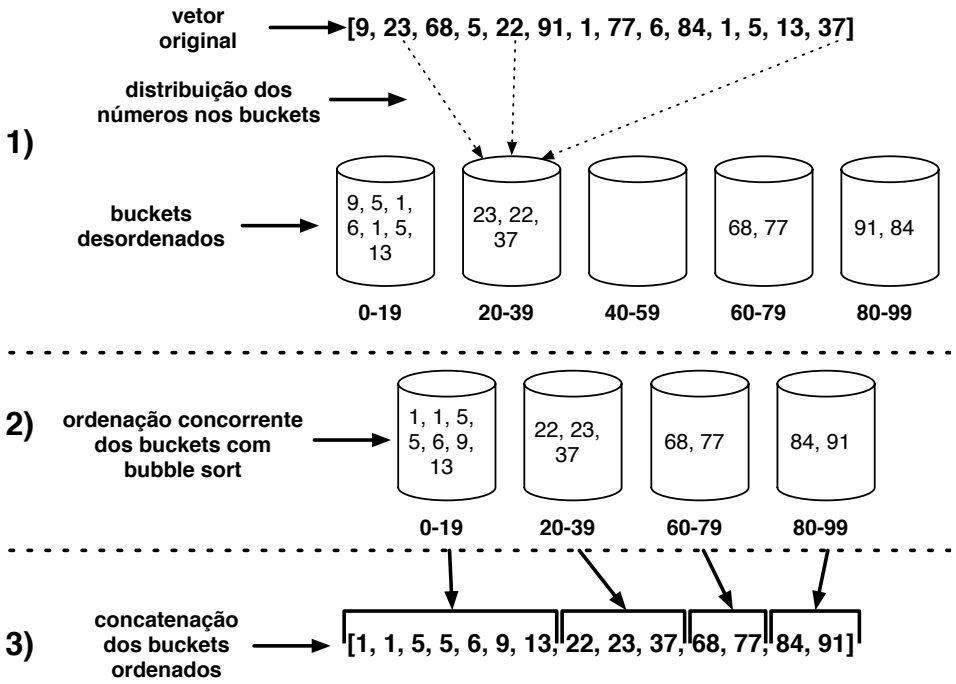
Trabalho 1 – Bucket Sort Concorrente

INE5410 - Programação Concorrente
Prof. Márcio Castro

2015/1

1 Definição

O *bucket sort* é um algoritmo de ordenação no qual o vetor a ser ordenado é dividido em um número finito de recipientes, denominados *buckets*. Cada *bucket* é então ordenado individualmente através de um algoritmo de ordenação (nesse trabalho utilizaremos o algoritmo *bubble sort*). A Figura 1 mostra o funcionamento do *bucket sort*, o qual é composto por 3 fases: 1) distribuição dos números do vetor desordenado nos *buckets*; 2) ordenação individual dos *buckets* utilizando o algoritmo *bubble sort*; e 3) redistribuição dos *buckets* ordenados no vetor original.



No exemplo mostrado acima considerou-se somente números inteiros positivos na faixa $[0; 99]$ os quais foram divididos em 5 *buckets*, cada um armazenando números em uma faixa contendo 20 valores possíveis ($[0, 19], [20, 39], ..., [80, 99]$). Note que um dos *buckets* ficou vazio, pois não havia nenhum número pertencente ao intervalo $[40, 59]$. Logo, esse *bucket* não deverá ser considerado durante os passos 2) e 3).

2 Trabalho

O primeiro trabalho da disciplina de Programação Concorrente consiste em implementar uma versão **concorrente** do *bucket sort* utilizando a linguagem C e a biblioteca POSIX Threads. Os parâmetros de entrada do programa serão:

- O tamanho do vetor (*tamvet*) a ser ordenado;
- O número de *buckets* (*nbuckets*);
- O número de *threads* (*nthreads*).

