Algoritmo Paralelo e Distribuído para Ordenação Chave-Valor

Michel B. Cordeiro[1], **Rodrigo Morante Blanco**[1], **Wagner M. Nunan Zola** [1]

[1] Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

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

Resumo. *Este trabalho apresenta um algoritmo de ordenação chave-valor com execução tanto paralela quanto distribuída. Experimentos realizados demonstraram que o algoritmo foi capaz de alcançar uma aceleração de até 3.3 vezes em relação ao algoritmo std::par com biblioteca Intel TBB. Além disso, o algoritmo demonstrou boa escalabilidade tanto em relação ao número de threads quanto à quantidade de nodos de processamento.*

1. Introdução

A ordenação é uma operação fundamental em computação, sendo essencial para diversas aplicações, como bancos de dados e aprendizado de máquina. Nesse contexto, a computação paralela e distribuída tem se mostrado uma abordagem promissora para acelerar algoritmos de ordenação, com técnicas que exploram processadores *multicore* [Reinders 2007], GPUs [Ashkiani et al. 2017] e *clusters* [Siebert 2011]. A ordenação chave-valor é uma forma de ordenação em que os dados são organizados em pares (chave, valor), sendo ordenados com base nas chaves, enquanto os valores permanecem associados às suas respectivas chaves. Na ordenação distribuída, cada nodo de processamento armazena apenas um subconjunto dos dados totais. Formalmente, a ordenação distribuída pode ser definida da seguinte maneira: seja $D = \{(k_1, v_1), \ldots, (k_m, v_m)\} \subseteq K \times V$ o conjunto de dados desordenado, onde K representa o conjunto de chaves e V o conjunto de valores. A entrada para o algoritmo é composta por uma coleção de subconjuntos $\{D_1, D_2, \ldots, D_n\}$, onde $D_j \subseteq D$ corresponde ao subconjunto de dados desordenado armazenado no nodo N_j. Após a ordenação, obtêm-se uma coleção de subconjuntos ordenados $\{D'_1, D'_2, \ldots, D'_n\}$, onde cada subconjunto $D'_j \subseteq K \times V$ está localmente e globalmente ordenado. A ordenação global é garantida pela seguinte condição: $\forall i < j, \quad \max(k_a \in D'_i) \leq \min(k_b \in D'_j)$. Dessa forma, os elementos do conjunto D'_i precedem corretamente os elementos do conjunto D'_j na ordenação final. A eficiência da ordenação distribuída depende da capacidade de balancear a carga de trabalho entre os nodos, otimizar a comunicação entre eles e minimizar o tempo gasto nas fases de trocas de elementos. Este trabalho apresenta um algoritmo eficiente de ordenação chave-valor que executa de forma paralela em processadores *multicore*, e de maneira distribuída em *clusters* de computadores.

2. Trabalhos Relacionados

A ordenação eficiente é um tema amplamente estudado. Uma abordagem para realizar a ordenação paralela é dividir o conjunto de entrada em partições que serão ordenadas em paralelo pelas unidades de processamento. [Siebert 2011] propôs um algoritmo de

ordenação paralela composto por quatro etapas: ordenação local, determinação dos divisores (splitters), redistribuição dos dados e a combinação das partições ordenadas. A primeira etapa ordena os dados locais em paralelo usando ordenação sequencial. A segunda etapa, identifica cada divisor por meio de uma busca binária global. Embora a redução do espaço de busca possa ser feita por meio da seleção da mediana local e global, a escolha entre a metade superior ou inferior requer a determinação da posição global da mediana, selecionada por meio de uma operação de redução entre todos os processos. Uma vez particionados, os dados são redistribuídos com a operação coletiva *all-to-all*. Finalmente, as partições ordenadas são combinadas por meio de um algoritmo de *p-way merging*, resultando em uma sequência globalmente ordenada. Dessa forma, esse algoritmo consegue balancear a carga de trabalho de maneira eficiente, sem exigir conhecimento prévio da distribuição dos dados e utilizando uma quantidade reduzida de memória adicional. No entanto, ele requer a ordenação prévia dos dados antes da determinação dos *splitters* e, com a ordenação sendo a primeira etapa do algoritmo, torna-se necessário a utilização do *p-way merging* para combinar as partições na etapa final, o que pode implicar em um alto custo computacional em grandes volumes de dados. Por esse motivo, este trabalho propõe uma abordagem na qual a ordenação é realizada na etapa final do algoritmo, após a redistribuição dos elementos, eliminando a necessidade do *p-way merging* e, consequentemente, reduzindo o tempo total do algoritmo.

