Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Paralela

(*OpenMP*)

Bucket-Sort

Raphael Oliveira A78848
Francisco Araújo A79821

14 de Dezembro de 2018

Conteúdo

Resumo

O presente documento incide sobre a área de Computação Paralela, referente ao trabalho prático desta unidade curricular, relacionado com a programação com memória partilhada, mais especificamente, a implementação do algoritmo *bucket-sort*, com o *OpenMP*.

Numa primeira fase, é descrito o algoritmo, sendo identificadas todas as decisões tomadas pela equipa em relação ao desenho do mesmo, bem como a sua implementação (sequencial e paralela), neste caso, em *C++*.

Numa fase término, é efetuada a apresentação de gráficos, demonstrando a relação entre a versão sequencial e a versão paralela, para vários *inputs*, bem como a sua possível análise dos resultados.

1 Descrição do Algoritmo

O *bucketsort* consiste num algoritmo de ordenação, em que os elementos do vetor original, recebido como *input*, são distribuídos por um determinado número de *buckets*. Após essa distribuição, cada *bucket* (não vazio) é ordenado, seja por outro algoritmo de ordenação como também pelo mesmo algoritmo, sendo aplicada a recursividade do mesmo. Finalizada a ordenação de cada *bucket*, os elementos desses *buckets* são introduzidos no vetor original, terminando assim este processo de ordenação.

Nas seguintes secções serão apresentados estes processos, sendo que cada processo atravessou várias fases de desenvolvimento.

1.1 Inicialização de um vetor de *buckets*

Tal como dito anteriormente, a primeira fase deste algoritmo é a inicialização do vetor de *buckets* de forma a inserir lá os elementos do vetor original, na próxima fase.

Para identificar o melhor número de *buckets*, para um dado tamanho do vetor original, realizaram-se alguns testes, sendo exibidos na secção Anexos, resultando em:

- **nBuckets = (int)sqrt(sizeInput) + 1**;

Desta forma, idealizou-se as seguintes estruturas, para guardar os elementos:

- **buckets[sizeBuckets]**: em que *sizeBuckets* é o resultado da multiplicação do tamanho do vetor original pelo número de *buckets*, determinado anteriormente. Desta forma cada *bucket* tem no máximo o tamanho do vetor original.
- **contadores[nBuckets]**: quantidade de elementos que cada *bucket* possui, de forma a facilitar a próxima fase de inserção.

Assim, a inicialização dos *buckets*, é, na prática, a inicialização dos contadores de cada *bucket*:

```
void criarBuckets(int contadores[], int nBuckets){
        for (int i = 0; i < nBuckets; i++){
                contadores[i] = 0;
        }
}
```

Listing 1: Função *criarBuckets*

Com esta estrutura ***buckets***, em vez de, por exemplo, uma matriz ou uma estrutura que contivesse os dados necessários para este algoritmo, permite uma melhor escalabilidade e uma melhor localidade na memória. Neste seguimento, é possível justificar os resultados posteriormente evidenciados (*cache misses*).

1.2 Inserção dos elementos no vetor de *buckets*

Realizada a fase anterior, pode-se proceder à inserção dos elementos do vetor original em cada *bucket*.

Nesta fase existiram 3 versões de implementação, cujos resultados serão exibidos posteriormente, sendo que cada versão possuiu uma coisa em comum: a posição de inserção de um elemento num determinado *bucket*, isto é, para um determinado ***índice i***, a inserção desse elemento é realizada na posição ***buckets[i*sizeInput + contadores[i]]***, em que, após essa inserção, o *contadores[i]* é incrementado.

Para calcular esse índice foram utilizados 2 algoritmos (em 3 fases distintas) também pressupondo que o vetor original não possui elementos maiores que o tamanho do vetor (sendo a função que gera os números aleatórios implementada dessa forma).

- **1ª versão:** O número de *buckets* é determinado pela raíz quadrada do tamanho do vetor original e para cada elemento do vetor original, é calculada a sua raíz quadrada, sendo esse o índice calculado. Desta forma era inserido na posição indicada anteriormente. Com este algoritmo, para tamanhos extensos do vetor original, os elementos ficariam maioritariamente nos últimos *buckets*, resultando numa distribuição não uniforme e, também, o cálculo da raíz quadrada para cada elemento tornaria a função ineficiente, como se pode comprovar nos resultados obtidos, evidenciados posteriormente.

