Algoritmo Paralelo Eficiente 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 paralelo eficiente para a ordenação de pares chave-valor em processadores* multicore*. A estratégia principal consiste em dividir os dados de entrada em um grande número de mini-partições. Essa abordagem melhora o balanceamento de carga entre* threads*, permite a construção de faixas com tamanhos adequados, maximiza a vazão da etapa de ordenação paralela e, consequentemente, o desempenho global. Experimentos mostram que o algoritmo atinge boa aceleração em relação ao estado da arte, tanto na ordenação de pares quanto de chaves isoladas, e demonstra boa escalabilidade com o aumento do número de* threads*, característica essencial para sua aplicação em processadores modernos com muitos núcleos.*

Abstract. *This work presents an efficient parallel algorithm for sorting key-value pairs on multicore processors. The main strategy consists of dividing the input data into a large number of mini-partitions. This approach improves load balancing among threads, enables the construction of appropriately sized partitions, maximizes the throughput of the parallel sorting phase, and consequently enhances overall performance. Experimental results show that the algorithm achieves good speedup compared to the state of the art, both for key-value pairs and for key-only sorting. It also demonstrates good scalability with the increase in the number of threads, a feature that is essential for its application on modern processors with a high number of cores.*

1. Introdução

A ordenação de dados é uma operação fundamental em computação, com amplas aplicações em sistemas de banco de dados, inteligência artificial e computação gráfica. Algoritmos de ordenação são criticamente importantes na área de bancos de dados, tanto para a otimização de desempenho quanto para a execução correta de consultas. Apesar de não ser diretamente visível ao usuário, a ordenação se faz necessária para a operação eficiente de sistemas de banco de dados, por viabilizar consultas ágeis, mecanismos eficazes de indexação, análises de dados e o processamento adequado de transações. Internamente, os mecanismos de bancos de dados utilizam algoritmos de ordenação para entregar dados de consultas na ordem solicitada. Frequentemente, em operações de *join*, os bancos de dados ordenam primeiro as duas entradas da junção, o que costuma ser eficiente pois os dados já estão parcialmente ordenados.

A importância da ordenação também se observa em diversos trabalhos que empregam a ordenação como uma etapa crucial em suas abordagens, como em técnicas de imputação de dados ausentes [Tavares et al. 2024], na organização de anúncios

[Casimiro et al. 2014], na modelagem de tópicos para a recuperação de casos legais [Novaes et al. 2023], no encadeamento de notícias [Cavalcante e Pinheiro 2013], na replicação descentralizada em bancos de dados distribuídos [Ferreira et al. 2023] e na evolução de esquemas em bancos de dados orientados a documentos [Rodrigues et al. 2024], entre muitos outros exemplos na área de banco de dados.

A ordenação consiste em organizar um conjunto de elementos em uma determinada sequência, geralmente em ordem crescente ou decrescente, com base em um critério definido. Já a ordenação chave-valor é uma variação da 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.

Sendo assim, a ordenação de pares chave–valor consiste em reorganizar os elementos de um vetor de pares (k_i, v_i), com base apenas nas chaves k_i, mantendo a associação original entre cada chave k_i e seu respectivo valor v_i [Blanco et al. 2025]. A ordenação de pares chave–valor impõe desafios adicionais em relação à ordenação de chaves isoladas, pois é necessário garantir que cada valor permaneça corretamente associado à sua respectiva chave. Isso exige cópias coordenadas, o que aumenta o volume de dados trafegando entre os níveis da hierarquia de memória, reduzindo o reaproveitamento de dados em *cache* — devido a estruturas maiores — e, com isso, a vazão efetiva de dados.

Apesar da existência de algoritmos clássicos e amplamente estudados, como *Quicksort*, *Merge Sort* e *Heap Sort*, muitos trabalhos ainda buscam por técnicas de ordenação mais eficientes. Em particular, têm sido investigadas formas de tornar os algoritmos de ordenação mais eficientes na prática, considerando as complexas características do *hardware* moderno, como: processadores *multicore* [Bingmann et al. 2017], paralelismo de instruções [Hou et al. 2015], predição de desvio (*branch prediction*) [Edelkamp e Weiß 2019, Bingmann et al. 2017], *caches* [Kokot et al. 2018] e computação distribuída [Siebert 2011].

Em contraste, os algoritmos de ordenação utilizados nas bibliotecas padrão de linguagens de programação como Java ou C++ utilizam algoritmos híbridos. Por exemplo, o `std::sort` de C++11 usualmente utiliza o algoritmo *Quicksort* [Hoare 1962], *Heap Sort* e *Insertion Sort*, aplicando o melhor método em cada caso ou etapa do algoritmo. O *Quicksort* continua sendo um algoritmo extremamente eficiente, que pode ser paralelizado [Reinders 2007], implementado de forma a evitar predições de desvio incorretas [Edelkamp e Weiß 2019], e ainda adaptado para arquiteturas de memória distribuída, como demonstrado pelo *HykSort* [Sundar et al. 2013].

Este trabalho apresenta um algoritmo eficiente para ordenação de chaves ou de pares chave-valor, otimizado para ambientes paralelos com processadores *multicore*. A proposta busca aprimorar abordagens tradicionais ao explorar o paralelismo em nível de dados e considerar aspectos essenciais de desempenho, como localidade de memória, melhor aproveitamento das hierarquias de *cache* e balanceamento de carga entre as unidades de processamento. As principais contribuições do método proposto estão em:

- Utilização de multiparticionamento dos dados em quantidades significativamente grandes de mini-partições (pequenas faixas).
- Balanceamento eficiente da carga de trabalho entre as *threads*, que pode ser adequadamente obtido pelo agrupamento de muitas mini-partições.