A faixa de valores possíveis no vetor de entrada será sempre $[0; tamvet - 1]$. Por exemplo, um vetor de entrada de tamanho 10 ($tamvet = 10$) poderá conter somente números dentro da faixa $[0, 9]$ enquanto um vetor de entrada de tamanho 30 ($tamvet = 30$) poderá conter somente números dentro da faixa $[0, 29]$.

As faixas de números permitidos em cada *bucket* deverão ser determinadas durante a execução do programa em função do tamanho do vetor de entrada (*tamvet*) e do número de *buckets* (*nbuckets*):

- **Sempre que possível**, as faixas de números permitidos em cada *bucket* deverão ter o **mesmo tamanho**. Por exemplo, para $tamvet = 50$ e $nbuckets = 5$ cada *bucket* deverá permitir números dentro de uma faixa de 10 números ($50/5 = 10$). Logo, os intervalos seriam $[0, 9]$, $[10, 19]$, $[20, 29]$, $[30, 39]$ e $[40, 49]$.
- **Nos casos onde a divisão não é inteira**, as faixas de números permitidos em cada *bucket* deverão ter **uma diferença de tamanho de no máximo 1 número**. Por exemplo, para $tamvet = 20$ e $nbuckets = 8$ teríamos $20/8 = 2$ com resto igual à 4. Nesse caso, 4 *buckets* teriam intervalos de 3 elementos e 4 *buckets* teriam intervalos de 2 elementos. Um exemplo de intervalos para esse caso seria: $[0; 2]$, $[3; 5]$, $[6; 8]$, $[9; 11]$, $[12; 13]$, $[14; 15]$, $[16; 17]$ e $[18; 19]$.

O *bucket sort* **concorrente** deverá funcionar da seguinte forma:

1. A *main thread* inicializa o vetor com valores aleatórios de acordo com o número de elementos especificado no parâmetro de entrada (*tamvet*). As funções `srand()` e `rand()` deverão ser utilizadas para gerar números aleatórios dentro do intervalo $[0; tamvet - 1]$. Pesquise sobre o funcionamento delas. O programa deverá gerar números diferentes em cada execução.
2. A *main thread* imprime o vetor desordenado.
3. A *main thread* distribui os elementos do vetor desordenado nos seus respectivos *buckets*, de acordo com o número de *buckets* especificado. Os *buckets* deverão ser identificados com valores no intervalo $[0, nbuckets - 1]$
4. A *main thread* cria *nthreads*. As *threads* deverão realizar os seguintes passos de maneira **concorrente**:
   4.1. Selecionar o próximo *bucket* desordenado de acordo com a ordem crescente dos identificadores dos *buckets*: $0, 1, 2, \ldots, nbuckets - 1$.
   4.2. Imprimir o identificador da *thread* e do *bucket* selecionado. As *threads* deverão ser identificadas por um número no intervalo $[0, nthreads - 1]$.
   4.3. Ordenar o *bucket* selecionado utilizando o algoritmo *bubble sort*.
   4.4. Caso ainda existam *buckets* desordenados, voltar para o passo 4.1.
5. A *main thread* redistribui os elementos ordenados dos *buckets* no vetor original, sobreescrevendo os valores antigos armazenados no vetor.
6. A *main thread* imprime o vetor ordenado.

O programa deverá funcionar em todos os casos, independentemente do tamanho do vetor, número de *buckets* ou *threads*, exceto nos seguintes casos: quando o número de *threads* for menor que 1 ou quando o número de *buckets* for maior que o tamanho do vetor. Nesses casos, o programa deverá informar um erro ao usuário.

Os parâmetros de entrada (*tamvet*, *nbuckets* e *nthreads*) deverão ser informados na linha de comando ou definidos dentro do código através do uso de constantes (ou seja, `#define`).

