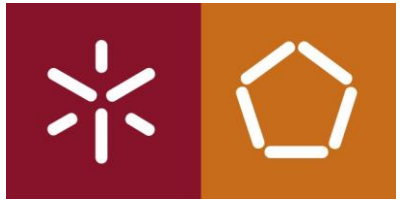


BUCKET-SORT

Mestrado Integrado em Engenharia Informática

Computação Paralela

(OpenMP)

Raphael Oliveira A78848
Francisco Araújo A79821

ÍNDICE

INICIALIZAÇÃO DO VETOR BUCKETS

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

buckets[sizeBuckets]:

sizeBuckets = sizeInput × nBuckets

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.

INSERÇÃO DOS ELEMENTOS NO VETOR BUCKETS

1ª Versão

- Cálculo da raiz quadrada ineficiente e uma distribuição não uniforme;

2ª Versão

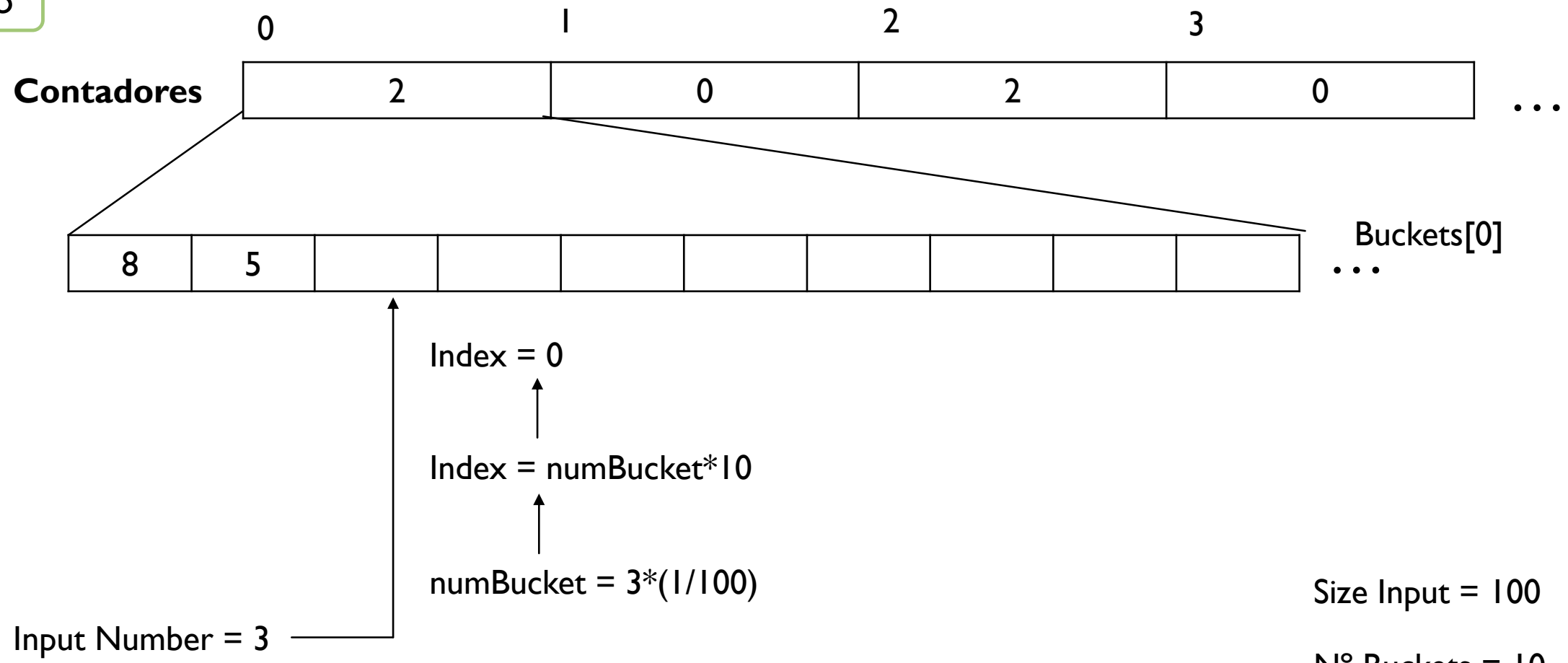
- Número de buckets aleatório. O índice é determinado por:
  - Nº do Bucket = Elemento Input / Tamanho vetor original
  - Índice = Nº do bucket × Nº total de buckets

