EP2 - MAC0422

Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação
Alunos:
Luciano Rodrigues Saraiva Leão(11221817)
Davi de Menezes Pereira(11221988)

Decisões gerais

- As thread funcionam como Workers e a main funciona como Coordenador. Inicialmente pensamos em usar pthread_barrier para sincronizar a execução das threads em cada instante de tempo, mas depois achamos melhor sincronizarmos usando pthread_cond, pois estávamos com problemas para trocar a variável da barreira e para garantir que o coordenador fosse o último a executar.
- As decisões de ultrapassagem foram deixadas a critério do escalonador.
- Deixamos uma seed fixa para o rand (8080).
- Na opção de debug, em vez de colocar um argumento específico, decidimos apenas colocar que se o número de argumentos na linha de comando for maior que três, ativamos o modo de debug. O debug imprime na saída de erro padrão. (e a saída “normal” é imprimida na stdout).

Decisões gerais - Main

Ideia da estrutura geral da main:

- inicia
- enquanto ainda tem gente que não acabou fica num loop
  - espera todo mundo processar
  - debuga (opcional)
  - imprime volta(s) finalizada(s) no momento
  - ajusta o intervalo e acrescenta o tempo
- imprime as colocações finais
- finaliza as estruturas

Decisões gerais - Threads

Ideia da estrutura geral da thread:

- inicia
- enquanto perdeu ou não quebrou
  - se já foi processado no momento espera os outros serem processados
  - se ainda não avançou o suficiente para trocar de posição, atualiza e volta pro início do loop
  - senão checa as posições da frente para ver se pode avançar, se não puder volta para o início
  - senão avança
  - se mudou de volta checa se quebrou, foi eliminado e outras coisas.
- atualiza algumas coisas
- finaliza

Decisões gerais - Semáforos e estruturas

Usamos alguns mutex para proteger as seções críticas, como mutex_n para proteger as áreas que manipulam o n_atual, lock e lock main para proteger as áreas em onde fazemos a sincronização, vetor mutex_volta para proteger uma volta e uma matriz de mutex para proteger cada posição da pista.

Além disso, utilizamos alguns _Atomic para controlar o acesso de algumas variáveis específicas que precisavam ser manipuladas como uma operação atômica e usamos pthread_cond para fazer as threads e a main pararem, ajudando a sincronizar como se fosse uma barreira.

Implementação do controle de acesso à pista

Como explicado no slide anterior, o acesso à pista está sendo controlado por uma matriz de mutex que controla cada posição da pista. Então quando vamos checar as posições da pista que estamos querendo manipular, protegemos todas elas cada uma com seu respectivo mutex.

Implementação da sincronização das threads

Queríamos fazer com que todas as threads executassem, e só depois a main executasse enquanto as threads esperassem. Para isso usamos pthread_cond, similarmente ao que fizemos no EP1.

Para a main:

```
pthread_mutex_lock(&lock_main);
  while(n_atual && processados < n_atual && terminados < n)
  {
      pthread_cond_wait(&cond_main, &lock_main);
  }
  processados = 0;
  pthread_mutex_unlock(&lock_main);
```

