IPPD

Algoritimo Bucket Sort
&
Discussão sobre Algoritmo de Fibonacci utilizando OpenMP

Prof: GERSON GERALDO HOMRICH CAVALHEIRO
Aluno: Mateus C S Santos - 14101915

Implementação do Algoritmo Bucket Sort e Bubble Sort

O algoritmo foi desenvolvido em C, utilizando estruturas e listas encadeadas, optei pelo uso de listas encadeadas por razão de economia de memória, o problema também poderia ter sido resolvido reservando um espaço máximo de memória previamente para a quantidade de valores possíveis em cada bucket, porém haveria desperdício de memória e necessitaria um tratamento exclusivo no código para contar os espaços que não foram usados. Com uma lista encadeada, em cada bucket não há limites de quantos elementos ficaram disponíveis para uso e cada bucket terá um número N diferente alocado, dependendo exclusivamente do conjunto de entradas a ser ordenado.

O código automaticamente cria uma *lista e aloca a quantidade de elementos passada por parâmetro no momento da inicialização (segundo parâmetro a ser informado), este parâmetro indica a quantidade de itens na lista e também o valor máximo que pode ser gerado de forma aleatória nesta lista, se informado 1000, a lista possuirá 1000 elementos e cada elemento poderá assumir um valor entre 0 a 1000. Após a lista ser gerada os elementos da lista serão divididos de acordo com seu bucket correspondente, a quantidade de buckets é informada como primeiro parâmetro de inicialização do programa, cada bucket possui a head para uma lista encadeada podendo ter quantos elementos desejar.

Por fim o programa deve inicializar a fazer de benchmark, onde você deve indicar como terceiro parâmetro ao inicializar o programa, 0 indica não paralelismo, 1 indica paralelismo.

Ao inicializar a função bubbleSort um timer passa a contar para verificar o tempo de execução do programa. Alguns testes foram realizados e os resultados se mostraram de acordo com a expectativa.

O primeiro teste foi feito com 100 buckets para 1000 números, perceba que o impacto do paralelismo teve um impacto negativo na execução do algoritmo pois sua execução foi cerca de 3.3x mais demorada do que a execução sem paralelismo.

```
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 100 1000 0
BucketSize: 100
MaxItemSize: 1000
Nao Paralelo
Tempo de execucao: 0.161600 ms
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 100 1000 1
BucketSize: 100
MaxItemSize: 1000
Paralelo
Tempo de execucao: 0.549200 ms
```

Um novo teste foi feito, desta vez 1000 buckets para 10000 números, perceba que ainda a execução com paralelismo teve efeitos negativos.

```
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 1000 10000 0
BucketSize: 1000
MaxItemSize: 10000
Nao Paralelo
Tempo de execucao: 0.641200 ms
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 1000 10000 1
BucketSize: 1000
MaxItemSize: 10000
Paralelo
Tempo de execucao: 1.137500 ms
```

Um novo teste foi feito, desta vez 10000 buckets para 100000 números, perceba que pela primeira vez a execução do código com paralelismo trouxe melhores substanciais no tempo de execução

```
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 10000 100000 0
BucketSize: 10000
MaxItemSize: 100000
Nao Paralelo
Tempo de execucao: 22.783100 ms
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 10000 100000 1
BucketSize: 10000
MaxItemSize: 100000
Paralelo
Tempo de execucao: 7.653200 ms
```

Um novo teste foi feito, desta vez 1000 buckets para 1000000 números, perceba que novamente execução do código com paralelismo trouxe melhorias substanciais no tempo de execução.

```
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 1000 100000 1
BucketSize: 1000
MaxItemSize: 100000
Paralelo
Tempo de execucao: 43.908700 ms
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\sort.exe 1000 100000 0
BucketSize: 1000
MaxItemSize: 100000
Nao Paralelo
Tempo de execucao: 115.020800 ms
```

Após executar alguns testes podemos notar o padrão e o comportamento já esperado para o funcionamento do programa, as primeiras duas execuções do programa obtiveram resultados negativos quanto aplicado paralelismo ao bubbleSort, este resultado já esperado se dá ao fato que o paralelismo de tarefas possui um custo e nem sempre este custo compensa a sua implementação, neste caso para lidar com listas pequenas o custo

de implementar um paralelismo é negativo e não possui qualquer benefício, a implementação de um execuções em paralelo deve ser estudada antes de aplicadas aos problemas, para de fato verificar se é necessário o esforço de implementação. Para os dois últimos testes, podemos começar a perceber o custo da implementação do paralelismo a se pagar para grandes quantidades de dados, neste caso dividir as tarefas do bubbleSort começaram a se pagar a partir de mais de 100.000 valores na lista.

