os trechos já estão previamente ordenados. Diversas abordagens existem para realizar a mescla desses trechos ordenados, sendo que a abordagem apresentada neste artigo aparece ilustrada na Figura 1.

A abordagem apresentada na figura faz a mescla de forma in-place. De modo geral, é encontrado o elemento intermediário do primeiro trecho (elemento **X**, na posição **q1**). Em seguida, é procurando o primeiro elemento do segundo trecho que é superior ao elemento selecionado (posição **q2**). O próximo passo envolve mover todos os elementos do segundo trecho que aparecem antes de **q2** para antes de **q1**. Ao final desse passo, o elemento **X** já estará na posição correta. Os elementos à sua esquerda estarão divididos em dois trechos ordenados, assim como os elementos à sua direita. Ou seja, manteve-se o problema original, mas com trechos de tamanho menor. Isso permite que a mesma abordagem seja aplicada recursivamente para ordenar todo o vetor [DUVANENKO, 2012].

Após analisarmos o funcionamento do algoritmo, decidimos paralelizar a partir do ponto em que ocorre a separação dos trechos menores do que **X** e maiores do que **X**. Como esses trechos são completamente independentes entre si, é possível aplicar a técnica de paralelização sem que isso incorra em conflitos no acesso aos elementos do vetor.

3. *Framework* de Teste e Ambiente de Execução

Para garantir uma maior portabilidade e usabilidade, utilizamos a linguagem de programação Java, a qual implementa nativamente *threads*. Além disso, implementamos um *framework* de testes para verificar diferentes configurações do vetor de entrada, composto por elementos do tipo numérico. As configurações possíveis envolvem o uso

de valores repetidos ou únicos, e vetores com elementos dispostos de forma aleatória, ordenada ou invertida.

Realizamos os testes com três tamanhos diferentes de vetores: 100 mil, 1 milhão e 10 milhões de elementos. Para cada configuração foram realizadas mil execuções, das quais foi calculado o tempo médio. Nesse cálculo não consideramos as execuções iniciais correspondentes a 10% do total, para garantir uma maior integridade da coleta.

Os testes foram executados em um ambiente contendo Sistema Operacional Windows 7 Ultimate 64 Bits (Service Pack 3) e Java Versão 7 Update 9 (JDK 7u9). Nossa plataforma de testes foi um computador com processador Intel® Core™ i7 2630QM, 2.0~2.9 GHz com 8 cores e 4Gb de memória.

4. Resultados

Com a paralelização concluída testamos o algoritmo paralelo criando de 2 à 64 *threads*. Com os testes executados, foi possível observar algumas características do algoritmo: *(i)* a ordenação de um vetor ordenado possui um tempo de execução desprezível, não se observando ganhos na versão paralela; *(ii)* vetores pequenos, com menos de 100 mil elementos, também se incluem no caso anterior e *(iii)* o tempo de execução com o vetor invertido não difere do tempo com um vetor aleatório, comprovando que não há pior caso específico.

A Figura 2 apresenta um gráfico do tempo de execução (em milissegundos) variando-se o número de *threads* de 1 (sequencial) a 64 para vetores de 100 mil, 1 milhão e 10 milhões de elementos. Nele observa-se que a redução do tempo de execução foi mais significativa para a maior entrada (10 milhões de elementos – linha superior do gráfico). Confirmando esta tendência e para facilitar a visualização, a Figura 3 mostra os mesmos resultados só que apenas para as entradas menores: 100 mil e um milhão de elementos.

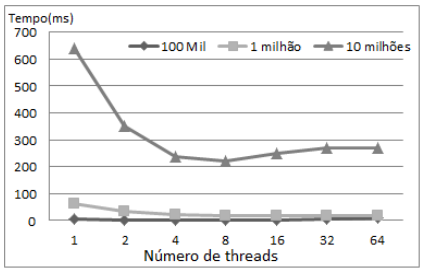

Figura 2: Gráfico do tempo de execução (ms) variando-se o número de *threads* para os 3 tamanhos de vetores.

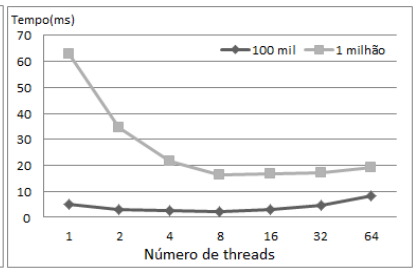

Figura 3: Gráfico do tempo de execução (ms) variando-se o número de *threads* para 100 mil e 1 milhão de elementos.

A Tabela 1 apresenta os *speedups* obtidos nas execuções paralelas para cada tamanho de vetor. Em geral, o pico de desempenho acontece quando temos 8 *threads* paralelas independentemente do tamanho do vetor. Este resultado está relacionado à existência de 8 *cores* em nossa arquitetura paralela. Contudo, podemos observar também que o *speedup* mais próximo do ideal, ocorre quando utilizamos apenas 2 *threads*.

Tabela 1: *Speedups* com 2, 4, 8, 16, 32 e 64 threads para cada tamanho de vetor.

Tamanho	Número de threads					
	2	4	8	16	32	64
100 mil	1,67	2,06	2,25	1,71	1,14	0,61
1 milhão	1,73	2,76	3,63	3,59	3,50	3,14
10 milhões	1,82	2,70	2,88	2,57	2,37	2,38

Uma explicação para o fato de o *speedup* cair com mais threads é o fato de que o algoritmo precisa mover elementos antes que a paralelização ocorra, conforme ilustrado na Figura 1. Como essa operação não foi paralelizada, o seu custo pode estar afetando a eficiência geral do algoritmo.

6. Conclusões

Fazendo uma análise dos resultados, concluímos que foi possível melhorar o desempenho e a eficiência do algoritmo de ordenação aplicando Java Threads. Obtivemos bons resultados, com uma redução de até 74% no tempo execução do algoritmo sequencial quando executado com 8 *threads*.

Embora o maior ganho obtido tenha sido ao utilizar 8 threads paralelas, foi ao criar apenas 2 que o *speedup* se aproximou mais do ideal. Adicionalmente, identificamos que há um trecho sequencial no inicio da execução do algoritmo e pretendemos investir esforços para torná-lo mais eficiente e aumentar os ganhos de nossa implementação. Também, na próxima etapa do nosso trabalho, testaremos o algoritmo com entradas maiores e realizaremos a comparação da nossa implementação paralela com outras similares, em relação ao desempenho e ao consumo de memória.

Referências

Cormen, T. H. et al. “Algoritmos: Teoria e Prática”. Rio de Janeiro: Elsevier, 2002.

Szwarcfiter, J. L. e Markenzon, L. “Estruturas de dados e seus algoritmos”. LTC editora, 1994.

Ziviani, N. “Projeto de algoritmos: implementações em Java e C++”. Thomson, 2007.

Duvanenko, V. J. (2012) “Parallel In-Place Merge”. Disponível em: http://www.drdobbs.com/parallel/parallel-in-place-merge/240008783. Acessado em Janeiro de 2013.

Pok-Son Kim e Arne Kutzner “A Simple Algorithm for Stable Minimum Storage Merging”. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, Springer-Verlag, 2007.