Implementação da sincronização das threads

Para as threads:

```
pthread_mutex_lock(&lock);
while(self->tempo > tempo)
      pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

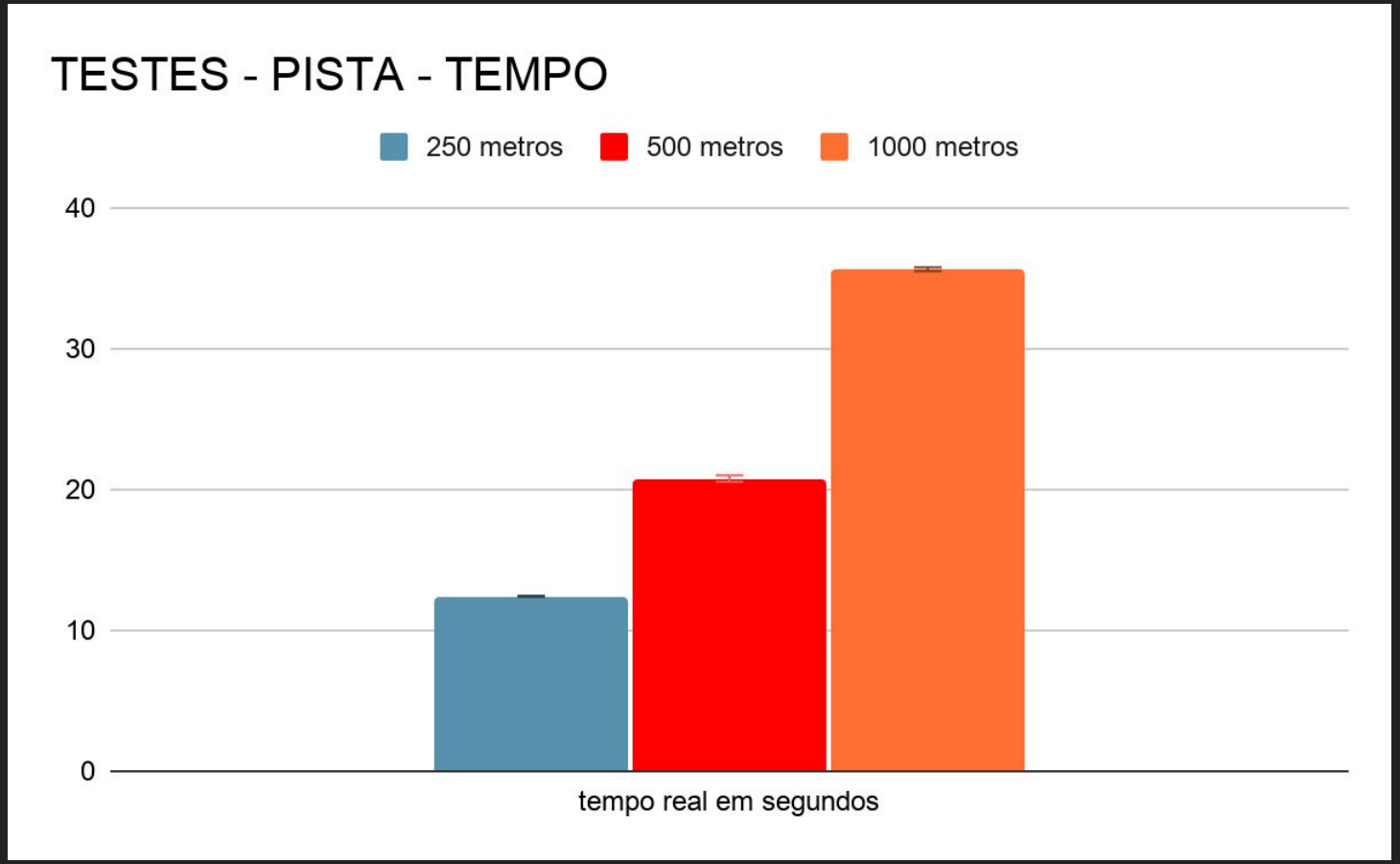
enquanto ja foi processado naquele momento, espera sinal da main para continuar. (a main manda o sinal com um boradcast, para todas as threads)

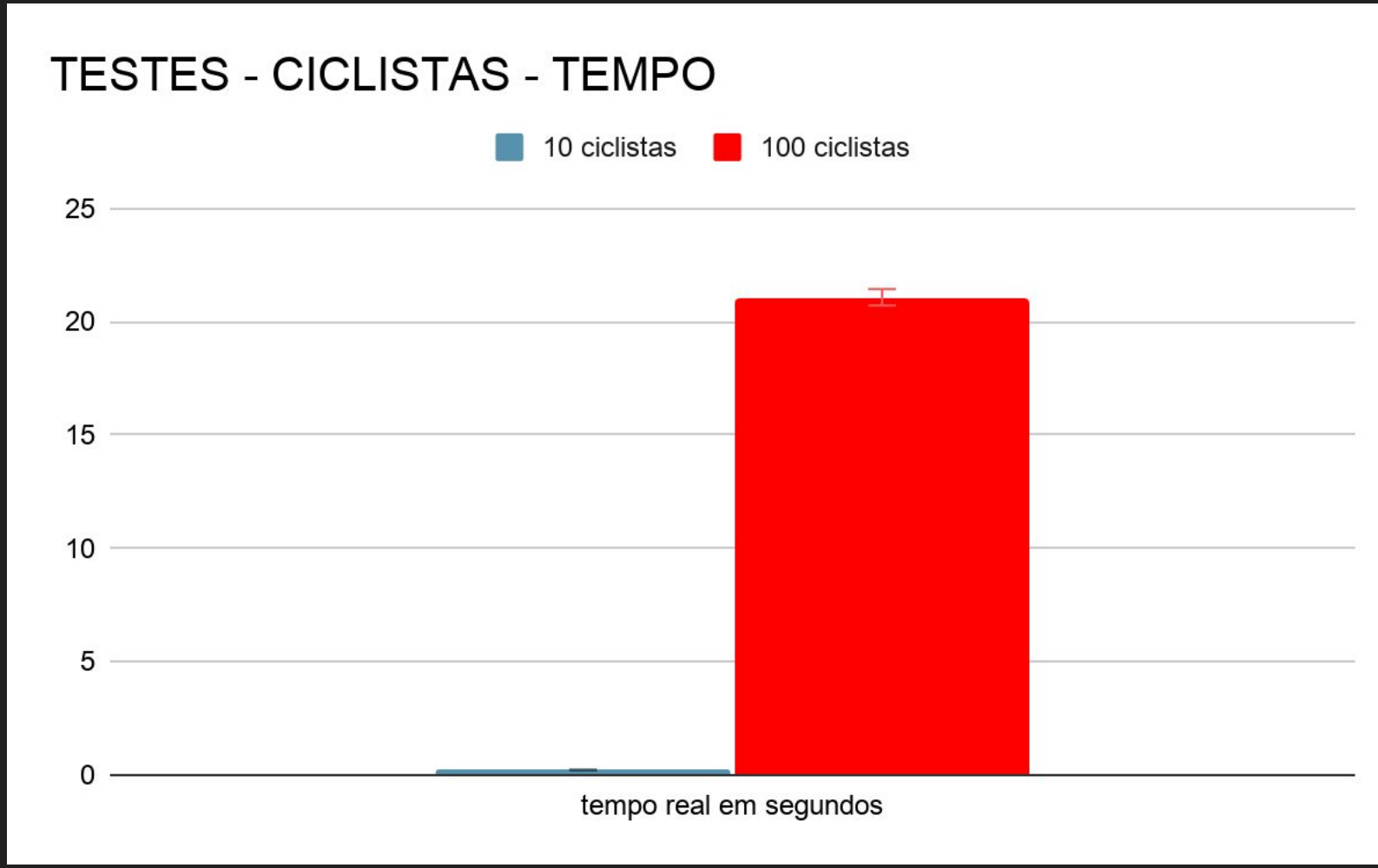
TESTES

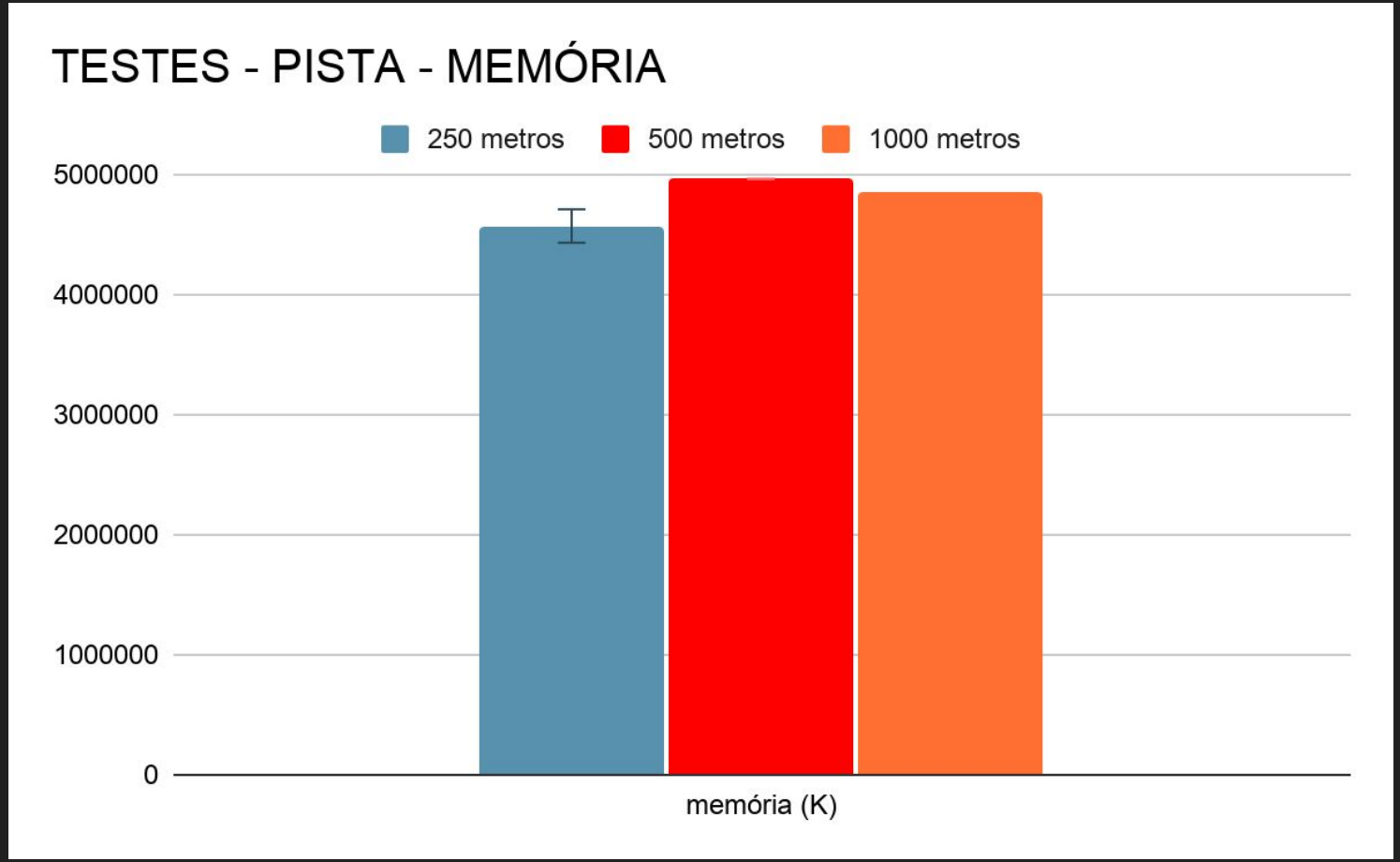
Os testes foram feitos separadamente variando o tamanho da pista e depois variando o número de ciclistas. Os valores para os testes das pistas foram 250, 500 e 1000 para 100 