Por fim, os resultados dos testes estão de acordo com o esperado e se mostram de acordo com o que já estudamos em aula.

A escolha do algoritmo bubbleSort foi feita por ser um algoritmo fácil de se implementar em uma lista encadeada, entretanto outro algoritmo sort pode ser facilmente implementado no futuro, pois o código foi devidamente modularizado.

Link para o GitHub

https://github.com/thehatb0y/bucketSort/tree/main

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <windows.h>

struct List {
    int value;
    struct List * next;
};

struct Bucket {
    int size;
    struct List * head;
};

void list Gen(int *list, int size) {
    int i;
    for(i = 0; i < size; i++) {
        list[i] = rand() % size;
    }
};

void printList(int *list, int size) {
    int i;
    for(i = 0; i < size; i++) {
        printf("%d ", list[i]);
    }
    printf("\n");
};
```

```
struct Bucket * createBucket(struct Bucket *bucketList, int BucketSize) {
    int i;
    bucketList = (struct Bucket *) malloc((BucketSize) * sizeof(struct Bucket));

    for (i = 0; i < (BucketSize); i++) {
        bucketList[i].size = 0;
    }

return bucketList;
```

```
};
```

```
int getItemGroup(int number, int BucketSize, int MaxItemSize) {
    int groupSize = MaxItemSize / BucketSize;
    int grupo = (number - 1) / groupSize + 1;

    if (grupo >= 0 && grupo <= BucketSize) {
        return grupo-1;
    } else {
        return -1; // Ou um valor que indique um erro, se necessário
    }
}
```

```
void printBucket(struct Bucket *bucketList, int BucketSize) {
    int i;
    struct List * aux;
    for (i = 0; i < BucketSize; i++) {
        printf("Bucket %d: ", i);
        aux = bucketList[i].head;
        while (aux != NULL) {
            printf("%d ", aux->value);
            aux = aux->next;
        }
        printf("\n");
    }
}
```

```
void addItem(struct Bucket *bucketList, int number, int BucketSize, int MaxItemSize) {
    int grupo = getItemGroup(number, BucketSize, MaxItemSize);
    struct List * aux;

    if (grupo != -1) {
        if (bucketList[grupo].size == 0) {
            bucketList[grupo].head = (struct List *) malloc(sizeof(struct List));
            bucketList[grupo].head->value = number;
            bucketList[grupo].head->next = NULL;
            bucketList[grupo].size++;
        } else {
            aux = bucketList[grupo].head;
            while (aux->next != NULL) {
                aux = aux->next;
            }
            aux->next = (struct List *) malloc(sizeof(struct List));
            aux->next->value = number;
            aux->next->next = NULL;
            bucketList[grupo].size++;
        }

    }

}
```

```c
void fillBucket(struct Bucket *bucketList, int *ItemList, int BucketSize, int MaxItemSize)
{
    int i;
    for (i = 0; i < MaxItemSize; i++) {
        addItem(bucketList, ItemList[i], BucketSize, MaxItemSize);
    }
}
```

```c
int *joinBuckets(struct Bucket *bucketList, int BucketSize, int MaxItemSize, int *auxList)
{
    int i;
    int j;
    int k = 0;
    auxList = (int *) malloc(MaxItemSize * sizeof(int));
    for (i = 0; i < BucketSize; i++) {
        struct List * aux;
        aux = bucketList[i].head;
        for (j = 0; j < bucketList[i].size; j++) {
            auxList[k] = aux->value;
            aux = aux->next;
            k++;
        }
    }
    return auxList;
}
```

```c
void pbubbleSort(struct Bucket *bucketList, int BucketSize) {
    int i, j;
    struct List *auxList;
    struct List *auxList2;

    #pragma omp parallel for private(auxList, auxList2, j) shared(bucketList, BucketSize)
    for (i = 0; i < BucketSize; i++) {
        auxList = bucketList[i].head;
        for (j = 0; j < bucketList[i].size; j++) {
            auxList2 = auxList->next;
            while (auxList2 != NULL) {
                if (auxList->value > auxList2->value) {
                    // Troca os valores
                    int temp = auxList->value;
                    auxList->value = auxList2->value;
                    auxList2->value = temp;
                }
                auxList2 = auxList2->next;
            }
            auxList = auxList->next;
        }
    }

