Aula prática 6 – Estrutura de Dados

Algoritmo de Ordenação - BubbleSort

Iago da Silva Rodrigues Alves - 2022035881

Tópico 1

Em relação aos acessos de memória esperados, podemos observar o seguinte:

1. Acesso à memória para o vetor:
    - O vetor é criado e posteriormente preenchido com valores usando um loop.
    - O acesso à memória ocorre durante a inicialização do vetor.
2. Acesso à memória durante a ordenação:
    - O algoritmo de ordenação BubbleSort realiza acessos repetidos ao vetor para comparar e trocar os elementos.
    - Os acessos de leitura ocorrem ao comparar os elementos adjacentes.
    - Os acessos de escrita ocorrem ao trocar os elementos quando necessário.

Em relação à localidade de referência, podemos esperar o seguinte comportamento:

1. Localidade espacial:
    - O algoritmo de ordenação BubbleSort faz acessos consecutivos ao vetor, percorrendo-o várias vezes.
    - Isso tende a ter uma boa localidade espacial, já que elementos adjacentes são acessados em sequência.
2. Localidade temporal:
    - O algoritmo faz múltiplas iterações sobre o vetor, o que pode levar a um bom aproveitamento do cache L1.
    - A troca de elementos adjacentes também pode beneficiar a localidade temporal, pois os elementos recentemente acessados são usados novamente.

Em relação às estruturas de dados e segmentos de código críticos:

1. Estruturas de dados:
    - A estrutura de dados crítica é o vetor que armazena os elementos a serem ordenados.
2. Segmentos de código críticos:
    - O segmento de código crítico é o laço aninhado dentro da função bubbleSort(), onde ocorre a comparação e a troca de elementos.
    - Esse segmento de código é responsável pelo desempenho geral do algoritmo e tem complexidade O(n^2), o que pode ser custoso para grandes conjuntos de dados.

Tópico 2

Após analisar o algoritmo a ideia foi executar o programa com diferentes tamanhos vetor que permitiria uma análise completa. Utilizando ferramentas como o valgrind é possível coletar informações sobre os acessos à memória durante a execução do programa, fornecendo métricas como os tipos de acessos (leitura/escrita), os endereços de memória acessados e as frequências de acesso.

Tendo acesso aos resultados, analisarei os dados para identificar padrões de acesso à memória, e com base nas informações obtidas, eu poderia pensar se é possível realizar otimizações no código para melhorar a localidade de referência.

Tópico 3

O vetor usado para ordenação no BubbleSort e analisado nos tópicos abaixo foi um vetor de 1000 elementos desordenado aleatoriamente.

Tópico 4

Saída do cachegrind

```
==15895== I   refs:      52,531,434
==15895== I1  misses:         2,215
==15895== LLi misses:         2,099
==15895== I1  miss rate:       0.00%
==15895== LLi miss rate:       0.00%
==15895==
==15895== D   refs:      30,898,264  (19,868,679 rd   + 11,029,585 wr)
==15895== D1  misses:        16,201  (    13,654 rd   +      2,547 wr)
==15895== LLd misses:         9,357  (     7,691 rd   +      1,666 wr)
==15895== D1  miss rate:        0.1% (       0.1%     +        0.0%  )
==15895== LLd miss rate:        0.0% (       0.0%     +        0.0%  )
==15895==
==15895== LL refs:           18,416  (    15,869 rd   +      2,547 wr)
==15895== LL misses:         11,456  (     9,790 rd   +      1,666 wr)
==15895== LL miss rate:         0.0% (       0.0%     +        0.0%  )
```

Funções com maior impacto na execução

1. A função *std::vector* teve 17.995.152 referências (34.26% do total de referências de instruções) e 4.498.788 referências de escrita (40.79% do total de referências de dados de escrita). Essa função teve um impacto significativo no desempenho do programa.
2. A função bubbleSort teve 16.997.964 referências (32.36% das referências de instruções) e 2.000.846 referências de escrita (18.14% das referências de dados de escrita).

Tópico 5

Saída do callgrind

1. A função std::vector é responsável por 17.995.152 (34,26%) das instruções lidas.
2. A função bubbleSort é responsável por 16.997.964 (32,36%) das instruções lidas.
3. A função Troca é responsável por 8.036.416 (15,30%) das instruções lidas.
4. A função std::remove_reference é responsável por 5.273.898 (10,04%) das instruções lidas.
5. Outras funções e localizações de arquivo contribuem com uma porcentagem menor das instruções lidas.

Tópico 6

1. **Quão bem o programa se comporta em termos de memória?**

Analisando a saída do cachegrind percebemos que o algoritmo apresentou uma baixa perda de memória para a entrada analisada. Dessa forma, tanto o cache de instruções quanto o cache de dados estão sendo eficientemente utilizados pelo programa, com taxas de misses muito baixas. Isso indica que o programa está aproveitando bem o uso da memória cache.

2. **Quais estruturas de dados devem ser caracterizadas para melhor entendimento?**

std::vector - Essa estrutura de dados está presente em duas funções e representam uma parte significativa das instruções lidas no programa.

Troca - Está relacionada à operação de troca de elementos do tipo int. Seu uso também é considerável no programa.

3. **Quais segmentos de código devem ser instrumentados para suportar a caracterização?**

Função std::vector. Essa função é responsável por uma quantidade significativa de instruções lidas e está associada à manipulação da estrutura de dados std::vector. Instrumentar essa função permitirá entender melhor como ela é usada e se há oportunidades de otimização.

Função bubbleSort. Essa função também é responsável por uma quantidade considerável de instruções lidas e está relacionada à ordenação do vetor. Instrumentar essa função ajudará a analisar o desempenho do algoritmo de ordenação e identificar possíveis melhorias.

Função Troca. Essa função está associada à operação de troca de elementos int e também é responsável por uma parte significativa das instruções lidas. Instrumentar essa função permitirá uma análise mais detalhada da operação de