- **2ª versão:** O número de *buckets* é aleatório (realizado numa fase de testes, para encontrar o número ideal), sendo que para cada elemento do vetor original, o índice de inserção num determinado *bucket* é calculado da seguinte forma:

  Nº do bucket = Elemento vetor original / Tamanho do vetor original

  *Índice = Nº do bucket * Nº total de buckets*

- **3ª versão (final):** Esta última versão é uma junção das outras duas versões, ou seja, o número total de *buckets* é determinado pela raíz quadrada do tamanho do vetor original, enquanto que o cálculo do índice de inserção, para cada elemento, é equivalente ao da 2ª versão.

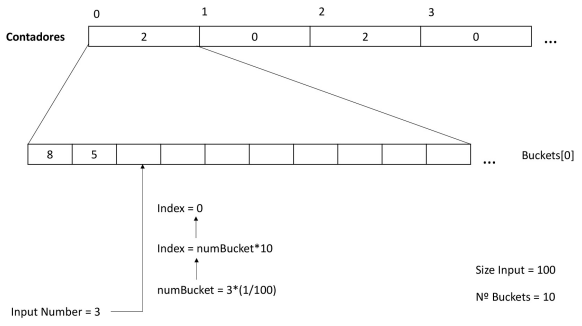

Figura 1: Esquema do algoritmo de inserção

1.3 Ordenação dos elementos em cada vetor de *buckets*

Finalizada a distribuição dos elementos do vetor original por cada *bucket*, é necessário proceder à ordenação dos elementos de cada *bucket*.

Nesta fase, decidiu-se não utilizar o mesmo algoritmo recursivo, pois isso causaria uma enorme ineficiência. Por outro lado, decidiu-se utilizar o algoritmo de ordenação *mergesort*, reutilizando código já implementado, que depois de alguma pesquisa, verificou-se que seria o algoritmo mais eficiente.

1.4 Colocação dos elementos no vetor original

Depois de realizar a ordenação de cada *bucket*, procede-se à introdução dos elementos de cada *bucket*, ordenadamente, no vetor original.

Neste seguimento, as seguintes variáveis são importantes para esta última fase:

- ***c***: inicializado a zero, incrementando cada vez que é inserido um elemento no vetor original.
- ***pos = i*sizeInput*** : posição inicial de um *bucket*.
- ***max = contadores[i]*** : máximo de elementos num certo *bucket*.

```
void ordenaInput(int buckets[],int contadores[], int numerosInput[], int
↪  sizeInput, int nBuckets){
    int c=0, pos, max;
    for (int i = 0; i < nBuckets; i++){
        pos = i*sizeInput;
        max = contadores[i];
        for (int j = 0; j < max ; j++){
            numerosInput[c++] = buckets[pos+j];
        }
    }
}
```

Listing 2: Função *ordenaIput*

2 Implementação da versão paralela

Para proceder à implementação da versão paralela analisou-se o comportamento de cada função utilizada para este algoritmo de ordenação e a sua influência no tempo total da ordenação.

Nestes testes utilizaram-se vários tamanhos de *input*, cada um respetivo para cada nível de *cache*: 100, 1000, 10000. Como os resultados, tanto em termos de tempo como em relação às *cache misses* para estes tamanhos, foram bastante limitados, só serão apresentados nesta secção os resultados para o tamanho de um milhão (sendo que as *caches misses* continuaram a ser mínimas):

Função	Tempo sequencial(µs)
criarBuckets	1
insereBuckets	9678
ordenaBuckets	67696
ordenaInput	401
total=	**77776**

Tabela 1: Tempo sequencial para tamanho de 1000000

Como se pode verificar, a função *ordenaBuckets* é a que ocupa maior parte do tempo deste algoritmo. Desta forma, esta foi a primeira função a ser paralelizada.

Neste seguimento, utilizou-se, numa primeira fase, a seguinte diretiva para paralelizar esta função:

```
#pragma omp parallel for schedule (dynamic,nBuckets/numThreads)
```

Listing 3: Diretiva de paralelização, 1ª fase

Depois de alguns testes, percebeu-se que a diretiva com ***schedule(static)*** é mais eficiente. Conclui-se assim que os elementos encontram-se bem distribuídos pelos *buckets* e o *overhead* da diretiva *dynamic* torna menos eficiente esta paralelização.

