Benchmarks dos protótipos

1) O código lê arquivos com registros de 100 bytes, sendo que cada registro possui 4 bytes para chaves e 96 bytes para dados. Mapea-os na memória, ordena utilizando Multithreaded, e por fim, escreve em um arquivo de saída
2) Será usado como base um script que executa o código 50 vezes por quantidade de Threads de maneira iterativa, para que seja possível obter uma média dos tempos e compará-los no final.
3) As chaves dos vetores foram sorteadas de maneira aleatória

Benchmarks do projeto

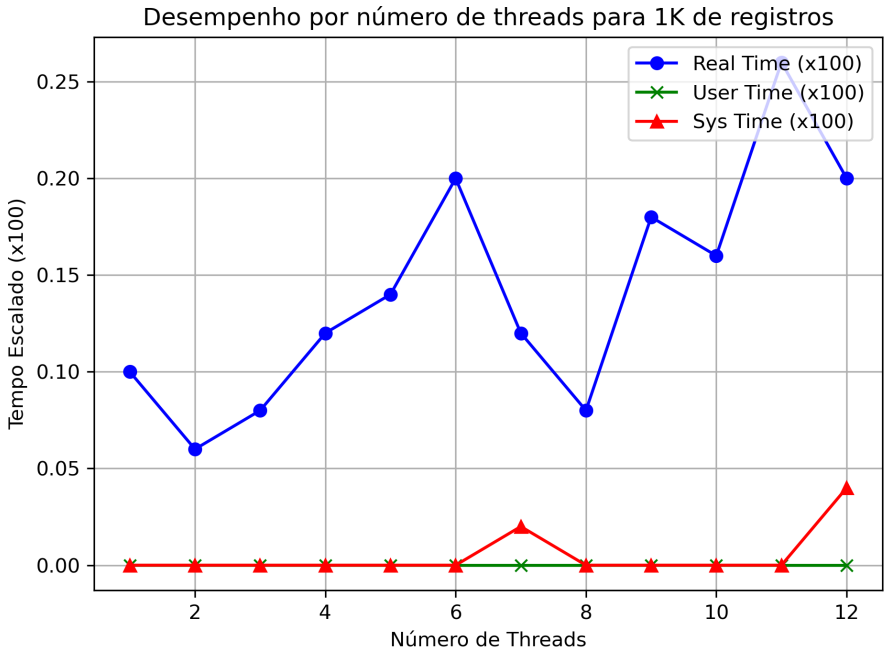

Figura 1: Gráfico do tempo médio para 1 mil registros

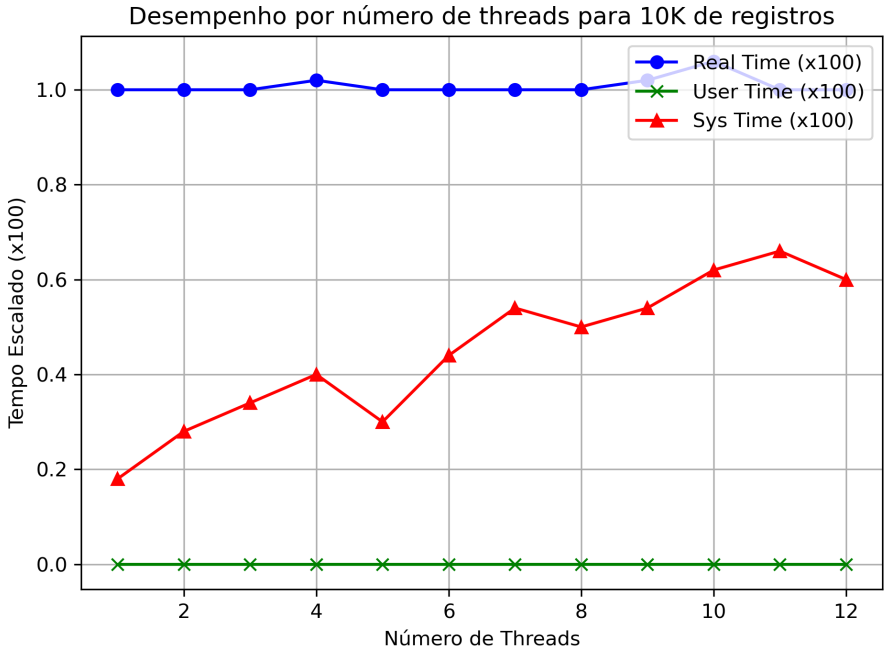

Figura 2: Gráfico do tempo médio para 10 mil registros

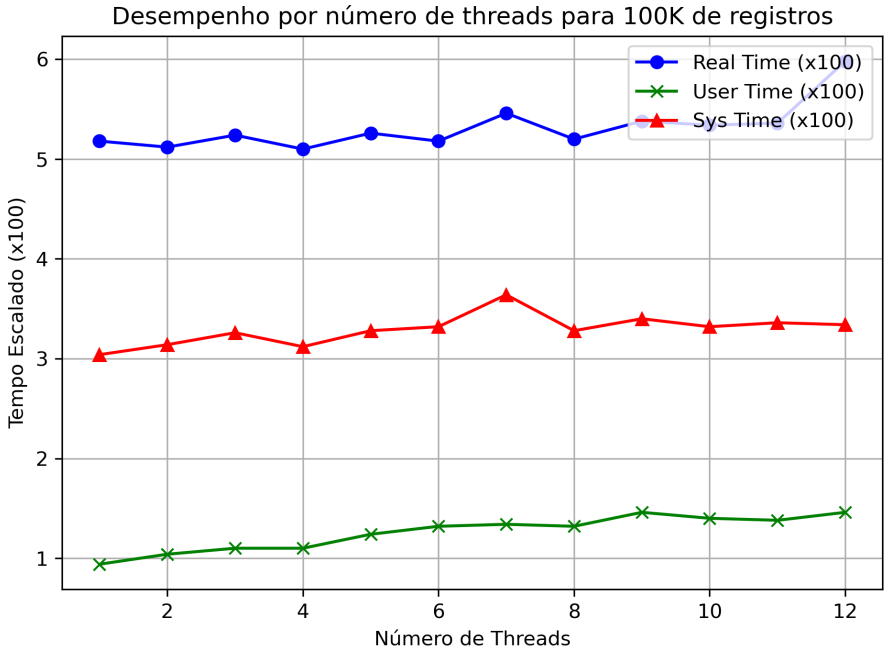

Figura 3: Gráfico do tempo médio para 100 mil registros

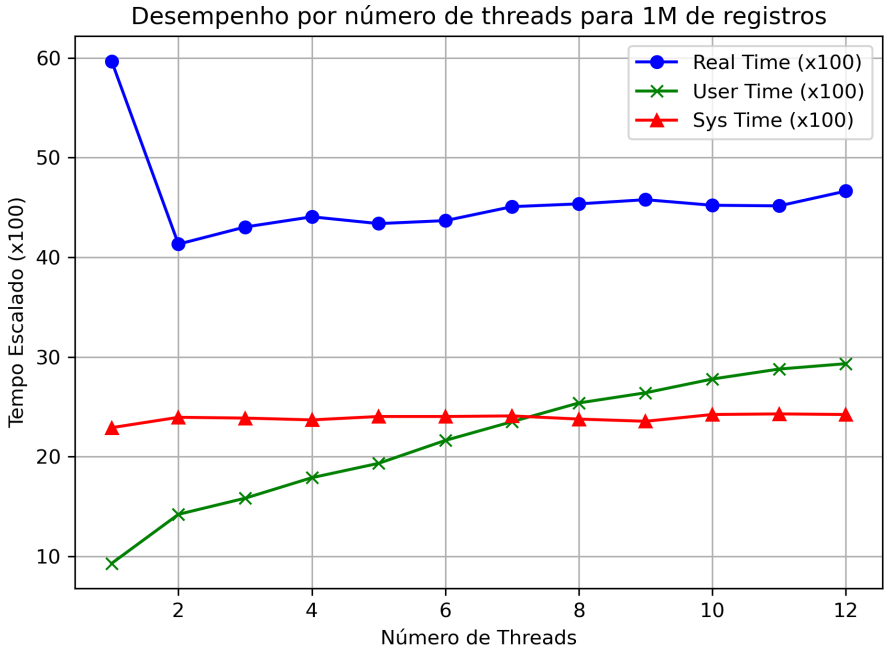

Figura 4: Gráfico do tempo médio para 1 milhão de registros

Decisões para algoritmo de ordenação Multi-Thread

Algoritmo escolhido: ~~MERGE SORT~~

- Como algoritmo escolhido para o desenvolvimento do projeto, seguiremos como base o *Merge Sort*, algoritmo de ordenação recursivo, que tem como motivação o princípio de divisão e conquista. Consiste em ir dividindo recursivamente o meu vetor pela metade, até chegar em um momento em que terei vários "vetores" de tamanho 1. Quando chegar nesse ponto, começo a combiná-los e ordená-los.