- Adoção de histogramas e somas de prefixos e barreiras como estratégia de balanceamento, eliminando a necessidade de sincronização de baixa granularidade, como *locks*.
- Agrupamento das mini-partições em conjuntos de "tamanho ideal". Os conjuntos de "tamanho ideal" são ordenados em paralelo, possibilitando a maximização da vazão de ordenação por meio do aproveitamento eficiente da memória *cache*.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; na Seção 3 são detalhadas as estratégias de implementação; na Seção 4, é descrita a metodologia dos experimentos; na Seção 5, é realizada a análise dos resultados; e, finalmente, na Seção 6, são apresentadas as conclusões deste estudo.

2. Trabalhos Relacionados

O algoritmo *Quicksort* [Hoare 1962] e suas variantes estão entre os métodos de ordenação mais utilizados, devido à sua eficiência prática em uma ampla variedade de cenários. O *Quicksort* funciona selecionando um elemento pivô e particionando o vetor de forma que todos os elementos menores que o pivô sejam posicionados em índices anteriores, enquanto os elementos maiores ocupam índices posteriores. Com isso, o pivô é colocado em sua posição final no vetor ordenado, e o processo se repete recursivamente para os sub-vetores anteriores e posteriores ao pivô.

Algumas variantes do algoritmo evitam o uso de recursividade ao processar o vetor de forma sequencial, da esquerda para a direita [Bing-Chao e Knuth 1986]. Isso é possível ao utilizar um posicionamento cuidadoso dos pivôs, de modo que a maioria dos elementos seja deslocada para a parte direita do vetor. Uma dessas variantes, o *Introselect* [Musser 1997], utiliza ainda o *Heap Sort* como mecanismo de *fallback* para evitar a degradação do desempenho no pior caso. Este algoritmo é atualmente empregado no `std::sort`, o algoritmo de ordenação da biblioteca padrão da linguagem C++.

Existem também variantes que utilizam múltiplos pivôs [Kushagra et al. 2014], como as versões com dois ou três pivôs, as quais podem alcançar ganhos de desempenho de até 8% em relação à abordagem tradicional com um único pivô. Embora o princípio fundamental do *Quicksort* seja preservado, essas variantes particionam os dados em três ou quatro subproblemas, em vez de dois, otimizando o balanceamento e o uso da hierarquia de memória.

O algoritmo *Sample Sort* [Frazer e McKellar 1970][Huang e Chow 1983] pode ser interpretado como uma generalização do *Quicksort*, utilizando $k - 1$ divisores para particionar a entrada em k subproblemas, chamados de *buckets*, tentando manter tamanhos aproximadamente equilibrados. Uma das vantagens do *Sample Sort* é apresentar características que o tornam especialmente adequado à paralelização, além de ser mais eficiente no uso da memória *cache* do que o *Quicksort* tradicional. No *Sample sort*, o particionamento da entrada em subconjuntos disjuntos possibilita a ordenação paralela de cada subconjunto de maneira independente.

Variantes do *Sample Sort*, como o *S4o* (*Super Scalar Sample Sort*) [Sanders e Winkel 2004], foram projetadas para aproveitar características modernas de *hardware*, incluindo instruções vetoriais e hierarquias de *cache*, obtendo desempenho competitivo em relação aos algoritmos tradicionais mesmo em ambientes com muitos

núcleos de processamento. Como resultado, o *S4o* pode ser até duas vezes mais rápido que implementações convencionais de *Quicksort*. O algoritmo *Block Quicksort* [Edelkamp e Weiß 2019] adota abordagens similares ao aplicar a eliminação de desvios ao *Quicksort* com pivô único, resultando em um algoritmo altamente eficiente, com desempenho comparável ao do *S4o*.

O artigo "*Engineering In-place (Shared-Memory) Sorting Algorithms*" [Axtmann et al. 2022] apresenta avanços significativos introduzindo o *IPS4o* (*In-place Parallel Super Scalar Samplesort*), um algoritmo de ordenação que combina técnicas de distribuição eficiente com árvores de decisão sem desvios condicionais. O *IPS4o* adapta dinamicamente o grau de distribuição e trata eficientemente elementos duplicados, resultando em um algoritmo robusto que supera os concorrentes em diversas configurações. O *IPS4o* demonstrou desempenho superior em comparação com os principais algoritmos de ordenação, tanto em ambientes paralelos quanto sequenciais.

O *Radix Sort* é um algoritmo altamente paralelizável que não utiliza operações de comparação. Esse algoritmo costuma ser mais eficiente no processamento massivamente paralelo com GPUs [Merrill e Grimshaw 2011] superando algoritmos tradicionais baseados em comparações, como o *Quicksort* e o *Merge Sort*, especialmente quando aplicado a grandes volumes de dados numéricos.

3. Descrição do Algoritmo

Este trabalho propõe um algoritmo de ordenação eficiente para arquiteturas *multicore*, denominado *Multi-Partition Parallel Sort* (*mppSort*), implementado com o uso da biblioteca `pthreads`. A utilização de *threads*, em vez de MPI, mostra-se mais apropriada em arquiteturas de memória compartilhada, uma vez que *threads* são processos leves (*lightweight processes*) e, portanto, tendem a apresentar melhor desempenho do que processos tradicionais nesse contexto.

Uma abordagem comum para realizar ordenação paralela consiste em dividir o conjunto de entrada em partições que serão ordenadas concorrentemente pelas unidades de processamento, utilizando técnicas similares ao *Sample Sort* [Frazer e McKellar 1970]. Ao utilizar essa abordagem no algoritmo *mppSort*, é possível distribuir a carga de trabalho de forma equilibrada entre as *threads*. Como as partições são disjuntas, não existem conflitos de acesso à memória nem, consequentemente, condições de corrida. Além disso, o uso de barreiras e o balanceamento de carga baseado em histogramas e somas de prefixos dispensam a necessidade de sincronizações de baixa granularidade, como *locks*.