3ª Versão

- Número de Buckets calculado no início pela raiz quadrada do vetor original;
- É uma junção da 1ª e da 2ª versão.

INSERÇÃO DOS ELEMENTOS NO VETOR BUCKETS

3ª Versão



ORDENAÇÃO DOS ELEMENTOS EM CADA VETOR DE BUCKETS

Mergesort

Algoritmo de ordenação de cada bucket

Código reutilizado (possível paralelização);

Mais eficiente que a recursividade da bucket-sort;

Mais eficiente que outros algoritmos de ordenação.

COLOCAÇÃO DOS ELEMENTOS NO VETOR ORIGINAL

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.

IMPLEMENTAÇÃO DA VERSÃO PARALELA

1ª fase: Análise

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

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

2ª fase: Implementação

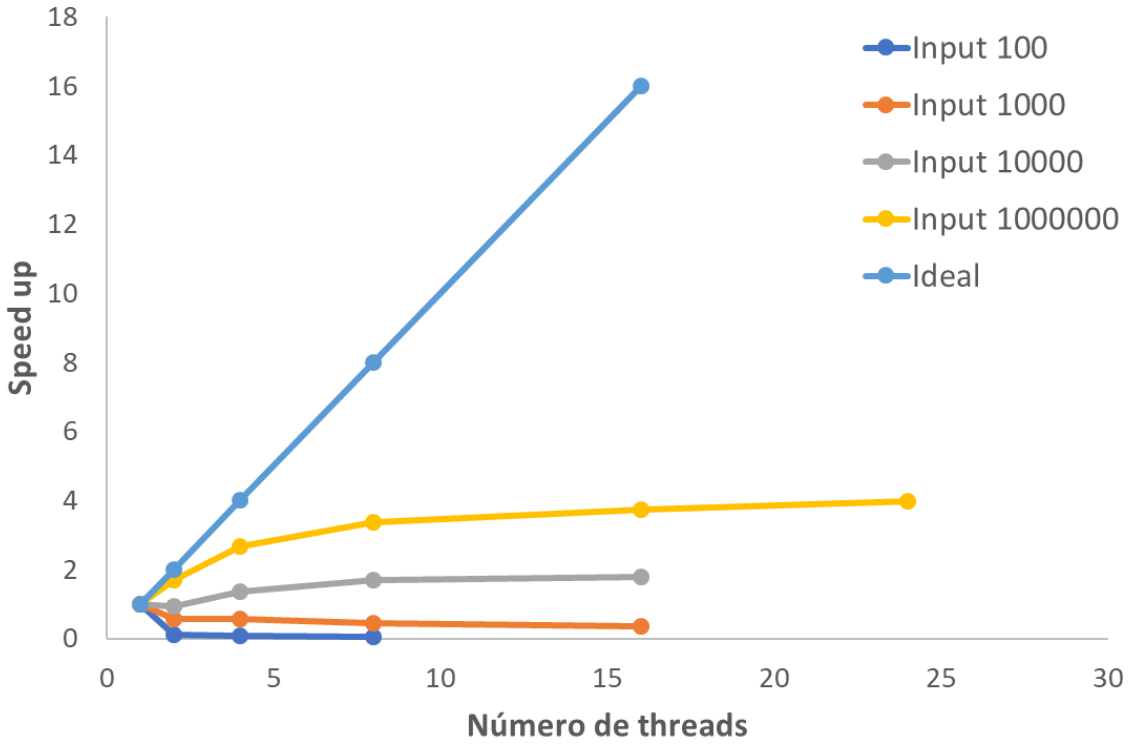
#pragma omp parallel for schedule (static)