- Esse algortimo tem complexidade $nlog(n)$ de tempo e complexidade $nlog(n)$ de espaço, ou seja, ele é um algoritmo eficiente, mas não é um algoritmo *in loco*, ou seja, ele precisa de vetores temporários que auxiliam nas combinações.

Observações até o momento (23/10/2024)

- O protótipo criado para testar a implementação de um Merge Sort com uso de Threads, até o momento, está se mostrando pouco eficiente, visto que está sendo criada uma thread para cada sub-vetor até chegar em um ponto em que não temos mais threads disponívies, e continuamos com um *Merge Sort sequencial*, ou seja, sem o uso de paralelismo.
- O custo para as ações que envolvem threads, como criação e união compromete o desempenho total.

Observações até o momento (28/10/2024)

- O protótipo está evoluindo, deixando as Threads serem "independentes". Os problemas que ainda estão sendo investigados são os de situação de corrida, onde mais de uma Thread tenta acessar regiões críticas ao mesmo tempo, falha de segmentação, onde uma Thread tenta acessar endereços não mais disponíveis (sendo esse o problema mais crítico até então), entre outros problemas.
- Enquanto isso, os tempos dos algoritmos de ordenação sem usar Multithread eram mais rápidos do que usando Multithreaded, visto que o tempo de criar e gerenciar threads custa muito caro.

Observações até o momento (29/10/2024)