ciclistas e os valores para os testes dos ciclistas foram 10, 100 e 1000 para uma pista de tamanho 500. Para cada um desses valores foi testado o uso de memória utilizando o comando pmap, e o consumo de tempo utilizando o comando /usr/bin/time com a opção -p, cada um rodando 30 testes, como pedido no enunciado.

TESTES

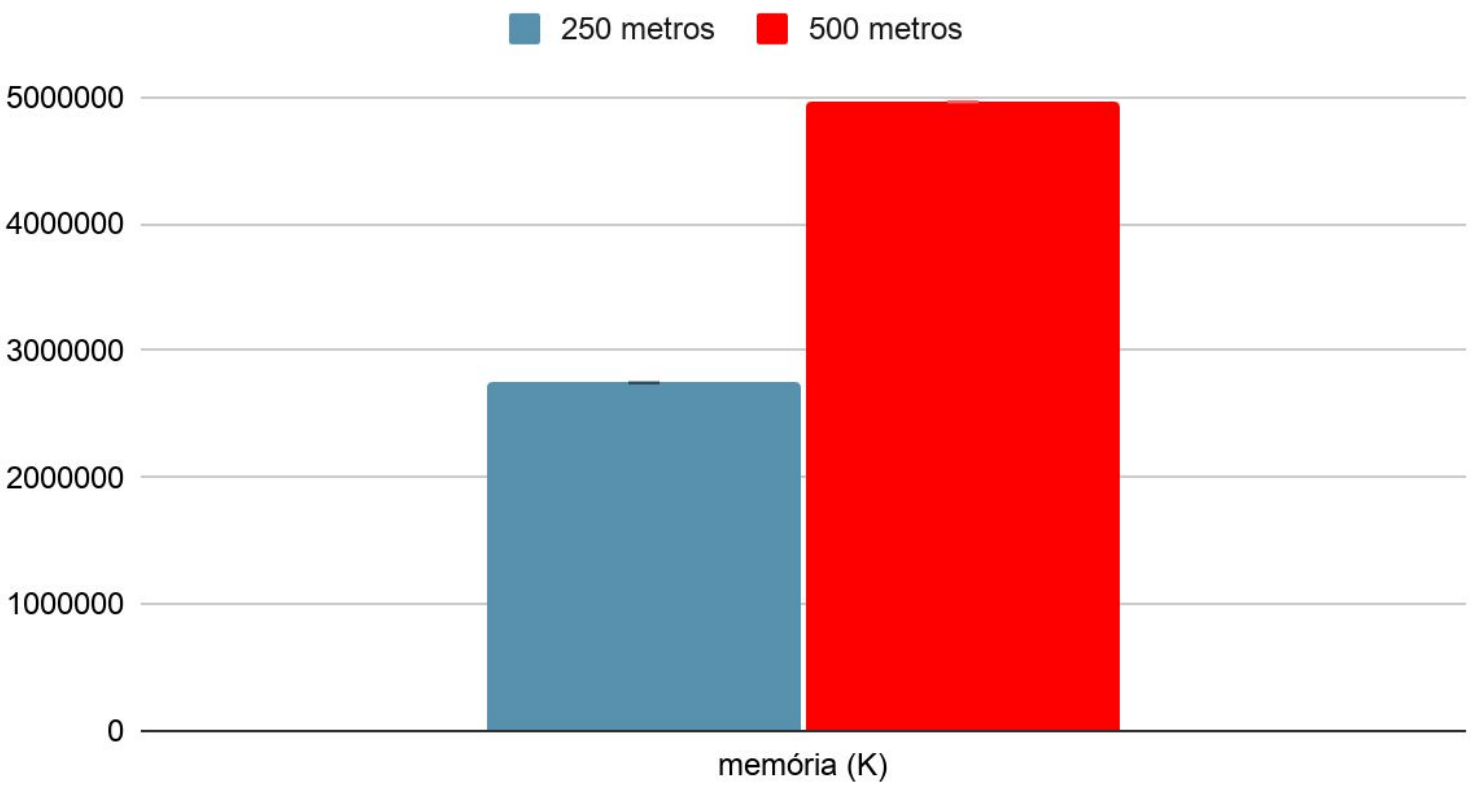
Fizemos um script em bash para rodar os testes e colocá-los numa saída, depois fizemos um programa em C para analisar esses dados, tirando o desvio padrão e nos dando um intervalo de confiança de 30 medições com nível de confiança de 95%.


TESTES - PISTA - TEMPO
250 metros