3ª fase: Resultados

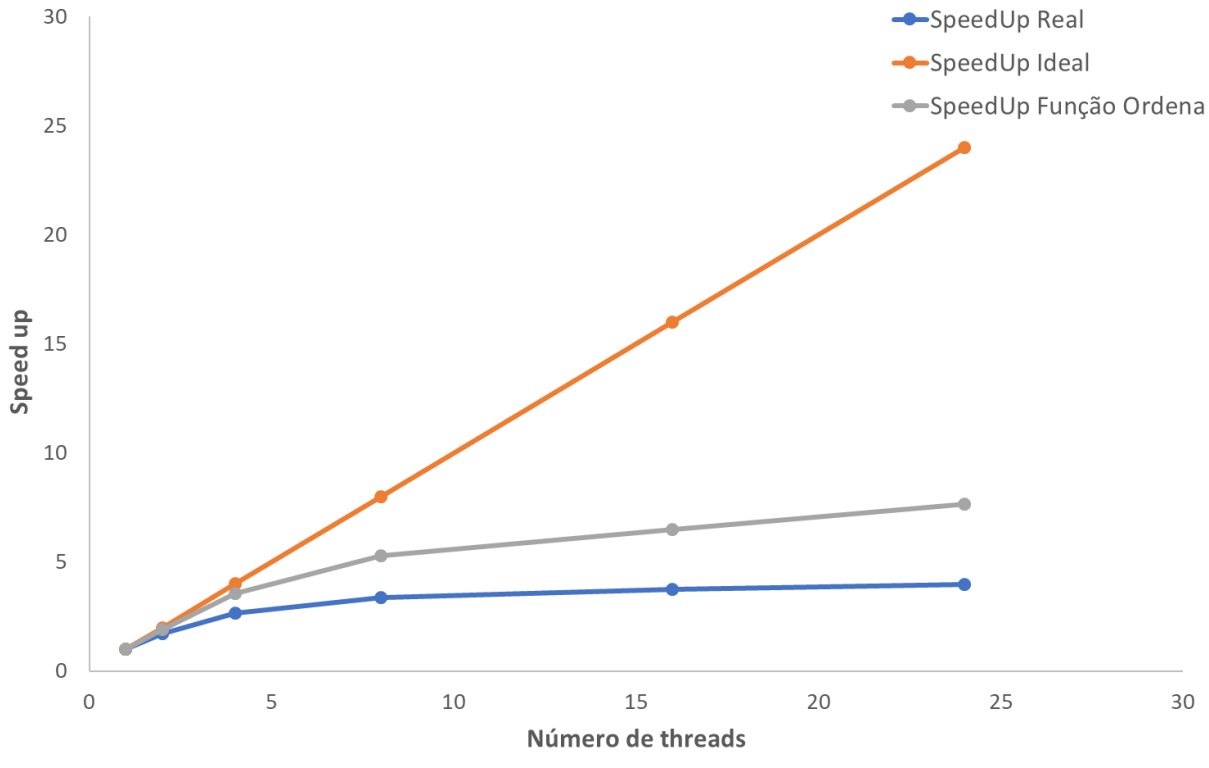
Nº threads	Função	*Tempo Sequencial(µs)*	*Tempo Paralela(µs)*	***SpeedUp***
1	criarBuckets	1	1	1
	insereBuckets	9678	9678	1
	ordenaBuckets	67696	67696	1
	ordenaInput	401	401	1
	total=	**77776**	**77776**	**1**
2	criarBuckets	1	1	1
	insereBuckets	9678	9748	0,99
	ordenaBuckets	67696	35203	1,92
	ordenaInput	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**

