Bucket Sort Paralelo

Alana Schwendler

[1]Conceitos de Linguagens de Programação – Prof. Gerson Geraldo H. Cavalheiro
Bacharelado em Ciência da Computação – UFPEL
Pelotas, RS - Brasil

aschwendler@inf.ufpel.edu.br

Resumo. *Trabalho 2 da disciplina de Conceitos de Linguagens de Programação, que visa implementação paralela do algoritmo bucket sort utilizando as ferramentas OpenMP e PThreads.*

1. Bucket Sort

O algoritmo do *bucket sort* recebe um vetor de entrada, cria alguns baldes e mapeia os valores do vetor para estes baldes. Em seguida, cada balde é ordenado com algum algoritmo de ordenação (insertion sort, bubble sort, merge sort, ...) e ao final desta ordenação, os valores são devolvidos para o vetor original pela ordem dos baldes. O vetor final estará com os elementos ordenados.

2. Implementação sequencial

Inicialmente, o algoritmo `bucket sort` foi implementado na sua versão sequencial convencional, utilizando o `insertion sort` como algoritmo para ordenação dos *buckets*. O programa cria três vetores alocados dinamicamente e preenche-os com valores randômicos utilizando a função `rand()`, e isso é feito na função `void fill(int *v, int size)`. Esses vetores tem tamanho `n = 1000, 10000 e 20000`. O algoritmo é executado uma vez para cada vetor desses. Na função `void bucketSort(int *v, int size)`, os valores são mapeados para os baldes utilizando a operação `mod`. Uma vez que os valores estão mapeados para os baldes, estes são ordenados com a função `void sort(int *v, int size)`, que foi implementada com o algoritmo `insertion sort`. A seguir, a implementação sequencial do algoritmo.

```
1  /*
2   * Fills the vector according to the size
3   */
4  void fill(int *v, int size) {
5    int i;
6    for(i = 0; i < size; ++i)
7      v[i] = rand()%RAND_MAX;
8  }
9
10 /*
11  * Bucket sort algorithm
12  */
13 void bucketSort(int *v, int size) {
14   Bucket_t *b = calloc(sizeof(Bucket_t), size); //creates a vector of buckets, initializing with 0
15
16   int i, j, idx;
17
18   for(i = 0; i < size; ++i) {
```

```
19      idx = v[i] % size; //calculates the index of the value in the new vector
20      (b[idx].num) += 1;
21      b[idx].bucket[(b[idx].num)-1] = v[i];
22    }
23
24    for(i = 0; i < size; ++i) { //sort each bucket
25      if(b[i].num != 0)
26        sort(b[i].bucket, b[i].num);
27    }
28
29    idx = 0;
30    for(i = 0; i < size; ++i) //put all the sorted buckets in the original vector
31      for(j = 0; j < b[i].num; ++j)
32        v[idx++] = b[i].bucket[j];
33    free(b);
34  }
35
36  /*
37   * Sort algorithm applied on each bucket (insertion sort)
38   */
39  void sort(int *v, int size) {
40    int i, j, tmp;
41
42    for(i = 1; i < size; ++i) { //i starts in 1
43      tmp = v[i]; //tmp is the current value
44      for(j = i - 1; (j >= 0) && (tmp < v[j]); --j) { //j starts one before i, j is between 0 and current
45        v[j+1] = v[j]; //change places of the elements
46      }
47      v[j+1] = tmp;
48    }
49  }
```

Listing 1. Implementação do bucket sort sequencial em C.

3. Implementação Paralela

Na implementação paralela, o numero de baldes criado para cada execução é igual ao numero de *threads* usada em cada execução, que são 1, 2, 4 e 8. A capacidade de cada *bucket* criado é `(n/n_threads) * 2`, sendo `n` o tamanho do vetor de entrada e `n_threads` o número de *threads* da execução.