Dessa forma, a principal estratégia do *mppSort* consiste na aplicação de uma função paralela de multi-particionamento que distribui os dados entre as *threads*, permitindo que cada partição seja ordenada em paralelo de forma independente. Contudo, uma desvantagem dessa abordagem é que o particionamento pode resultar em uma divisão desbalanceada dos dados. Para mitigar esse problema, os dados são inicialmente divididos em um número significativamente maior de partições do que o número de *threads*, gerando mini-partições que são posteriormente agrupadas para formar as partições finais atribuídas a cada *thread*.

Para obter ganhos de desempenho, é fundamental que o algoritmo de multi-particionamento seja otimizado. Uma implementação eficiente já foi demonstrada em

GPUs por [Ashkiani et al. 2017][Cordeiro e Zola 2025]. O algoritmo intitulado *GPU Multisplit* emprega otimizações que fazem uso eficaz da memória compartilhada e dos registradores, reduzindo a aleatoriedade nas escritas na memória global e melhorando desempenho. A estratégia adotada neste trabalho foi inspirada em uma versão do algoritmo *GPU Multisplit* adaptada para CPUs.

Além disso, o algoritmo adota uma estratégia adicional que é combinar as mini-partições de modo a formar faixas com tamanhos que maximizem o desempenho do algoritmo de ordenação utilizado pelas *threads*. Como a quantidade de elementos a serem ordenados afeta diretamente diversos fatores, como a possibilidade de os dados se manterem em *cache*, o tamanho das faixas tem impacto direto na vazão do algoritmo. Dessa forma, ajustar as faixas para que se aproximem de um tamanho "ideal" permite que as *threads* operem com máxima eficiência, contribuindo para um desempenho superior do *mppSort*.

Sendo assim, o *mppSort* conta com dois parâmetros configuráveis: o número de mini-partições e o tamanho ideal das faixas a serem ordenadas. Os valores desses parâmetros dependem do processador utilizado mas, em geral, utilizar quantidades de partições entre 100 e 1.000 vezes maiores do que a quantidade de *threads* e definir o tamanho das faixas de forma que caibam no *cache* do processador costumam ser boas escolhas para o desempenho do algoritmo. Considerando que a latência de acesso à memória principal é significativamente maior do que a do *cache*, definir faixas que se ajustem ao *cache* L2 ou L3 contribui diretamente para o desempenho do algoritmo. Na Seção 4 descrevemos como esses dois parâmetros configuráveis do algoritmo *mppSort* foram obtidos experimentalmente para o presente trabalho. No entanto, a obtenção desses parâmetros poderia ser feita por um algoritmo de ajuste (*tunning*) que encontre os valores otimizados para um dado processador, testando diferentes tipos de distribuições comuns. Também estão previstos métodos alternativos na API do *mppSort* em que esses parâmetros adicionais possam ser fornecidos pelo usuário, possibilitando a adaptação de maneira customizada, no caso de se trabalhar com uma distribuição de dados específica.

Sendo assim, o *mppSort* é composto por cinco etapas principais:

1. O vetor de entrada é dividido entre as *threads*, que identificam a faixa correspondente de cada elemento e constroem, simultaneamente, um histograma local por *thread* e um histograma global para toda a entrada.
2. Aplica-se a operação de *scan* exclusivo sobre os histogramas, a fim de determinar a posição em que os elementos de cada faixa devem ser incluídos.
3. As *threads* percorrem novamente o vetor de entrada e inserem os elementos em suas respectivas partições no vetor particionado.
4. Realiza-se uma etapa de balanceamento, em que as partições são agrupadas em conjuntos de tamanho ideal para ordenação.
5. As *threads* ordenam o vetor, aplicando o algoritmo de ordenação de forma independente sobre cada conjunto de partições.

A visão geral do algoritmo pode ser observada na Figura 1. O pseudo-código está descrito no Algoritmo 1. Inicialmente, o conjunto de dados $X = \{x_1, x_2, \ldots, x_n\}$ é particionado em subconjuntos igualmente distribuídos entre as *threads* disponíveis, formando o vetor W. Cada *thread* calcula um histograma local $\mathcal{H}_t$ com base em sua porção de dados (linhas 2–5). Em seguida, todos os histogramas locais são somados para compor um

histograma global $\mathcal{H}_g$ (linha 7), o qual passa por uma soma de prefixos (*prefix sum*) para calcular os deslocamentos corretos de inserção dos elementos nas partições globais (linha 9). Nas linhas 11 a 16, os elementos são efetivamente gravados no vetor particionado, de modo que cada partição $\mathcal{P}_i$ contém um subconjunto reduzido de dados.

Depois disso, o algoritmo passa por uma etapa de balanceamento, na qual as partições são agrupadas em blocos de tamanho ideal para a ordenação posterior (linhas 19–23). A função de busca binária é utilizada para localizar, no vetor de posições, o índice da primeira partição cujo início é maior ou igual a cada múltiplo do tamanho ideal. Esse procedimento divide o vetor particionado em blocos (*chunks*) cujos tamanhos se aproximam do valor ideal. Após o particionamento e o balanceamento, o algoritmo está pronto para a fase de ordenação, como ilustrado na Figura 1. Por fim, na etapa de ordenação (linhas 26–28), as *threads* trabalham em paralelo sobre esses blocos, resultando no vetor ordenado X'.