APRESENTAÇÃO E ANÁLISE DE RESULTADOS

SpeedUp para tamanhos pequenos

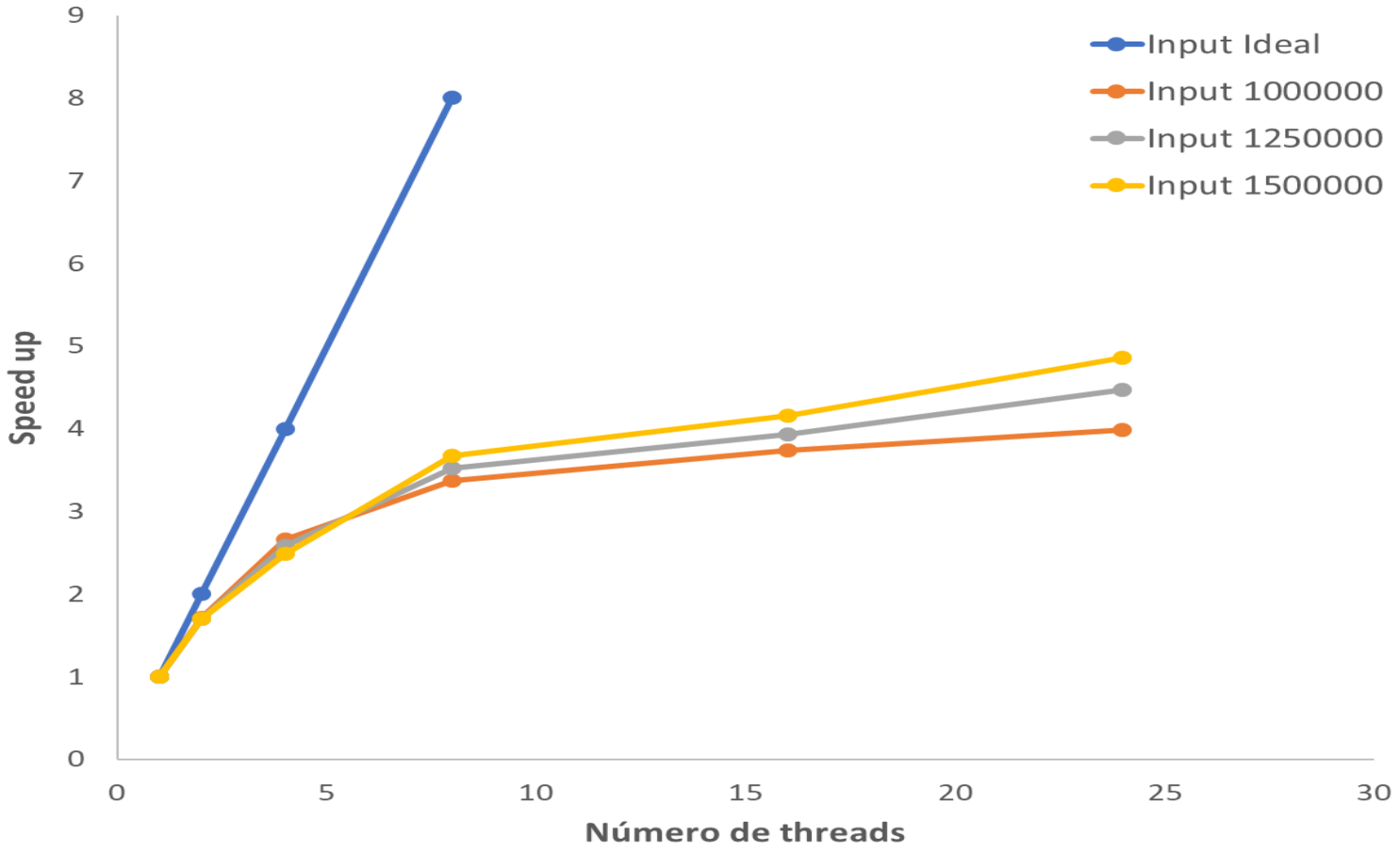


SpeedUp da Função ordenaBuckets



APRESENTAÇÃO E ANÁLISE DE RESULTADOS

SpeedUp para tamanhos maiores



CONCLUSÃO

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*;