Corrida por eliminação

EP2 de MAC0422

Emanuel Lima e João Seckler

9009493 4603521

Detalhes de implementação

Detalhes de implementação

A cada ciclista corresponde uma `struct corredor`. O vetor compartilhado `pista` é uma matriz `d` por `10` de ponteiros para `struct corredor` (guardando `NULL` numa posição se nenhum corredor estiver aí presente). A `pista` corresponde uma matriz de mutex, de mesmas dimensões, para controlar o acesso à posições.

Detalhes de implementação

- Opção debug: passe `debug` como terceiro argumento. A cada 60ms (ou 20ms, se alguém estiver correndo a 90km/h) o programa imprime a pista. O sentido da pista é da esquerda para a direita. Os ciclistas são identificados por números. Um lugar da pista vazio é identificado por um underscore (_).

Detalhes de implementação

Assim como na corrida real, os corredores começam antes da linha de chegada. O nosso algoritmo conta um "turno fantasma", onde uma variável `turn` é igual a `-1`, que corresponde a esse momento entre a partida e a primeira vez que o corredor passa pela linha de chegada.

Detalhes de implementação

A posição x de cada corredor (deslocamento frente/trás) é guardada como o dobro do que de fato é. Para ler/escrever, dividimos inteiramente por 2. Quando houver alguém correndo a 90, essa constante passa a ser 6.

Detalhes de implementação

- Quando um ciclista correr rodadas suficientes para determinar sua posição em todas as rodadas da corrida, a thread correspondente é desalocada e tudo se passa como se esse corredor fora eliminado, exceto que a sua posição em cada volta foi armazenada e poderemos anunciar sua colocação.

- Em toda rodada, cada ciclista tenta andar quantas posições for possível à sua esquerda.

Detalhes de implementação

- **Variáveis relevantes**

```
struct pr {
    int id;
    enum velocity vel; // velocidade atual
    int x; // deslocamento frontal
    int y; // deslocamento lateral
    pthread_t thread;
    int state; // máscara, 1º bit: correndo? 2º bit: thread desalocada?
    int * placing; // vetor de colocações por rodada
    int final_turn; // última rodada corrida
    float final_time; // instante que passou pela linha de chegada pela
última vez
};
```

Detalhes de implementação

- **Variáveis globais**

```
int corredor_count; // número de corredores na pista
pthread_mutex_t corredor_count_mutex;//

pthread_barrier_t *barrier; // vetor de duas barreiras, uma para rodadas pares e outra
                            // para rodadas ímpares
int * placing_count; // para cada índice i, armazena quantos corredores já
                     // completarm a i-ésima volta
int * placing_count_max; // para cada índice i, guarda a pior colocação da i-ésima rodada
pthread_mutex_t * placing_count_mutex;

int global_turn; // em que turno está o pior colocado
pthread_mutex_t global_turn_mutex;

float elapsed_time; // tempo decorrido desde o início da corrida

int final_run; // tem alguém correndo a 90km/h?
pthread_mutex_t final_run_mutex;
```

Resultados de experimentos

Condições

Computador:

- A: Intel(R) Core(™) i7-4500, **4 núcleo físicos**

Entradas usadas:

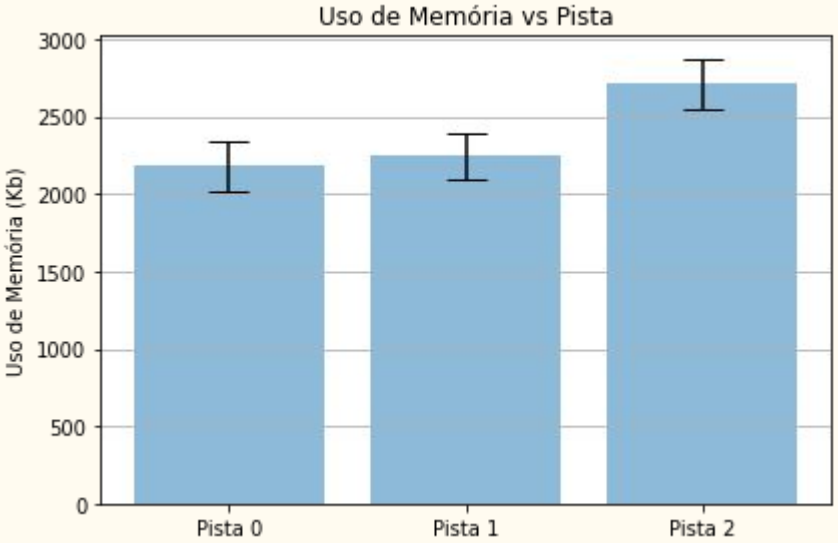
Tamanho da pista 0: 10	Nº ciclistas 0: 5
Tamanho da pista 1: 100	Nº ciclistas 1: 25
Tamanho da pista 2: 1000	Nº ciclistas 2: 250

