CCF 252 - Organização de Computadores I
Trabalho Prático 04 - Memória Cache

Mateus Aparecido - 3858, Artur Papa - 3886, Luciano Belo - 3897

7 de dezembro de 2020

1 Introdução

O trabalho em questão tem como objetivo abordar os conteúdos trabalhados na disciplina de Organização de Computadores I, em específico, como visto no módulo 4, a otimização de algoritmos para aproveitar o desempenho da memória cache [4]. A cache é uma memória menor e mais rápida que armazena cópia dos dados de lugares usados frequentemente pela memória principal e surgiu quando a memória RAM não estava mais acompanhando o desenvolvimento do processador [2].

A priori, a memória de acesso aleatório (Computer Random Access Memory) do computador é um dos componentes mais importantes para determinar o desempenho do seu sistema, se a sua máquina tiver pouca RAM, ele pode ficar cada vez mais lento, mas, na extremidade oposta, você pode instalar muito com pouco ou nenhum benefício adicional.

2 Organização da Cache

Antes da explicação dos algoritmos de ordenação segue uma tabela com as configurações do computador usado nos testes.

Organização da cache do computador usado nos testes

Memória Cache	Valores
Processador	i3-4170
Data width	64 bit
Número de cores da CPU	2
Número de threads	4
Tamanho Cache L1	2 x 32 KB 8-way set associative instruction caches 2 x 32 KB 8-way set associative data caches
Tamanho Cache L2	2 x 256 KB 8-way set associative caches
Tamanho Cache L3	3 MB 12-way set associative shared cache

3 Algoritmos de Ordenação

A posteriori o módulo em questão tem como objetivo mostrar os algoritmos de ordenação usados no desenvolvimento do trabalho, fazendo uma observação de que o grupo optou por usar e otimizar o algoritmo ***heapsort***. Ademais, o algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem. Em outras palavras efetua sua elaboração completa ou parcial. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista.

3.1 Bubblesort

Primeiramente, o bubblesort [1] é um dos algoritmos mais simples levando em consideração a sua implementação, sendo dessa forma o algoritmo com menor desempenho comparado aos demais. Além disso, o algoritmo em questão tem como base sempre iterar todo o array quantas vezes forem necessárias até que os itens estejam ordenados.

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```

Figura 1: Algoritmo Bubblesort

3.2 Radixsort

A ideia do radixsort [8] é fazer a classificação dígito por dígito, começando pelo número menos significativo ao mais significativo. O radixsort usa classificação por contagem como uma sub-rotina. Suponha que tenhamos um array de oito elementos. Primeiro classificaremos os elementos com base no valor da casa da unidade, em seguida, classificaremos os elementos com base nos valores da casa da dezena e assim por diante.

```
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);
    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Figura 2: Algoritmo Radixsort

3.3 Quicksort

O quicksort [7] é o algoritmo de ordenação mais rápido que se conhece para uma ampla variedade de situações, sendo provavelmente mais utilizado do que qualquer outro algoritmo. Nesse aspecto, a ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. A seguir, os dois problemas menores são ordenados independentemente, e, depois os resultados são combinados para produzir a solução do problema maior.

```c
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Figura 3: Algoritmo Quicksort

3.4 Heapsort

Heapsort [3] é um método de ordenação cujo princípio de funcionamento seleciona o menor item do vetor e em seguida o troca pelo item que está na primeira posição da lista; repita essas duas operações com os ***n-1*** elementos restantes, depois com as ***n-2***, e assim sucessivamente. O custo para encontrar o menor (ou maior) item entre n itens custa ***n-1*** comparações.

```c
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
                                int i
    for (int i = n / 2 - 1; i ≥ 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--)
    {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Figura 4: Algoritmo Heapsort

4 Métricas de desempenho

4.1 Arquivos de entrada

Antes de tudo vale fazer uma observação de que o grupo optou por criar um programa em python para gerar os números de forma randômica para cada caso de teste, sendo que os valores vão de 0 a 100 e a quantidade de números gerados é escolhido via linha de comando no terminal.

```
import random
from random import seed
seed(100)
t=int(input("Digite a quantidade de itens no arquivo:"))
arq=open("../src/data/%d.txt"%t,"w")
chave=[random.randint(0,100)for i in range(t)]
for i in range(t):
    arq.write("%d \n"%(chave[i]))
arq.close()
```

Figura 5: Algoritmo para gerar os testes

4.2 Métricas de desempenho para o Perf

Em primeiro lugar vale ressaltar que o perf é uma ferramenta de análise, depuração e monitoramento, ele oferece suporte a contadores de desempenho de hardware, pontos de rastreamento, contadores de desempenho de software (como o hrtimer) e sondas dinâmicas (como o kprobes). Além disso, em 2012, dois engenheiros da IBM reconheceram o perf como uma das duas ferramentas de perfil de contador de desempenho mais comumente usadas no Linux.

4.3 Entrada com 1000 números

Para começar iremos fazer a comparação para cada algoritmo com uma entrada de mil números. A partir das análises realizadas pelo grupo, pode-se inferir que o algoritmo de ordenação heapsort obteve o melhor resultado, seguido pelo quicksort, radixsort e por último bubblesort.