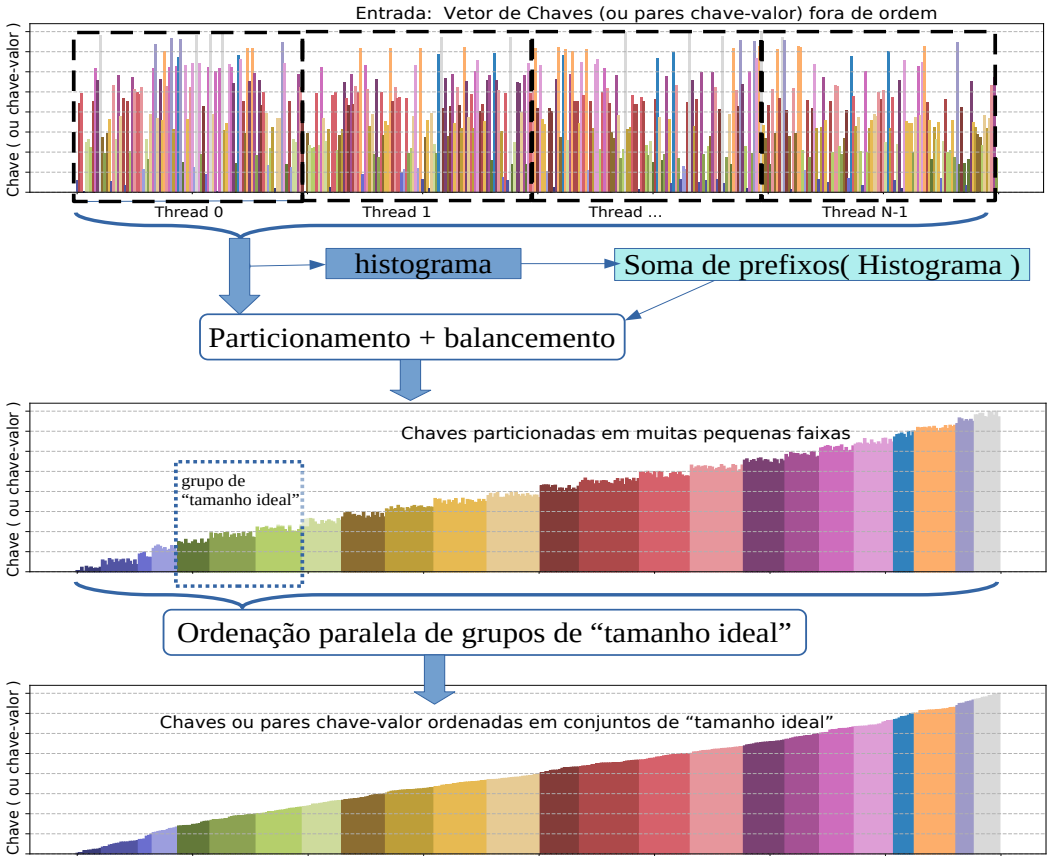

Figura 1. Visão geral do *mppSort*.

O algoritmo de ordenação utilizado pelas *threads* na etapa final do *mppSort* é arbitrário e pode ser substituído. Dessa forma, qualquer algoritmo pode ser utilizado para ordenar as faixas de tamanho ideal. Com o objetivo de maximizar o desempenho do *mppSort*, dois algoritmos foram considerados: `std::sort` e *IPS4o*. Os experimentos realizados indicaram que o *IPS4o* apresenta desempenho superior ao `std::sort` quando os dados cabem na memória *cache*, tornando-o a opção mais adequada para ser utilizada pelas *threads* na etapa final.

Algoritmo 1 *mppSort*

Entrada: $X = \{x_1, x_2, \ldots, x_n\}$ ▷ Conjunto de pares chave-valor desordenados
Saída: $X' = \{x'_1, x'_2, \ldots, x'_n\}$ ▷ Conjunto de pares chave-valor ordenados

1: W_t = subconjunto de X atribuído à *thread* t ▷ X é dividido igualmente entre as *threads*
2: **for** $w \in W_t$ **do** ▷ cada *thread* calcula $\mathcal{H}_t(W_t)$
3: $p = encontra\_partição(w)$
4: $\mathcal{H}_t[p] = \mathcal{H}_t[p] + 1$ ▷ $\mathcal{H}_t$ = histograma local de W_t
5: **end for**
6: $threads\_barrier()$
7: $\mathcal{H}_g$ = soma de todos os $\mathcal{H}_t$ ▷ $\mathcal{H}_g$ = histograma global de X
8: $threads\_barrier()$
9: $Pos = prefix\_sum(\mathcal{H}_g)$
10: $threads\_barrier()$
11: **for** $w \in W_t$ **do** ▷ *threads* gravam os elementos de W_t em suas respectivas partições
12: $p = encontra\_partição(w)$
13: $pos = Pos[p]$
14: $\mathcal{P}_p[pos] = w$ ▷ $\mathcal{P}_p$ = Partição p
15: $Pos[p] = Pos[p] + 1$
16: **end for**
17: $threads\_barrier()$
18: $i, j = 0$
19: **repeat** ▷ partições são agrupadas em *chunks* de "tamanho ideal"
20: $j = lower\_bound\_binary\_search(Pos, i * ideal\_size)$
21: $Chunk[i] = j$ ▷ j representa o início de cada chunk
22: $i = i + 1$
23: **until** $j \geq |X|$
24: $threads\_barrier()$
25: C_t = conjunto de chunks dividido igualmente entre as threads
26: **for** $c \in C_t$ **do** ▷ *threads* ordenam seus cojuntos de *chunks* em paralelo
27: $X'_t = sort(c)$ ▷ os elementos são colocados nas suas posições no vetor final
28: **end for**
29: $threads\_barrier()$
30: **return** X'

4. Metodologia

Para avaliar a eficiência do *mppSort*, seu desempenho foi comparado com o algoritmo de ordenação paralela *IPS4o*, e com o algoritmo padrão de ordenação paralela da biblioteca C++, o `std::sort(std::execution::par, ...)`, que será abreviado para `std::par`. A implementação atual do `std::par` tem seu funcionamento apenas em conjunto com a biblioteca *Intel Threading Building Blocks* (TBB) como *backend* [Reinders 2007]. Os experimentos foram realizados com conjuntos de dados no formato chave-valor, em que as chaves são representadas pelo tipo `long long int` (64 *bits*) e os valores pelo tipo `unsigned int` (32 *bits*). Além disso, conjuntos de dados com apenas chaves do tipo `long long int` também foram gerados, a fim de avaliar o desempenho dos algoritmos em ambos os cenários.