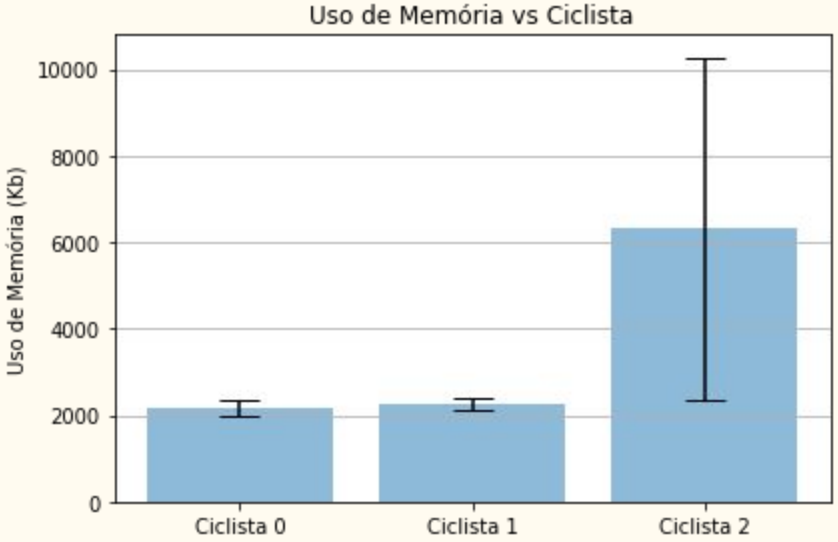
Para testar os impactos do tamanho da pista, usamos 25 ciclistas. Para testar os impactos do número de threads, usamos pista de tamanho 100

Ferramenta usada para medição:

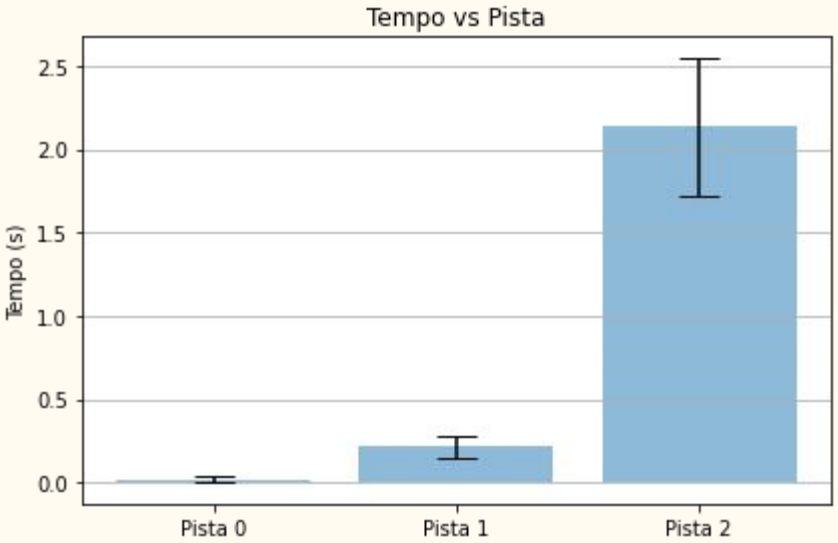
comando `time` no argumento `-v` informa o tempo de relógio (“User time”) e o máximo de memória usada pelo programa (“maximum resident set size”)

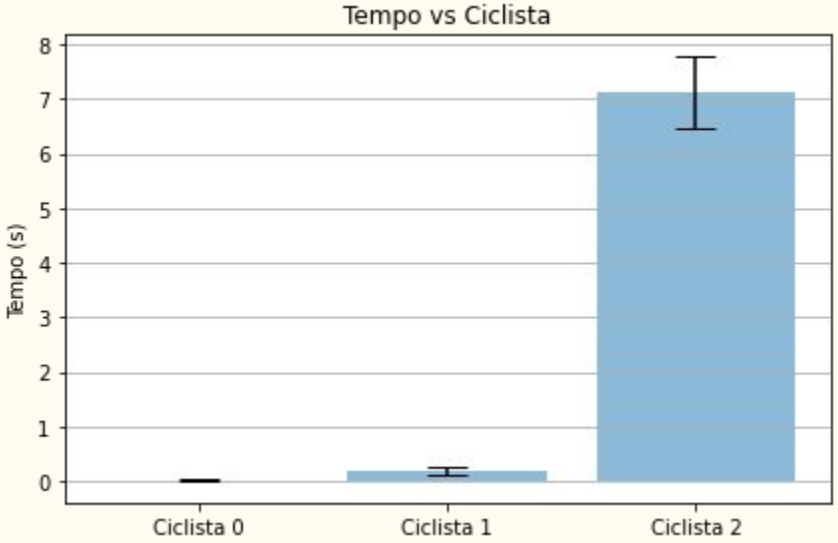
Gráficos - Uso de Memória





Gráficos - Tempo





Análise

Um resultado esperado e bastante simples é que, quanto mais threads e quanto maior a pista, mais tempo e memória se gasta, o que é natural.

Os gráficos que analisam o tempo revelam um crescimento linear do tempo de execução, tanto para o tamanho da pista quanto para o número de threads. Assim como os dados de entrada, as três barras parecem estar num crescimento exponencial.

Análise

Os gráficos que analisam a memória revelam um fato interessante sobre o código: a maior parte da alocação de memória é proporcional ao número de threads. Isso faz sentido, uma vez que usamos muito mais memória para criar e armazenar informações sobre as threads do que sobre a pista.

Aliás, no nosso código, a memória relativa à pista é proporcional a `d`, enquanto a memória relativa às threads é proporcional a n^2. Podemos interpretar o gráfico "uso da memória vs pista", à luz disso, entendendo que há um overhead de memória devido ao número de threads e que a memória relativa à pista pouco muda varia acima disso.