3. Descrição do Algoritmo

O algoritmo proposto, denominado *Multi Partition Parallel Sort* (*mppSort*), adota dois níveis de paralelismo: o nível intra-nodo, na qual unidades paralelas (*threads*) são executadas no mesmo processador *multicore*, e o nível extra-nodo, onde as unidades paralelas correspondem a processos com múltiplas *threads* em diferentes processadores que se comunicam por troca de mensagens no padrão MPI (*Message Passing Interface*). A Figura 1 apresenta a visão geral do algoritmo de ordenação proposto executando no modo distribuído. Inicialmente é utilizada uma etapa paralela de multi particionamento para dividir os dados entre os nodos que, após uma etapa de comunicação, passa por outra etapa de multi particionamento e pela ordenação das partições localmente, sem a necessidade de uma função de *p-way merging*. A função de multi particionamento é paralelizada entre as *threads* e consiste em três etapas principais: (i) o vetor de entrada é distribuído entre as *threads*, que utilizam uma função de categorização para determinar a faixa correspondente de cada elemento e construir um histograma local para cada *thread*, além de um histograma global para toda a entrada; (ii) aplica-se a operação de *scan* exclusivo sobre os histogramas, permitindo determinar a posição exata onde os elementos de cada faixa devem ser alocados; (iii) as *threads* percorrem novamente o vetor de entrada e inserem os elementos nas respectivas partições do vetor de saída. Além do vetor particionado, o algoritmo retorna a posição inicial de cada faixa no vetor de saída, a qual é definida pelo *scan* exclusivo do histograma global. Para evitar que as cargas de trabalho fiquem desbalanceadas, os dados são particionados em um número muito maior de partições do que a quantidade de nodos, gerando mini-partições que serão combinadas para formar as faixas a serem ordenadas por cada nodo. Após a etapa de comunicação, os dados recebidos também passam por um multi particionamento, cujo objetivo é dividir os dados entre as *threads*. Novamente, a quantidade de partições em que os dados serão divididos é muito maior do que a quantidade de *threads*, pois, além de permitir um melhor balanceamento, as mini-partições podem ser unidas em faixas de tamanhos que aumentam o desempenho

da ordenação. Como a quantidade de elementos influencia diversos fatores, como a possibilidade dos dados caberem no cache, ela tem um grande impacto na vazão do algoritmo. Sendo assim, escolher um tamanho adequado para as mini-partições a serem unidas faz com que o *mppSort* utilize sempre a melhor vazão do algoritmo de ordenação, aumentando ainda mais o desempenho do algoritmo. Sendo assim, o *mppSort* possui quatro etapas principais: multi particionamento global, etapa de comunicação, multi particionamento local e ordenação. No entanto, é importante ressaltar que, se o algoritmo estiver sendo executado em apenas um nodo, as duas etapas iniciais podem ser descartadas.

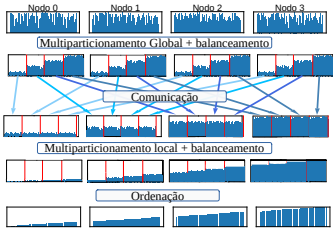

Figura 1. Visão geral do *mppSort*.