- Acabei consertando algumas coisas no código, pois o algoritmo com Threads não estava ordenando o vetor inteiro, e sim blocos do vetor, de modo que eu tinha trechos ordenados. Porém agora, foi adicionado uma chamada na função `merge()` ao final da função que inicia o MergeSort Multithreaded, mas não se sabe se é permitido ou não.

- Está sendo extremamente díficil otimizar o código, já que há concorrência entre as Threads se o tempo for prioridade, ou então um prejuizo no tempo final, se a confiabilidade for a prioridade

Mudança na escolha da abordagem

Algoritmos escolhidos: QUICK SORT, INSERTION SORT e trechos do MERGE SORT

- Após conversar com o professor responsável pela disciplina que propôs o Exercício de Programa (este projeto), a abordagem que o código usando Threads assumirá será:
  - Com base no número de Threads disponíveis, separar o vetor em sub-vetores, e que, de forma paralela, cada um irá executar um algoritmo de ordenação a depender do tamanho do sub-vetor (Caso tenha menos de 15 elementos, aplica Insertion Sort, caso contrário, aplica Quick Sort, disponível nos cabeçalhos padrões de C).
    - Caso a divisão dos sub-vetores venha a deixar elementos sobrando, iremos acrescentar o restante dos elementos na última Thread.
  - Após cada sub-vetor estar ordenado separadamente, será aplicado *Merge's* de forma iterativa, que terá o papel de mesclar cada sub-vetor, ordenando-os de par em par.
  - No fim, o vetor estará ordenado usando abordagens sequenciais e Multithreaded.