Nos experimentos, os dados foram gerados segundo as distribuições: uniforme, exponencial, inversamente ordenada, e quase ordenada, conforme detalhado posterior-

mente. Nos experimentos com diversos tamanhos de entradas, os dados foram gerados segundo uma distribuição normal com média de 1×10^9 e desvio padrão de 125×10^6. Essa escolha de distribuição visa introduzir uma maior variabilidade nos dados, tornando o problema de ordenação mais desafiador e dificultando o balanceamento da carga de trabalho entre as *threads*. Os experimentos têm como objetivo avaliar a escalabilidade dos algoritmos em diferentes cenários, por meio da variação dos seguintes parâmetros:

1. Tamanho do conjunto de dados: variando de 1 milhão a 256 milhões de elementos, gerados a partir de uma distribuição normal e processados utilizando 48 *threads*.
2. Número de *threads*: variando de 1 a 48, processando um conjunto de 32 milhões de elementos gerados por uma distribuição normal.
3. Distribuição dos dados: geração de 128 milhões de elementos a serem processados por 48 *threads*. As seguintes distribuições foram consideradas:
   (a) **Uniforme**: os valores das chaves são distribuídos de forma homogênea ao longo do intervalo.
   (b) **Exponencial**: os valores das chaves seguem uma distribuição exponencial com parâmetro $\lambda = 1 \times 10^9$, resultando em predominância de valores pequenos e alta taxa de repetição.
   (c) **Inversamente ordenada**: os elementos estão ordenados em ordem decrescente.
   (d) **Quase ordenada**: sequência majoritariamente ordenada, com 10% dos elementos posicionados aleatoriamente.

Com o objetivo de tornar os testes mais realistas, cada algoritmo foi executado 20 vezes dentro do programa do experimento, simulando o comportamento típico de aplicações que realizam múltiplas ordenações durante sua execução. Cada *script* foi repetido 10 vezes, totalizando 200 execuções do algoritmo para cada experimento. Para cada chamada da função de ordenação foi utilizado um conjunto de dados distinto, gerados em vetores distintos para não haver efeito de reutilização de dados em *cache* advindos de chamadas anteriores. No entanto, os mesmos conjuntos foram reutilizados entre os diferentes algoritmos, a fim de garantir uma comparação justa e consistente dos resultados.

Em cada experimento, foi calculada a vazão média, expressa em milhões de elementos processados por segundo (MEPS). Intervalos de confiança de 95% também foram estimados. No entanto, como em todos os casos não foram observadas variações superiores a 3% em relação à média, esses valores não foram reportados.

O ambiente de execução foi composto por um processador Intel Xeon Gold 6252 com 48 núcleos, localizados em dois *chips* de processamento, sem a utilização de *hyper-threading*. O processador possui 32KB de *cache* L1d e L1i, *cache* L2 de 1MB e *cache* L3 de 32MB. O sistema operacional utilizado foi o Red Hat Fedora 8.8 e compilador utilizado foi o Intel oneAPI DPC++/C++ 2025.0.4.

A quantidade de partições utilizada na primeira parte do algoritmo foi escolhida a partir da seguinte fórmula:

$$\text{Número de Partições} = \frac{\text{Número de Elementos} \times 9}{\text{Tamanho Ideal de Faixa}}$$

As partições foram agrupadas em faixas de 40 mil elementos, de modo que cada faixa possa ser alocada inteiramente em *cache* L3 do processador, mesmo no cenário com

dados no formato chave-valor. Essa estratégia permite que todas as *threads* realizem a ordenação de suas respectivas faixas sem a necessidade de acessar a memória principal, melhorando o desempenho. Nesse contexto, a fórmula utilizada contribui para que, em média, cada faixa seja composta por 9 mini-partições, o que reserva margem para variações no conjunto de dados e aumenta as chances de que as faixas mantenham o tamanho próximo do tamanho ideal escolhido.

5. Resultados e Discussões

Os resultados da comparação entre os algoritmos de ordenação, considerando a variação na quantidade de elementos, são apresentados nas Tabelas 1 e 2 e Figuras 2 e 3. No cenário de ordenação chave-valor, o *mppSort* apresentou desempenho superior em todos os testes realizados.

Algoritmo	Quantidade Elementos (Pares Chave-Valor: Chave: 64 bits, Valor: 32 bits) ------ 48 threads								
	1 Milhão	2 Milhões	4 Milhões	8 Milhões	16 Milhões	32 Milhões	64 Milhões	128 Milhões	256 Milhões
std::par (tbb)	108.79	142.06	165.53	188.00	167.38	178.45	156.66	167.57	161.28
IPS4o	79.17	170.08	262.84	380.23	392.69	434.03	430.69	422.08	419.22
mppSort	**295.46**	**332.69**	**439.54**	**464.34**	**532.14**	**577.36**	**614.04**	**649.27**	**679.50**
Speedup (mppSort/ melhor)	2.72	1.96	1.67	1.22	1.36	1.33	1.43	1.54	1.62

Tabela 1. Resultados dos experimentos variando a quantidade de elementos (chave-valor). Os experimentos foram executados com 48 *threads*, e a vazão reportada em milhões de elementos processados por segundo (MEPS).