500 metros
1000 metros
40
30
20
10
0
tempo real em segundos


TESTES - CICLISTAS - TEMPO
10 ciclistas
100 ciclistas
25
20
15
10
5
0
tempo real em segundos


TESTES - PISTA - MEMÓRIA
250 metros
500 metros
1000 metros
5000000
4000000
3000000
2000000
1000000
0
memória (K)

TESTES - CICLISTAS - MEMÓRIA



CONCLUSÃO

Em relação ao tempo, podemos ver que o número de ciclistas tem um impacto maior, o que faz sentido, uma vez que a parte que vai demorar no nosso código é quando alguém acaba uma volta e temos que fazer todas as checagem na matriz e em outros vetores, o que toma tempo, além de essa parte só está sendo acessada por um ciclista por vez, pois está protegida por um mutex. Quanto mais ciclistas, mais voltas.

A memória também é mais impactada pela mudança na quantidade de ciclistas, pois para cada um deles armazenamos vários parâmetros e alocamos o id, por exemplo, quando vamos chamar a thread. Porém, a variação no uso da memória é mais leve em relação à mudança de tempo, pois usamos quase tudo como matrizes e vetores de tamanho fixo (MAXN).

OBSERVAÇÃO

Até o momento não conseguimos testar para o caso dos 1000 ciclistas 30 vezes, então estamos enviando essa versão inicial para caso não consigamos concluir os testes até o horário de entrega do EP.