Observações até o momento (02/11/2024)

- A abordagem usando diversos algoritmos conhecidos de ordenação apresentou um tempo melhor comparado ao *Merge Sort* tradicional. Se por acaso fosse usado *Merge Sort's* sequencias em cada Thread, não conseguiriamos obter um ganho de desempenho, pelo contrário, teríamos um prejuízo por conta da criação e união das Threads (Não se sabe se essa "estratégia" para otimização é válida para o desafio proposto).
- Fazendo testes com vetores muito grandes e com número ímpar de elementos, aparentemente o tempo para ordenar ficou maior do que o esperado. Talvez possa ser por alguma limitação assintótica do *Quick Sort*, talvez as Threads não estejam conseguindo lidar tão bem com os seus sub-vetores, ou algo semelhante.
- Implementei a lógica para leitura dos arquivos em binário para um vetor de uma estrutura que guarda a chave (4 bytes) e os dados (96 bytes), usando a função `fread()`
- Executando alguns testes, resolvendo alguns problemas no código, ainda aparecem condições de corrida, em grande quantidade quando o programa executa com 6 Threads, mas caso seja solicitado 1 a 5 Threads, o programa não apresenta tantos resultados errados. Isso ainda deve ser investigado para ter certeza de que o código sempre funcionará, independente do tamanho do vetor e da quantidade de Threads.

Observações até o momento (05/11/2024)

- Como o código estava aparentando ter condições de corrida mesmo que as regiões críticas estivessem isoladas, foi acrescentado um `while()` que verifica se o vetor foi ordenado ou não, e caso ele não tenha sido, tenta ordenar novamente (uma abordagem custosa - mas talvez temporária)
- Modifiquei a lógica para obter os registros usando `mmap()`, diminuindo o número de *allocs*, diminuindo as chances de vazão de memória e diminuindo o tempo gastand alocando memória dinâmicamente. Também implementei a lógica para escrever os registros em um arquivo de saída (porém em binário)
- Acredita-se que o código possui complexidade *O(n + p)* de espaço, já que ainda utiliza da função `merge()` do *MergeSort* e também leva em conta as Threads criadas, e que possui uma complexidade de aproximadamente:

$$O(\frac{nlog(n)}{Threads} + Overhead)$$

  onde *nlog(n)* pode variar dependendo da qualidade dos vetores, mas que se resume ao tempo médio do *QuickSort*, *Threads* sendo o número de Threads disponíveis para serem paralelizadas e o *Overhead* sendo o tempo gasto na criação e união das mesmas.

Observações até o momento (07/11/2024)

- Foi feito uma mudança na função responsável por escrever no arquivo de saída, uma vez que da maneira que estava o programa demorava muito tempo para executar, sendo que na maior parte do tempo ele estava escrevendo no arquivo de saída

Observações até o momento (14/11/2024)

- Foi feito um teste implementado o algoritmo de ordenação *Radix Sort*, que é um algoritmo que não ordena por comparação de chaves e se aproveita do algoritmo *Counting Sort*, em que leva em consideração apenas o valor do i-ésimo byte da chave. Durante os testes, houve um rendimento satisfatório nos tempos de execução, porém o atual código que utiliza *Quick Sort* tem um rendimento melhor do que o esperado, já que as complexidades assintóticas são diferentes
- Foi feito mudanças no algortimo de ordenação *Radix Sort*. Ao invés de iterar por digitos decimais, itera por byte, reduzindo o número de iterações totais necessárias. Foram feitas também mudanças no código em geral, com **auxílio de terceiros**, usando estruturas auxiliares que movimentam ponteiros ao invés de movimentar os dados reais, bem como otimizações em manipulação de memória.