Algoritmo	Quantidade de Elementos (Chave: 64 bits) ------ 48 threads								
	1 Milhão	2 Milhões	4 Milhões	8 Milhões	16 Milhões	32 Milhões	64 Milhões	128 Milhões	256 Milhões
std::par (tbb)	134.65	183.99	209.43	262.94	253.05	260.84	240.31	245.43	225.1303
IPS4o	69.16	172.68	314.32	483.57	609.74	654.35	693.65	706.14	674.964
mppSort	**335.74**	**404.45**	**563.79**	**735.38**	**849.25**	**899.16**	**950.61**	**969.99**	**956.44**
Speedup (mppSort/ melhor)	2.49	2.20	1.79	1.52	1.39	1.37	1.37	1.37	1.42

Tabela 2. Resultados dos experimentos variando a quantidade de elementos (somente chave). Os experimentos foram executados com 48 *threads*, e a vazão reportada em milhões de elementos processados por segundo (MEPS).

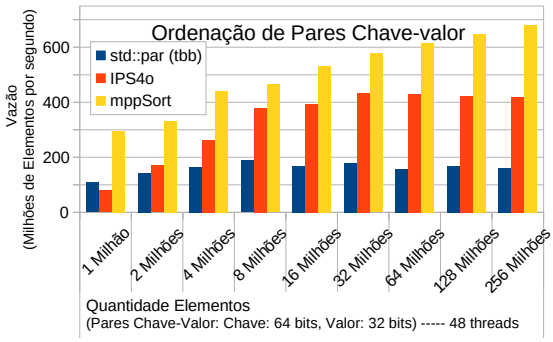

Figura 2. Gráfico dos experimentos variando a quantidade de elementos (chave-valor).

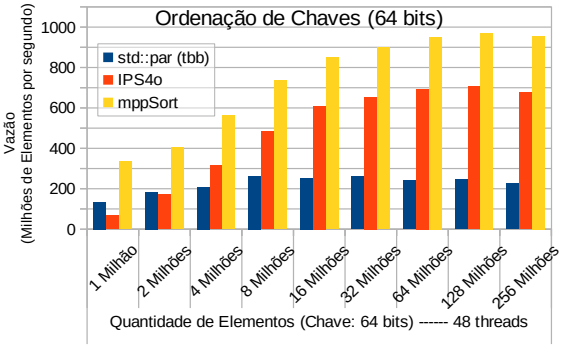

Figura 3. Gráfico dos experimentos variando a quantidade de elementos (somente chave).

O *mppSort* destacou-se especialmente para conjuntos menores, alcançando uma aceleração de 2.72 vezes em relação ao `std::par` no caso de 1 milhão de elementos.

Para conjuntos maiores, também manteve desempenho competitivo, com aceleração de até 1.62 vezes em relação ao *IPS4o* para 256 milhões de elementos. Na comparação entre o *IPS4o* e o `std::par`, observa-se que o `std::par` foi mais eficiente para conjuntos menores, enquanto o *IPS4o* apresentou melhor desempenho para volumes maiores de dados.

Algoritmo	32 milhões de elementos (Pares Chave-Valor: Chave: 64 bits, Valor: 32 bits)						
	QUANTIDADE DE THREADS						
	1	2	4	8	16	32	48
std::par (tbb)	5.70	11.18	21.14	39.97	72.12	107.91	122.69
IPS4o	23.79	45.52	86.54	158.85	295.71	423.24	431.29
mppSort	**24.35**	**45.52**	**85.80**	**163.56**	**301.84**	**493.85**	**593.13**
Speedup (mppSort/ melhor)	1.02	1.00	0.99	1.03	1.02	1.17	1.38

Tabela 3. Resultados dos testes variando a quantidade de *threads* para pares chave-valor. Vazão reportada em milhões de elementos processados por segundo (MEPS).

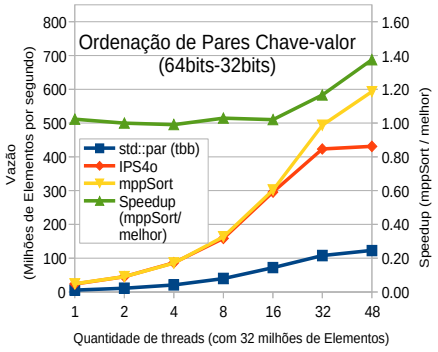

Figura 4. Resultados variando a quantidade de *threads* (chave-valor).

Algoritmo	QUANTIDADE DE THREADS						
	1	2	4	8	16	32	48
std::par (tbb)	6.08	11.90	22.84	43.57	83.62	146.20	183.94
IPS4o	32.08	**62.43**	**118.51**	**223.14**	405.75	600.78	639.70
mppSort	**32.11**	60.62	113.21	218.71	**411.49**	**665.28**	**869.56**
Speedup (mppSort/ melhor)	1.00	0.97	0.96	0.98	1.01	1.11	1.36

Tabela 4. Resultados dos testes variando a quantidade de *threads* (somente chaves). Vazão reportada em milhões de elementos processados por segundo (MEPS).

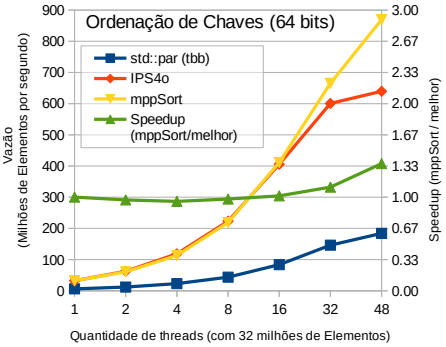

Figura 5. Resultados variando a quantidade de *threads* (somente chave).

No cenário de ordenação de apenas chaves, o *mppSort* também apresentou o melhor desempenho em todos os testes realizados. Embora as acelerações tenham sido menos expressivas em comparação ao cenário chave-valor, o *mppSort* alcançou aceleração de 2.49 vezes em relação ao `std::par` para 1 milhão de elementos, e de 1.42 vezes em relação ao *IPS4o* para 256 milhões de elementos.