		Quantidade de Pares Chave-Valor (Chave: 64 bits, Valor: 32 bits)											
		1 Milhão			8 Milhões			16 Milhões			32 Milhões		
		std::par (TBB)	mppSort	speedup	std::par (TBB)	mppSort	speedup	std::par (TBB)	mppSort	speedup	std::par (TBB)	mppSort	speedup
Quantidade de Threads	1	9.6	**17.3**	1.8	8.3	**13.3**	1.6	7.9	**12.4**	1.6	7.6	**11.6**	1.5
	2	17.7	**33.5**	1.9	16.2	**26.3**	1.6	15.6	**24.5**	1.6	14.9	**22.9**	1.5
	4	31.8	**50.5**	1.6	31.1	**48.1**	1.5	30.0	**47.5**	1.6	28.7	**44.2**	1.5
	8	57.0	**73.1**	1.3	53.4	**82.9**	1.6	51.3	**92.2**	1.8	48.1	**87.5**	1.8
	16	80.0	**92.1**	1.2	71.8	**134.3**	1.9	67.6	**131.3**	1.9	61.5	**145.6**	2.4
	32	97.2	**97.2**	1.0	69.8	**170.7**	2.4	64.7	**185.1**	2.9	58.7	**194.9**	3.3

Tabela 1. Vazão em MEPS dos algoritmos *std::par* (TBB) e *mppSort* na ordenação paralela.

4. Metodologia, Resultados e Discussões

Para avaliar a eficiência do *mppSort* em um nodo, foram gerados conjuntos de dados com tamanhos de 1, 2, 4, 8, 16 e 32 milhões de pares chave-valor do tipo *long long int* e *unsigned int*, respectivamente, seguindo uma distribuição normal com média de 1×10^9 e desvio padrão de 125×10^6. Os experimentos foram conduzidos com o objetivo de analisar a escalabilidade do algoritmo, variando dois parâmetros: (i) número de *threads*, que variou de 1 a 32 e; (ii) quantidade de elementos, considerando conjuntos de dados de 1 a 32 milhões de elementos. O desempenho do *mppSort* foi comparado com o algoritmo de ordenação paralela padrão do C++ std::sort(std::execution::par), que será abreviado para *std::par*, utilizando o *Threading Building Blocks* (TBB) [Reinders 2007] como *backend*. O ambiente de execução para esses experimentos consiste em um processador Intel Xeon Silver 4314 @ 2.40GHz, com 16 núcleos (32 *hyperthreads*), utilizando o sistema operacional Linux Ubuntu 20.04. Para analisar o desempenho do *mppSort* no cenário distribuído, foram utilizados 4 nodos na plataforma *Amazon Web Services* (AWS), com processadores Intel Xeon 8375C CPU @ 2.90GHz com 32 núcleos, dos quais 8 foram disponibilizados para os experimentos, usando sistema operacional Amazon Linux 2023.6. A interface de rede é de 25 gigabits. No entanto, testes realizados indicaram que a velocidade efetiva da rede estava em torno de 10 gigabits durante os experimentos. Os experimentos tinham como objetivo analisar a eficiência de cada etapa do algoritmo e sua escalabilidade em relação à quantidade de nodos. Para isso, foi utilizado um conjunto de dados com 32 milhões de pares chave-valor também gerados seguindo uma distribuição normal. Em ambos os cenários, os testes foram realizados 30 vezes, e a vazão média de milhões de elementos processados por segundo (MEPS) foi reportada. A quantidade de partições globais e locais foi 16 mil, agrupadas em faixas de 100 mil elementos na etapa de ordenação. Esses valores dependem da máquina utilizada, mas testes anteriores indicaram que utilizar quantidades de partições entre 100 e 1.000 vezes maiores do que a quantidade de nodos (ou *threads*) e definir o tamanho das faixas de forma que caibam no cache do processador costumam ser boas escolhas para o desempenho do algoritmo.

