

Multiparticionamento Flexível em Processadores Multicore

Michel B. Cordeiro¹, Guilherme S. Bluhm¹, Wagner M. Nunan Zola¹

¹Departamento de Informática – Universidade Federal do Paraná (UFPR)

michel.brasil.c@gmail.com, gsbluhm@inf.ufpr.br, wagner@inf.ufpr.br

Resumo. *Este trabalho propõe um algoritmo paralelo eficiente de multiparticionamento de dados em processadores multicore. O algoritmo permite definição dos limites de cada faixa de dados, possibilitando particionamentos variáveis, além de apresentar boa escalabilidade.*

1. Introdução

O particionamento pode ser considerado um mecanismo de decomposição. Tipicamente consiste em dividir um conjunto de dados, em subconjuntos ou partições que podem ser processadas em paralelo. O multiparticionamento tem como objetivo reorganizar os dados de entrada em “bins” (ou “buckets”), contíguos na memória, utilizando uma função que categoriza os elementos em suas respectivas partições. Um subproblema do multiparticionamento ocorre quando a função de categorização utiliza um vetor de faixas para organizar os elementos de entrada. Assim, dado um conjunto de elementos $S = \{x_0, x_1, \dots, x_n\}$ e um conjunto de faixas $P = \{p_0, p_1, \dots, p_k\}$, o objetivo é particionar S em k subconjuntos disjuntos B_0, B_1, \dots, B_{k-1} , tais que $B_i = \{x \in S \mid p_i \leq x < p_{i+1}\}$, para $i \in \{0, 1, \dots, k-1\}$, sendo $p_k = \infty$. O multiparticionamento é utilizado para balancear cargas de trabalho em muitas aplicações paralelas, como no algoritmo *Barnes-hut* e em algoritmos de ordenação. No entanto, apesar de sua relevância, o multiparticionamento recebeu pouca atenção na literatura, se considerado como uma primitiva paralela independente. Diante disso, este artigo propõe uma implementação eficiente de multiparticionamento paralelo utilizando *threads*.

2. Descrição do Algoritmo

Para categorizar os elementos de entrada de forma eficiente, é essencial utilizar um método de busca otimizado para determinar a faixa correspondente a um determinado elemento. Considerando que o conjunto de faixas P é representado por um vetor ordenado, a busca binária é a abordagem mais eficiente para essa tarefa. Estudos conduzidos por [Khuong and Morin 2017] exploraram diferentes estratégias para otimizar a busca binária em C++. Com base em testes e ajustes, os autores concluíram que, para valores pequenos de n (até 2^{21} , segundo o estudo), a melhor abordagem consiste na combinação da busca binária com técnicas de movimentos condicionais que visa minimizar erros de predição de ramificação, e com pré-busca (*prefetching*) para reduzir a latência de cache. Como o conjunto P tende a ser consideravelmente menor que o vetor a ser particionado, o algoritmo proposto neste estudo adota a busca binária otimizada desenvolvida por Khuong et al. O algoritmo proposto é estruturado em três etapas. Inicialmente, o vetor de entrada é distribuído entre as *threads*, que identificam a faixa correspondente a cada elemento, construindo um histograma local por *thread* e um histograma global para toda a entrada. Em seguida, aplica-se a operação de *scan* exclusivo aos histogramas, permitindo determinar a posição exata onde os elementos de cada faixa devem ser inseridos. Por

fim, as *threads* percorrem novamente o vetor de entrada e inserem os elementos em suas respectivas partições no vetor de saída. Além disso, o algoritmo retorna a posição inicial de cada faixa no vetor de saída, determinada pelo *scan* exclusivo do histograma global.

3. Resultados e Discussões

Para avaliar a eficiência do algoritmo proposto, foram gerados conjuntos de dados com tamanhos de 1, 2, 4, 8, 16 e 32 milhões de elementos do tipo *long long int*, seguindo uma distribuição uniforme. Os experimentos foram conduzidos com o objetivo de analisar a escalabilidade do algoritmo, variando três atributos: (i) número de *threads*, que variou de 1 a 32, utilizando 32 milhões de elementos particionados em 16 mil faixas; (ii) quantidade de elementos, que varia de 1 a 32 milhões, particionados em 16 mil faixas; e processados com 32 *threads*; e (iii) número de faixas, variando de 256 a 16 mil, com 32 milhões de elementos e 32 *threads*. Os testes foram realizados 100 vezes e a vazão média, expressa em milhões de elementos particionados por segundo (MEPPS), foi reportada. Também foram calculados intervalos de confiança de 95%, mas não foram observadas variações maiores do que 1% em relação à média e, por isso, esses valores não foram reportados. O ambiente de execução consiste em um processador Intel Xeon Silver 4314 @ 2.40GHz, com 16 núcleos (32 *hyperthreads*), utilizando o sistema operacional Linux Ubuntu 20.04.3 LTS.

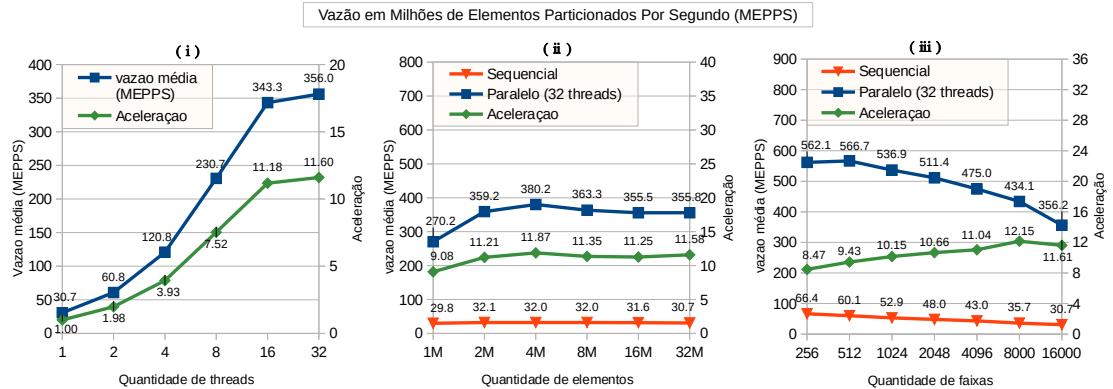


Figura 1. Resultado dos experimentos (i) com 32 milhões de elementos, variando a quantidade de *threads*, (ii) variando a quantidade de elementos e (iii) a quantidade de faixas. Os experimentos (ii) e (iii) foram executados com 32 *threads*

Os resultados dos experimentos são apresentados na Figura 1. No processador utilizado, o algoritmo demonstra uma boa escalabilidade em relação ao número de *threads*, alcançando um *speedup* de 7,52 para 8 *threads* em comparação com a execução sequencial. No entanto, o ganho de desempenho tende a se estabilizar com o aumento do número de *threads*. Além disso, a quantidade de faixas impacta negativamente a vazão, como esperado, uma vez que um maior número de faixas implica buscas mais extensas para categorizar os elementos, ou seja, para cada elemento de entrada mais trabalho é adicionado. Dessa forma, esse impacto também ocorre na versão sequencial. Assim, podemos observar boa aceleração, mesmo com aumento de faixas.

Agradecimentos

Parcialmente suportado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processo 407644/2021-0, bem como pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Programa de Excelência Acadêmica (PROEX).

Referências

Khuong, P.-V. and Morin, P. (2017). Array layouts for comparison-based searching. *ACM J. Exp. Algorithmics*, 22.