2.1 Saída

Ao final da execução, o seu programa deverá mostrar **obrigatoriamente** uma saída no seguinte formato: vetor desordenado, atribuição de *threads/buckets* (uma por linha, como indicado no passo 4.2 do pseudocódigo mostrado acima) e vetor ordenado. Um exemplo de saída válido é mostrado abaixo com $tamvet = 100$, $nbuckets = 20$ e $nthreads = 4$:

```
10 48 39 84 3 45 20 48 47 73 4 19 31 45 30 71 10 72 84 10 64 22 20 11 63 17 11 84 73 2 21 8 24 50 54 29 87 53 33 86
68 88 1 26 38 15 45 96 30 13 17 62 60 42 76 75 92 74 73 11 86 79 92 75 18 0 52 46 34 88 58 50 95 7 29 36 77 82 57 71
23 55 12 14 21 0 50 78 55 45 83 51 95 62 50 71 11 56 37 46
Thread 0 processando bucket 0
Thread 1 processando bucket 1
```

```
Thread 0 processando bucket 2
Thread 2 processando bucket 3
Thread 3 processando bucket 4
Thread 1 processando bucket 5
Thread 0 processando bucket 6
Thread 2 processando bucket 7
Thread 3 processando bucket 8
Thread 1 processando bucket 9
Thread 0 processando bucket 10
Thread 2 processando bucket 11
Thread 3 processando bucket 12
Thread 1 processando bucket 13
Thread 0 processando bucket 14
Thread 2 processando bucket 15
Thread 3 processando bucket 16
Thread 1 processando bucket 17
Thread 0 processando bucket 18
Thread 2 processando bucket 19
0 0 1 2 3 4 7 8 10 10 10 11 11 11 11 12 13 14 15 17 17 18 19 20 20 21 21 22 23 24 26 29 29 30 30 31 33 34 36 37 38 39
42 45 45 45 45 46 46 47 48 48 50 50 50 50 51 52 53 54 55 55 56 57 58 60 62 62 63 64 68 71 71 71 72 73 73 73 74 75 75
76 77 78 79 82 83 84 84 84 86 86 87 88 88 92 92 95 95 96
```

3 Grupos, Avaliação e Entrega

O trabalho deverá ser realizado **necessariamente** em grupos de **2 alunos**. Os alunos deverão apresentar o trabalho ao professor assim como mostrar sua solução em funcionamento. As apresentações serão feitas durante as aulas nos seguintes dias:

- **23/04/2015:** Grupos A ao G (demais grupos estão liberados desta aula e terão presença confirmada).
- **28/04/2015:** Grupos H ao O (demais grupos estão liberados desta aula e terão presença confirmada).

Pelo menos um dos integrantes de cada grupo deverá enviar através do Moodle um arquivo contendo o código fonte em C da solução para o trabalho. A data/hora limite para o envio dos trabalhos é **22/04/2015 às 23h55min**. **Não será permitida a entrega de trabalhos fora desse prazo.**

O professor irá avaliar não somente a corretude mas também o desempenho e a clareza da solução. Além disso, os alunos serão avaliados pela apresentação e entendimento do trabalho. **A implementação e apresentação valerão 40% e 60% da nota do trabalho, respectivamente**.

4 Bubble Sort

Os grupos poderão utilizar a implementação do algoritmo *bubble sort* mostrada a seguir para ordenar os elementos dentro dos *buckets*. Os parâmetros `v` e `tam` correspondem ao vetor a ser ordenado e o seu tamanho, respectivamente.

```
1  void bubble_sort(int *v, int tam){
2      int i, j, temp, trocou;
3      for(j = 0; j < tam - 1; j++){
4          trocou = 0;
5          for(i = 0; i < tam - 1; i++){
6              if(v[i + 1] < v[i]){
7                  temp = v[i];
8                  v[i] = v[i + 1];
9                  v[i + 1] = temp;
10                 trocou = 1;
11             }
12         }
13         if(!trocou) break;
14     }
15 }
```