O resultado da comparação entre o *std::par* e o *mppSort* está na Tabela 1. Observa-se que o *mppSort* apresentou bom desempenho, principalmente para grandes volumes de

dados, alcançando uma aceleração de até 3.3 vezes em relação ao *std::par*, como pode ser visto no gráfico da esquerda na Figura 2. Além disso, a vazão do *mppSort* aumenta à medida que a quantidade de pares a serem ordenados cresce, indicando que o algoritmo ainda não atingiu o pico de vazão para a máquina e os parâmetros escolhidos, ao contrário do *std::par*, que atinge o pico de vazão de 92.2 MEPS com 32 *threads* e 1 milhão de elementos. Outro fator relevante é a escalabilidade dos algoritmos em relação à quantidade de *threads*. Enquanto o *std::par* atinge a vazão máxima com 32 *threads* e apenas 1 milhão de elementos, o *mppSort* continua a escalar com aumento da quantidade de *threads*.

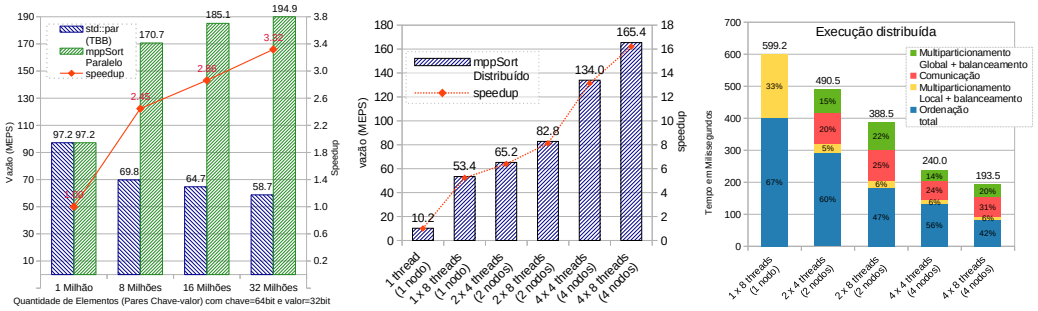

Figura 2. À esquerda, comparação do desempenho entre *std::par* e *mppSort*. No centro, resultados dos experimentos que analisam o desempenho do *mppSort* no cenário de ordenação distribuída para 32 milhões de pares chave-valor. À direita, análise do tempo gasto em cada etapa do algoritmo.

O desempenho do *mppSort* no cenário distribuído pode ser observado nos gráficos ao centro e à direita na Figura 2. É possível observar que o tempo total do algoritmo diminui não apenas com o aumento na quantidade de *threads*, mas também com o aumento na quantidade de nodos. Isso pode ser explicado pelo fato de que, como a ordenação está distribuída, o aumento na quantidade de nodos reduz a quantidade de pares chave-valor a serem ordenados e particionados localmente em cada nodo. Assim, o *mppSort* é capaz de escalar não apenas com o número de *threads*, mas também com o número de nodos.

5. Conclusão

Devido à sua estratégia de dividir os dados e balancear as cargas de trabalho, o algoritmo proposto *mppSort* demonstrou bom desempenho, sendo capaz de alcançar uma aceleração de até 3.3 vezes em comparação ao *std::par*, especialmente ao lidar com grandes volumes de dados. O algoritmo não apenas acelera significativamente a ordenação, mas também exibe excelente escalabilidade, tanto com o aumento na quantidade de *threads* quanto com o número de nós em um cenário distribuído. Essa escalabilidade, combinada com seu bom desempenho em diferentes configurações, demonstra o potencial do *mppSort* em acelerar aplicações que lidam com grandes quantidades de dados.

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

Ashkiani, S., Davidson, A., Meyer, U., and Owens, J. D. (2017). GPU Multisplit: an extended study of a parallel algorithm. *ACM Transactions on Parallel Computing*.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.".

Siebert, C. (2011). *A scalable parallel sorting algorithm using exact splitting*.