}
```

```c
void nbubbleSort(struct Bucket *bucketList, int BucketSize) {
    int i, j;
    struct List *auxList;
    struct List *auxList2;

    for (i = 0; i < BucketSize; i++) {
        auxList = bucketList[i].head;
        for (j = 0; j < bucketList[i].size; j++) {
            auxList2 = auxList->next;
            while (auxList2 != NULL) {
                if (auxList->value > auxList2->value) {
                    // Troca os valores
                    int temp = auxList->value;
                    auxList->value = auxList2->value;
                    auxList2->value = temp;
                }
                auxList2 = auxList2->next;
            }
            auxList = auxList->next;
        }
    }
}
```

```c
void freeBucket(struct Bucket *bucketList, int BucketSize) {
    int i;
    struct List *auxList;
    struct List *auxList2;

    for (i = 0; i < BucketSize; i++) {
        auxList = bucketList[i].head;
        while (auxList != NULL) {
            auxList2 = auxList->next;
            free(auxList);
            auxList = auxList2;
        }
    }
    free(bucketList);
}
```

```
int main(int argc, char **argv) {

    int *ItemList;
    int BucketSize = atoi(argv[1]);
    int MaxItemSize = atoi(argv[2]);
    int p = atoi(argv[3]);

    struct Bucket * bucketList = createBucket(bucketList, BucketSize);

    int i;
    int j;

    ItemList = (int *) malloc(MaxItemSize * sizeof(int));
    listGen(ItemList, MaxItemSize);
    //printList(ItemList, MaxItemSize);
    printf("BucketSize: %d\n", BucketSize);
    printf("MaxItemSize: %d\n", MaxItemSize);

    fillBucket(bucketList, ItemList, BucketSize, MaxItemSize);
    //printBucket(bucketList, BucketSize);

    LARGE_INTEGER start, end, frequency;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);

    if (p == 1){
        printf("Paralelo\n");
        pbubbleSort(bucketList, BucketSize);
    }
    else{
        printf("Nao Paralelo\n");
        nbubbleSort(bucketList, BucketSize);
    }

    QueryPerformanceCounter(&end);
    printf("Tempo de execucao: %lf ms\n", (double) (end.QuadPart - start.QuadPart) /
frequency.QuadPart * 1000);

    free(ItemList);
    freeBucket(bucketList, BucketSize);

return 0;
}
```

Discussão Fibonacci

A execução do código Fibo-1 em relação ao Fibo-2 foi um tanto intrigrante num primeiro momento, esperava que código do primeiro programa fosse substancialmente mais rápido do que o segundo código, entretanto não foi exatamente isto que ocorreu, após rodar o código por algumas vezes obtivemos os seguintes resultados:

```
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\fibo-1.exe 35
- 9227465 -
Tempo: 14.39
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\fibo-2.exe 35
- 9227465 -
Tempo: 7.16
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\fibo-1.exe 40
- 102334155 -
Tempo: 154.86
PS C:\Users\mtsc\OneDrive\Área de Trabalho\IPPD> .\fibo-2.exe 40
- 102334155 -
Tempo: 83.71
```

Perceba que nos testes realizados, o primeiro programa possui quase o dobro de tempo de execução. Após pensar bastante, ainda não consigo pensar de forma clara e objetiva o que causa o primeiro programa ter o tempo dobrado de execução a primeira impressão que dá é que paralelizando r1 e r2 a performance deveria ser melhor.

Porém posso especular baseado em experiências anteriores, observando as diferenças entre Fibo-1 e Fibo-2, podemos perceber que r2 é executado de forma sequencial no programa Fibo-2, aproveitando do mesmo contexto para continuar sua execução, enquanto r2 em Fibo-1 vira uma tarefa, o que pode indicar que o sistema pode estar tendo um overload de tarefas e dependendo da complexidade da função pode não compensar a abertura de uma nova tarefa, mas sim aproveitar do contexto de uma tarefa já aberta para executar um próximo comando.

Logo concluo que a diferença de tempo entre Fibo-1 e Fibo-2 se dá ao fato de um overhead de tarefas, e dependendo do problema pode ser mais recompensador utilizar do contexto da tarefa ao invés de criar novas tarefas para funções não tão complexas.