Através desta paralelização, o tempo resultante é demonstrado na seguinte tabela:

Nº Threads	Função	Tempo sequencial(µs)	Tempo paralelo(µs)
24	criarBuckets	1	1
	insereBuckets	9678	9930
	ordenaBuckets	67696	8825
	ordenaInput	401	745
	total=	**77776**	**19501**

Tabela 2: Paralelização, com 24 *threads*

Depois desta paralelização, tentou-se efetuar paralelizações tanto nas funções *insere-Buckets* (sendo a segunda função que ocupa mais tempo) e *ordenaInput*, mas em ambos os casos se deparou com *data races*, pois podem existir acessos concorrentes aos vetores, sendo que a paralelização aplicada, de forma a que o vetor original ficasse corretamente ordenado, piorava o tempo total.

3 Apresentação e Análise de Resultados

Finalizadas as fases de ambas as implementações, é necessário agora apresentar os resultados obtidos, isto é, os *SpeedUps*, relação entre os tempos da versão sequencial e os tempos da versão paralela, para cada tamanho de vetor original, e a possível análise aos mesmos.

Em cada teste, foram realizadas cinco repetições, sendo obtido o melhor resultado através da **mediana** envolvendo esses resultados, em que cada repetição se executou a versão sequencial e a versão paralela (os resultados aqui evidenciados são apenas os da 3ª versão) para um vetor original idêntico, intercaladas com limpeza da *cache*, para obter resultados mais consistentes.

Ainda, estes testes foram executados num dos nodos do *cluster*, com as seguintes principais especificações (nos anexos encontram-se outras):

- **Nodo:** r431
- **L1d cache:** 32K
- **L1i cache:** 32K
- **L2 cache:** 256K
- **L3 cache:** 12288K
- **Versão papi:** 5.5.0
- **Versão compilador(*gcc*):** 4.8.2

Assim, foram obtidos os seguintes resultados:

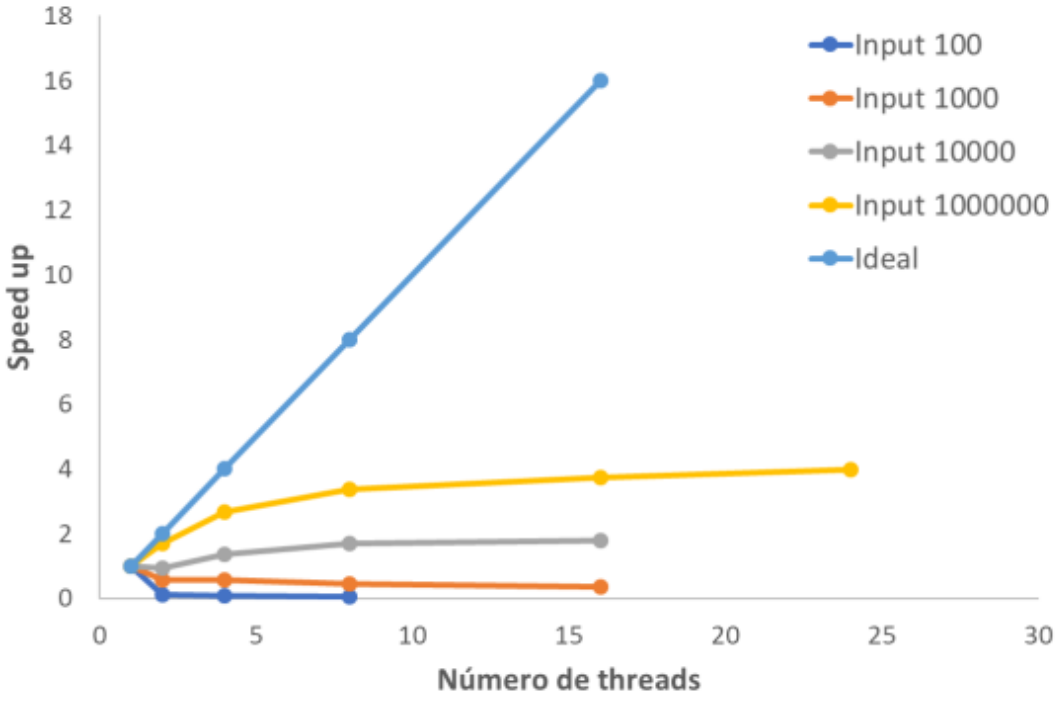

Figura 2: *SpeedUp* para tamanhos pequenos

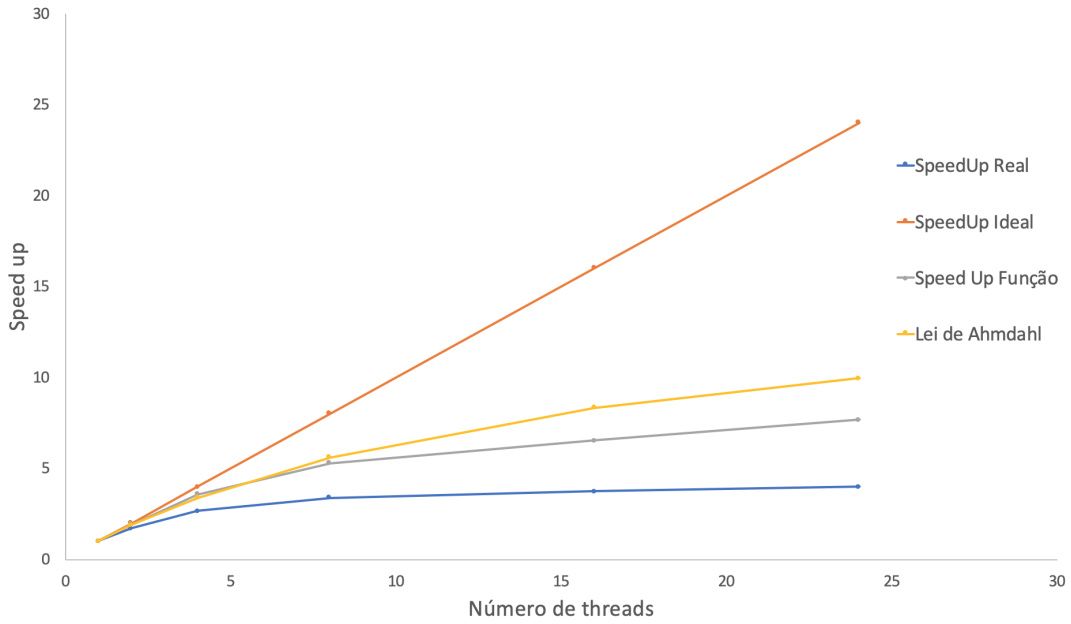

Figura 3: *SpeedUp* da Função *ordenaBuckets*

Como se pode verificar, pela Lei de *Ahmdahl*, o algoritmo implementado nunca irá apresentar muita escalabilidade pois o código não é 100% paralelizado.

Ainda assim, pode-se verificar, pelo 1º gráfico, que para tamanhos maiores o algoritmo já apresenta um nível de resultados superior. Sendo assim, testou-se para tamanhos maiores, para tal se comprovar:

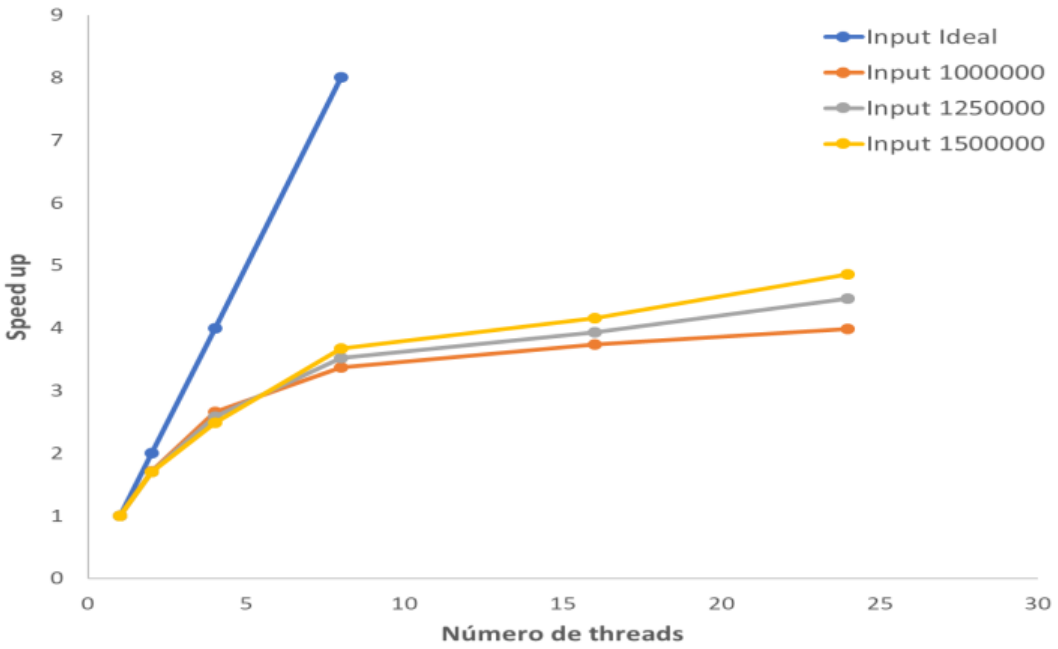

Figura 4: *SpeedUp* para tamanhos maiores

Através da imagem anterior, comprova-se a escalabilidade do algoritmo para tamanhos maiores. Mesmo assim, os *speedups* obtidos estão muito longe de serem os ideais.

Estes resultados podem ser devidos a:

- Para **tamanhos pequenos**, o algoritmo não escala pois o *overhead* de criar as *threads* necessárias para a paralelização é muito grande quando comparado ao tempo de execução da operação total, isto é, nestes casos existe sobrecarga de paralelismo (**paralelismo de grau fino**).
- ***Lei de Ahmdahl***: Como o código não é 100% paralelizado, o *speedup* nunca poderá ser o ideal, segundo esta lei.
- **Acessos à memória**: Os custos dos acessos à memória são elevados, o que afetará este algoritmo, já que é um algoritmo ***memory bound***.

4 Conclusão

Realizado todo o processo de implementação de ambas as versões, o objetivo prioritário de melhorar ao máximo a parte sequencial e, posteriormente, a parte paralela foi visado.

A equipa já acreditava que o algoritmo não iria possuir grande escalabilidade, pois era caraterizado por ser um algoritmo *memory bound*, mas, ainda assim, para tamanhos extensos, existe uma certa escalabilidade.

Como principais dificuldades, destacaram-se a quantidade de testes que foram efetuados para determinar a viabilidade de uma determinada ação e resultados de difícil interpretação e análise.