Os experimentos com variação na quantidade de *threads* são apresentados nas Tabelas 3 e 4 e Figuras 4 e 5. No cenário chave-valor, o *mppSort* apresentou desempenho semelhante ao *IPS4o* na maioria dos testes, superando-o com mais de 32 threads. Esses resultados indicam que o *mppSort* é o algoritmo que apresentou melhor escalabilidade na ordenação chave-valor com o aumento no número de *threads*, alcançando uma aceleração de 24.36 vezes em relação à sua execução com uma única *thread*, enquanto o `std::sort` e o *IPS4o* atingiram acelerações de 21.52 vezes e 18.13 vezes, respectivamente. No cenário com apenas chaves e pequeno número de threads, o *mppSort* também apresentou desempenho comparável ao do *IPS4o*, superando-o quando o número de

Algoritmo	Distribuição (128 milhoes, pares chave-valor)				
	Uniforme	Normal	Exponencial	Inversamente Ordenada	Quase Ordenada
std::par (tbb)	119.00	245.43	122.47	134.07	226.01
IPS4o	423.99	706.14	424.06	442.08	488.99
mppSort	**574.19**	**969.99**	**586.86**	**771.15**	**821.88**
Speedup (mppSort/ melhor)	1.35	1.37	1.38	1.74	1.68

Tabela 5. Vazão em milhões de elementos processados por segundo (MEPS) dos algoritmos de ordenação sob diferentes distribuições de dados (chave-valor).

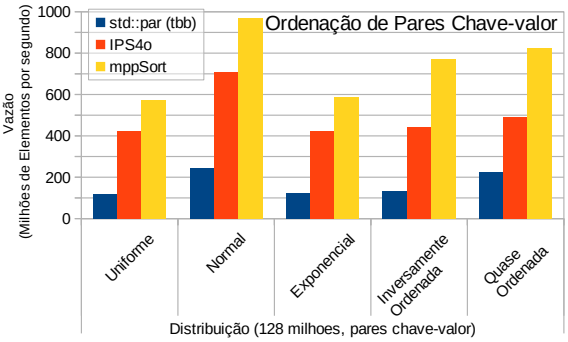

Figura 6. Resultados variando a distribuição (chave-valor).

Algoritmo	Distribuição (128 milhoes de chaves)				
	Uniforme	Normal	Exponencial	Inversamente Ordenada	Quase Ordenada
std::par (tbb)	165.10	245.43	170.03	233.26	384.04
IPS4o	714.52	706.14	708.87	784.43	932.85
mppSort	**796.37**	**933.81**	**829.05**	**1098.81**	**1098.81**
Speedup (mppSort/ melhor)	1.11	1.32	1.17	1.44	1.23

Tabela 6. Vazão em milhões de elementos processados por segundo (MEPS) dos algoritmos de ordenação sob diferentes distribuições de dados (somente chave).

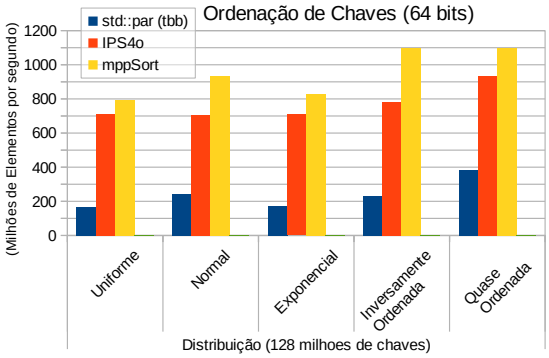

Figura 7. Resultados variando a distribuição (somente chave).

threads se aproxima da quantidade máxima de núcleos de processamento, assim utilizando o processador de maneira eficiente.

Por fim, os resultados dos experimentos com dados gerados a partir de diferentes distribuições estão nas Tabelas 5 e 6 e Figuras 6 e 7. Nota-se que o *mppSort* superou os demais algoritmos em todos os cenários avaliados. Esse desempenho evidencia a versatilidade do algoritmo, especialmente na sua capacidade de balancear a carga de trabalho mesmo em conjuntos de dados mais desafiadores. Em especial, observamos o bom desempenho do *mppSort* quando testado com distribuições de dados quase ordenados. Esse tipo de distribuição é importante em diversas aplicações onde os dados são ligeiramente modificados em um *loop* de iterações e necessitam de ordenação a cada iteração. Essa situação se apresenta comumente em simulações científicas, por exemplo. Assim, a cada iteração, dados quase ordenados podem ser processados com boa eficiência.

Observa-se também que a vazão do *mppSort* é maior no processamento de chaves apenas, assim como acontece com a maioria dos algoritmos de ordenação. Porém, a aceleração do *mppSort*, no cenário com apenas chaves, foi inferior à observada no caso chave-valor. Isso indica que, embora o *mppSort* apresente desempenho superior aos demais algoritmos em todos os testes, seu destaque é ainda mais expressivo na ordenação de pares chave-valor.

6. Conclusão

Este trabalho apresentou o *mppSort*, um algoritmo paralelo eficiente para a ordenação de pares chave-valor em arquiteturas *multicore*. A estratégia de divisão dos dados em mini-

partições demonstrou ser eficaz para melhorar o balanceamento da carga entre as *threads* e maximizar o desempenho da ordenação paralela. Além disso, a adoção de histogramas e somas de prefixos e barreiras como estratégia de balanceamento, eliminou a necessidade de sincronização de baixa granularidade, como *locks*. O agrupamento das mini-partições em conjuntos de "tamanho ideal" possibilitou a maximização da vazão de ordenação por meio do aproveitamento eficiente da memória *cache*.