3.1. OpenMP

Na implementação com OpenMP, dois laços foram paralelizados, ambos dentro da função `void bucketSort(int *v, int size)`. No primeiro laço foi usada a cláusula `#pragma omp parallel for schedule(guided)` para o mapeamento dos valores em cada `bucket`. No segundo laço, foi utilizado `#pragma omp parallel for schedule(guided) num_threads(n_thread)` para fazer a ordenação dentro de cada bucket. Foi decidido pelo escalonamento `guided` pois os baldes vão ter quantidades diferentes de elementos mapeados.

```
1   #pragma omp parallel for schedule(guided)
2   for(i = 0; i < size; ++i) {
3     idx = v[i] % n_thread; //calculates the index of the value in the new vector
4     (b[idx].num) += 1;
5
6     b[idx].bucket[(b[idx].num)-1] = v[i];
7   }
8
9   #pragma omp parallel for schedule(guided) num_threads(n_thread)
10  for(i = 0; i < n_thread; ++i) { //sort each bucket
11    if(b[i].num != 0)
12      sort(b[i].bucket, b[i].num);
```

Listing 2. Implementação do bucket sort paralelo com OpenMP em C.

3.2. PThreads

Na implementação com pthreads, algumas modificações foram feitas. Foi criada uma `struct` do tipo `Args_t` que contém um vetor e seu tamanho, e essa estrutura é passada como argumento para a função de criação de threads.

```
1  typedef struct args {
2    int *arr;
3    int sz;
4  } Args_t;
```

Listing 3. Estrutura criada no código.

A função usada foi a `pthread_create()` que recebe alguns parâmetros. São eles: um ponteiro para um vetor de threads, um ponteiro de `pthread_attr` que contem atributos de criação da thread (nesta implementação foi passado `NULL` como argumento, significando que as threads serão criadas com os atributos padrão), uma função do tipo ponteiro de `void` que recebe um ponteiro de `void` como parâmetros e, por fim, recebe um argumento do tipo ponteiro de `void` para ser o argumento de início da rotina.

```
1  /*
2   * Bucket sort algorithm
3   */
4  void bucketSort(int *v, int size) {
5    Bucket_t *b = calloc(sizeof(Bucket_t), n_thread); //creates a vector of buckets, initializing with 0
6    assert(b);
7
8    pthread_t thr[n_thread];
9    Args_t *args = malloc(sizeof(Args_t));
10   assert(args);
11
12   args->arr = v;
13   args->sz = size;
14
15   int i, j, idx, res;
16   for(i = 0; i < size; ++i) {
17     idx = v[i] % n_thread; //calculates the index of the value in the new vector
18     (b[idx].num) += 1;
19
20     b[idx].bucket[(b[idx].num)-1] = v[i];
21   }
22
23   for(i = 0; i < n_thread; ++i) { //sort each bucket
24     if(b[i].num != 0)
25       res = pthread_create(&thr[i], NULL, sort_threads, (void *)args);
26   }
27
28   idx = 0;
29   for(i = 0; i < n_thread; ++i) //put all the sorted buckets in the original vector
30     for(j = 0; j < b[i].num; ++j)
31       v[idx++] = b[i].bucket[j];
32   free(b);
33   free(args);
34 }
```

Listing 4. Implementação do bucket sort com PThreads em C.

Além disto, a função passada por parâmetro para `pthread_create()` foi implementada. Sua funcionalidade é simples, esta função chama a função `sort()` pas-

sando os parâmetros (`int *v, int size`) por meio do seu próprio parâmetro recebido (`void *params`).

```
1  /*
2   * Sorting function for pthreads
3   */
4  void *sort_threads(void *params) {
5    sort(((Args_t *)params)->arr, ((Args_t *)params)->sz);
6  }
```

Listing 5. Função passada para pthread_create().

4. Tempos de execução

As versões do algoritmo foram compiladas com `gcc -o b bucket.c` e os tempos de execução foram medidos utilizando o comando `time`. Para o tempo, os códigos foram executados 5 vezes e foi feita uma média de tempo dessas execuções.

4.1. Tempos do sequencial

n	**1000**	**10000**	**20000**
tempo (s)	0m0.012s	0m0.238s	0m0.765s

4.2. Tempos com OpenMP

n	**1000**	**10000**	**20000**
1 Thread	0m0.008s	0m0.103s	0m0.403s
2 Threads	0m0.009s	0m0.037s	0m0.125s
4 Threads	0m0.005s	0m0.009s	0m0.019s
8 Threads	0m0.004s	0m0.007s	0m0.025s

4.3. Tempos com PThreads

n	**1000**	**10000**	**20000**
1 Thread	0m0.002s	0m0.003s	0m0.004s
2 Threads	0m0.003s	0m0.003s	0m0.005s
4 Threads	0m0.003s	0m0.004s	0m0.005s
8 Threads	0m0.003s	0m0.005s	0m0.005s