A Anexos

A.1 Especificações da máquina

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
CPU socket(s):         2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Stepping:              2
CPU MHz:               1600.000
BogoMIPS:              5066.54
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
```

Listing 4: Especificações da máquina

A.2 Outros Resultados

Nº threads	Função	*Tempo Sequencial(µs)*	*Tempo Paralela(µs)*	***SpeedUp***
1	criarBuckets	1	1	1
	insereBuckets	1	1	1
	ordenaBuckets	5	5	1
	ordenaInput	1	1	1
	total=	**8**	**8**	**1**
2	criarBuckets	1	1	1,00
	insereBuckets	1	1	1,00
	ordenaBuckets	5	59	0,08
	ordenaInput	1	1	1,00
	total=	**8**	**62**	**0,13**
4	criarBuckets	1	1	1,00
	insereBuckets	1	1	1,00
	ordenaBuckets	5	89	0,06
	ordenaInput	1	1	1,00
	total=	**8**	**92**	**0,09**
8	criarBuckets	1	1	1,00
	insereBuckets	1	2	0,50
	ordenaBuckets	5	121	0,04
	ordenaInput	1	1	1,00
	total=	**8**	**125**	**0,06**

Figura 5: ***Input de 100*** - Tempos de ambas as versões

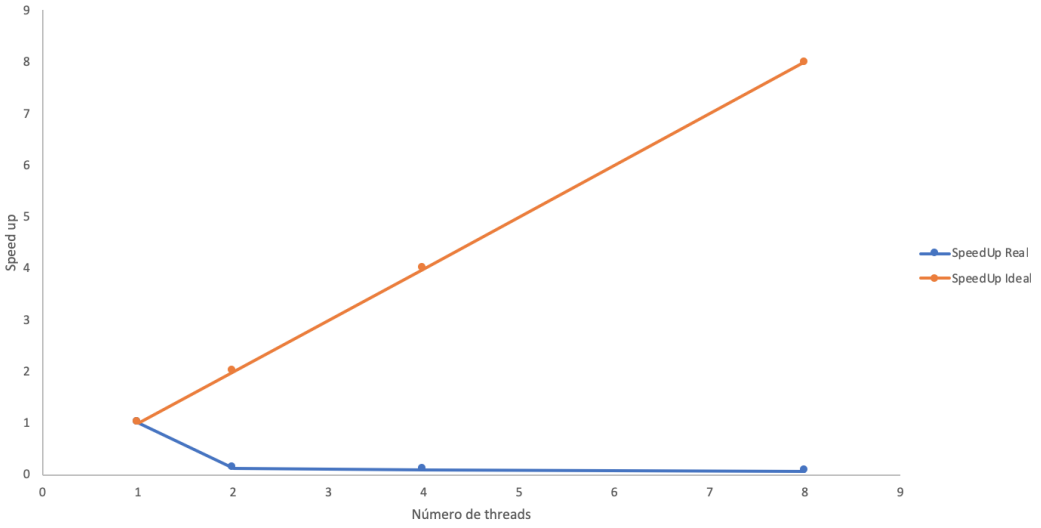

Figura 6: ***Input de 100*** - Gráfico de *SpeedUps*

Nº threads	Função	*Tempo Sequencial(µs)*	*Tempo Paralela(µs)*	***SpeedUp***
1	criarBuckets	1	1	1
	insereBuckets	10	10	1
	ordenaBuckets	46	46	1
	ordenaInput	1	1	1
	total=	**58**	**58**	**1**
2	criarBuckets	1	1	1,00
	insereBuckets	10	11	0,91
	ordenaBuckets	46	87	0,53
	ordenaInput	1	1	1,00
	total=	**58**	**100**	**0,58**
4	criarBuckets	1	1	1,00
	insereBuckets	10	10	1,00
	ordenaBuckets	46	92	0,50
	ordenaInput	1	1	1,00
	total=	**58**	**104**	**0,56**
8	criarBuckets	1	1	1,00
	insereBuckets	10	11	0,91
	ordenaBuckets	46	118	0,39
	ordenaInput	1	1	1,00
	total=	**58**	**131**	**0,44**
16	criarBuckets	1	1	1,00
	insereBuckets	10	10	1,00
	ordenaBuckets	46	148	0,31
	ordenaInput	1	1	1,00
	total=	**58**	**160**	**0,36**

Figura 7: ***Input de 1000*** - Tempos de ambas as versões

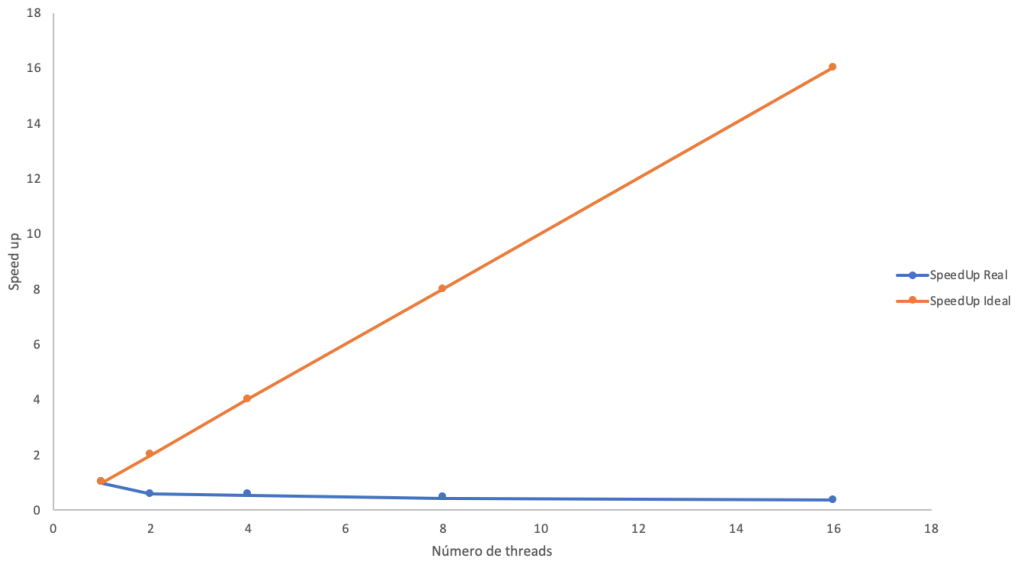

Figura 8: ***Input de 1000*** - Gráfico de *SpeedUps*