Os experimentos realizados mostraram que o algoritmo atinge boa aceleração e escalabilidade em diferentes cenários, incluindo variação no tamanho dos dados, número de *threads* e distribuições dos dados de entrada. Os resultados indicam que o *mppSort* é uma alternativa promissora para aplicações que exigem ordenação eficiente de grandes volumes de dados em ambientes paralelos, tanto para ordenação de pares chave-valor quanto para ordenação de chaves apenas.

Como trabalhos futuros, pretende-se investigar o uso de diferentes algoritmos de ordenação na etapa final, buscando combinações ainda mais eficientes para diferentes tamanhos de entrada. Além disso, iniciamos o desenvolvimento de uma versão distribuída do *mppSort* [Cordeiro et al. 2025], voltada para execução em ambientes com múltiplos nodos de execução (*clusters* computacionais), visando a ordenação eficiente de conjuntos de dados em larga escala. O principal desafio nesse contexto está em se obter boa escalabilidade do algoritmo em face à maior latência para a distribuição dos dados entre nodos computacionais, mesmo no ambiente de rede local de alta velocidade.

Agradecimentos

Este trabalho foi 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). Os autores agradecem ao Laboratório Nacional de Computação Científica (LNCC/MCTI, Brasil) pela disponibilização de recursos de HPC, que contribuíram para obtenção dos resultados finais nos experimentos deste trabalho.

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 (TOPC)*, 4(1):1–44.

Axtmann, M., Witt, S., Ferizovic, D., and Sanders, P. (2022). Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing (TOPC)*, 9(1):1–62.

Bing-Chao, H. and Knuth, D. E. (1986). A one-way, stackless quicksort algorithm. *BIT Numerical Mathematics*, 26(1):127–130.

Bingmann, T., Eberle, A., and Sanders, P. (2017). Engineering parallel string sorting. *Algorithmica*, 77:235–286.

Blanco, R. M., Cordeiro, M. B., and Zola, W. M. N. (2025). Ordenação distribuída de pares chave-valor utilizando MPI. In *Escola Regional de Alto Desempenho da Região Sul (ERAD-RS)*. SBC.

Casimiro, A., Broinizi, M. E., and Ferreira, J. (2014). Principais componentes na ordenação de anúncios: Um experimento em ambiente real de publicidade computacional. In *Anais do XXIX Simpósio Brasileiro de Banco de Dados (SBBD 2014)*, pages 147–156. SBC.

Cavalcante, P. S. and Pinheiro, W. A. (2013). Mecanismo de encadeamento de notícias por reconhecimento de implicação textual. In *Anais do XXVIII Simpósio Brasileiro de Banco de Dados (SBBD 2013)*, pages 157–162. SBC.

Cordeiro, M., Blanco, R., and Zola, W. (2025). Algoritmo paralelo e distribuído para ordenação chave-valor. In *Anais da XX Escola Regional de Banco de Dados (ERBD 2025)*, pages 133–136. SBC.

Cordeiro, M. and Zola, W. (2025). Multiparticionamento de dados em GPU. In *Anais da XXV Escola Regional de Alto Desempenho da Região Sul*, pages 165–166. SBC.

Edelkamp, S. and Weiß, A. (2019). Blockquicksort: Avoiding branch mispredictions in quicksort. *Journal of Experimental Algorithmics (JEA)*, 24:1–22.

Ferreira, D. P., González, S., and Vieira, G. M. D. (2023). Replicação descentralizada em bancos de dados distribuídos usando o algoritmo Paxos. In *Anais do XXXVIII Simpósio Brasileiro de Banco de Dados (SBBD 2023)*, pages 282–294. SBC.

Frazer, W. D. and McKellar, A. C. (1970). Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507.

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.

Hou, K., Wang, H., and Feng, W.-c. (2015). ASPaS: A framework for automatic SIMDization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 383–392.

Huang, J. S. and Chow, Y. C. (1983). Parallel sorting and data partitioning by sampling. In *Proceedings of the Seventh International Computer Software and Applications Conference (COMPSAC)*, pages 627–631. IEEE.

Kokot, M., Deorowicz, S., and Długosz, M. (2018). Even faster sorting of (not only) integers. In *Man-Machine Interactions 5: 5th International Conference on Man-Machine Interactions, ICMMI 2017 Held at Kraków, Poland, October 3-6, 2017*, pages 481–491. Springer.

Kushagra, S., López-Ortiz, A., Qiao, A., and Munro, J. I. (2014). Multi-pivot quicksort: Theory and experiments. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 47–60. SIAM.

Merrill, D. and Grimshaw, A. (2011). High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272.

Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993.

Novaes, L. P., Vianna, D., and da Silva, A. (2023). Modelagem de tópicos para a tarefa de recuperação de casos legais. In *Anais do XXXVIII Simpósio Brasileiro de Banco de Dados (SBBD 2023)*, pages 128–140. SBC.

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

Rodrigues, E., Pires, C. E., and Filho, D. N. (2024). Evolução incremental de esquemas de banco de dados orientado a documentos. In *Anais do XXXIX Simpósio Brasileiro de Banco de Dados (SBBD 2024)*, pages 260–273. SBC.

Sanders, P. and Winkel, S. (2004). Super scalar sample sort. In *European Symposium on Algorithms*, pages 784–796. Springer.

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

Sundar, H., Malhotra, D., and Biros, G. (2013). HykSort: a new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th international ACM conference on international conference on supercomputing*, pages 293–302.

Tavares, T., Belloze, K., Goldschmidt, R., and Soares, J. (2024). Avaliação de diferentes técnicas de agrupamento no contexto da imputação em cascata. In *Anais do XXXIX Simpósio Brasileiro de Banco de Dados (SBBD 2024)*, pages 687–693. SBC.