Nº threads	Função	*po Sequencia*	*mpo Paralela(*	***SpeedUp***
1	criarBuckets	1	1	1
	insereBuckets	103	103	1
	ordenaBucket	511	511	1
	ordenaInput	4	4	1
	total=	**619**	**619**	**1**
2	criarBuckets	1	1	1
	insereBuckets	103	95	1,08
	ordenaBucket	511	543	0,94
	ordenaInput	4	10	0,40
	total=	**619**	**649**	**0,95**
4	criarBuckets	1	1	1
	insereBuckets	103	96	1,07
	ordenaBucket	511	339	1,51
	ordenaInput	4	11	0,36
	total=	**619**	**447**	**1,38**
8	criarBuckets	1	1	1
	insereBuckets	103	93	1,11
	ordenaBucket	511	259	1,97
	ordenaInput	4	10	0,40
	total=	**619**	**363**	**1,71**
16	criarBuckets	1	1	1,00
	insereBuckets	103	94	1,10
	ordenaBucket	511	233	2,19
	ordenaInput	4	16	0,25
	total=	**619**	**344**	**1,80**

Figura 9: ***Input de 10000*** - Tempos de ambas as versões

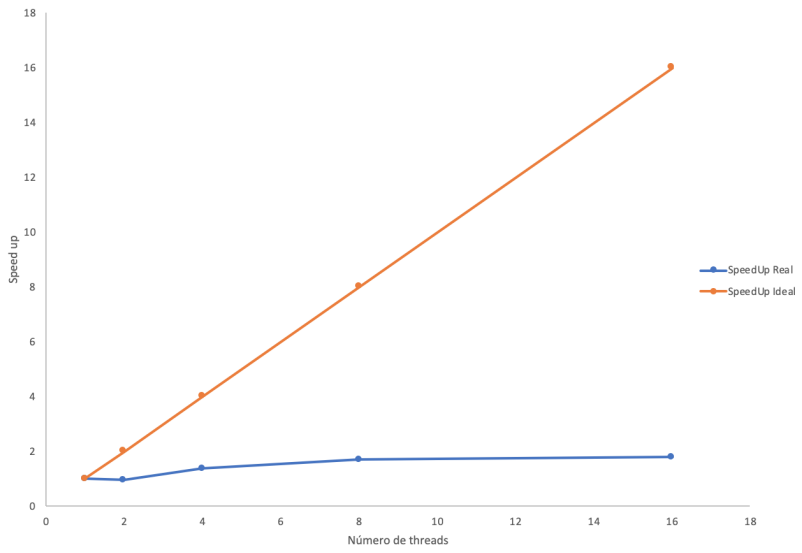

Figura 10: ***Input de 10000*** - Gráfico de *SpeedUps*

Nº threads	Função	*Tempo Sequencial(µs)*	*Tempo Paralela(µs)*	***SpeedUp***
1	criarBuckets	1	1	1
	insereBuckets	9678	9678	1
	ordenaBuckets	67696	67696	1
	ordenaInput	401	401	1
	total=	**77776**	**77776**	**1**
2	criarBuckets	1	1	1
	insereBuckets	9678	9748	0,99
	ordenaBuckets	67696	35203	1,92
	ordenaInput	401	560	0,72
	total=	**77776**	**45512**	**1,71**
4	criarBuckets	1	1	1
	insereBuckets	9678	9731	0,99
	ordenaBuckets	67696	18948	3,57
	ordenaInput	401	592	0,68
	total=	**77776**	**29272**	**2,66**
8	criarBuckets	1	1	1
	insereBuckets	9678	9669	1,00
	ordenaBuckets	67696	12782	5,30
	ordenaInput	401	622	0,64
	total=	**77776**	**23074**	**3,37**
16	criarBuckets	1	1	1,00
	insereBuckets	9678	9659	1,00
	ordenaBuckets	67696	10391	6,51
	ordenaInput	401	764	0,52
	total=	**77776**	**20815**	**3,74**
24	criarBuckets	1	1	1,00
	insereBuckets	9678	9930	0,97
	ordenaBuckets	67696	8825	7,67
	ordenaInput	401	745	0,54
	total=	**77776**	**19501**	**3,99**

Figura 11: ***Input de 1000000*** - Tempos de ambas as versões

Nº threads	*Miss Rate L1 (%)*	*Miss Rate L2 (%)*	*Miss Rate L3 (%)*
1	0,1844	0,0309	0,0005

Figura 12: ***Input de 1000000*** - *Miss Rates*