```
Performance counter stats for './main':

          1,51 msec task-clock                #    0,000 CPUs utilized
     5.492.242      cycles                    #    3,635 GHz
     3.958.290      instructions              #    0,72  insn per cycle
        35.441      cache-references          #   23,456 M/sec
        16.244      cache-misses              #   45,834 % of all cache refs

   3,457347103 seconds time elapsed

   0,002063000 seconds user
   0,000000000 seconds sys
```

Figura 6: Resultado do Heapsort para mil números

```
Performance counter stats for './main':

          1,75 msec task-clock                #    0,000 CPUs utilized
     6.223.402      cycles                    #    3,548 GHz
     3.584.987      instructions              #    0,58  insn per cycle
        43.522      cache-references          #   24,813 M/sec
        16.040      cache-misses              #   36,855 % of all cache refs

   4,166924401 seconds time elapsed

   0,002756000 seconds user
   0,000000000 seconds sys
```

Figura 7: Resultado do Quicksort para mil números

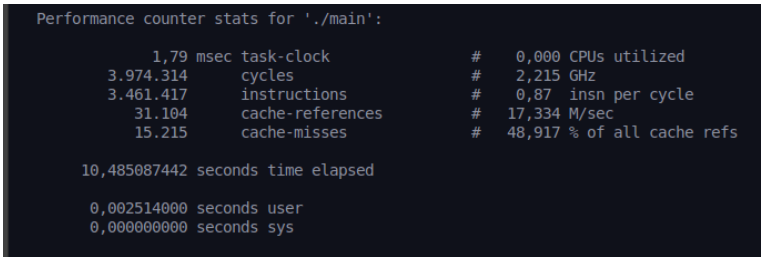

Figura 8: Resultado do Radixsort para mil números

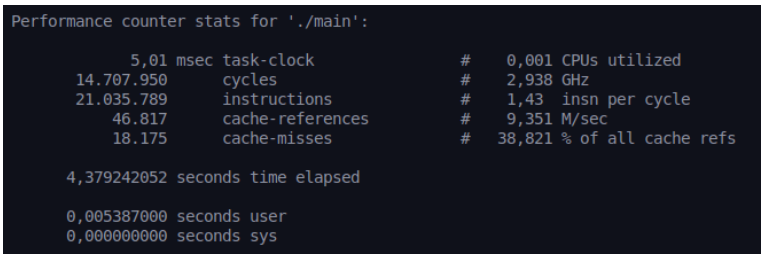

Figura 9: Resultado do Bubblesort para mil números

Tabela comparativa dos resultados entre os algoritmos e o heapsort

Resultados	Quicksort	Radixsort	Bubblesort
Tempo	15,89% maior	18,54% maior	231,78% maior
Ciclos	13,33% maior	27,63% menor	167,78% maior
Instruções	9,44% menor	12,55% menor	431,43% maior
Cache-Ref	22,80% maior	12,23% menor	32,09% maior
Cache-misses	1,25% menor	6,33% menor	11,88% maior

4.4 Entrada com 10000 números

Logo após fazermos os testes para mil números agora iremos testar os algoritmos para uma entrada grande de dez mil números. Diferentemente do primeiro teste, foi observado pelo grupo que o algoritmo quicksort se saiu melhor entre os demais, seguido pelo radixsort, heapsort e por último bubblesort.

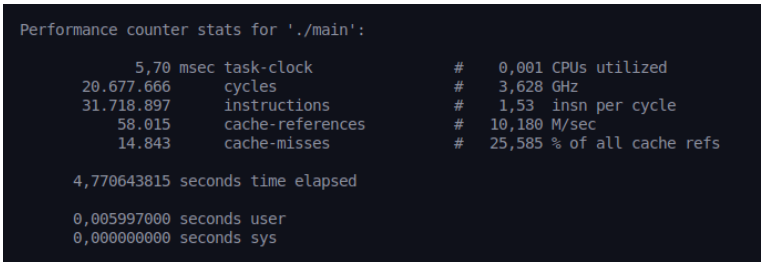

Figura 10: Resultado do Quicksort para dez mil números

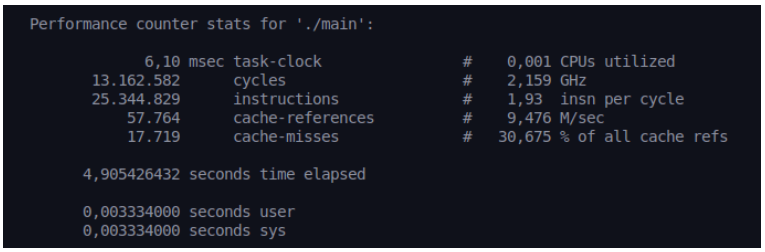

Figura 11: Resultado do Radixsort para dez mil números

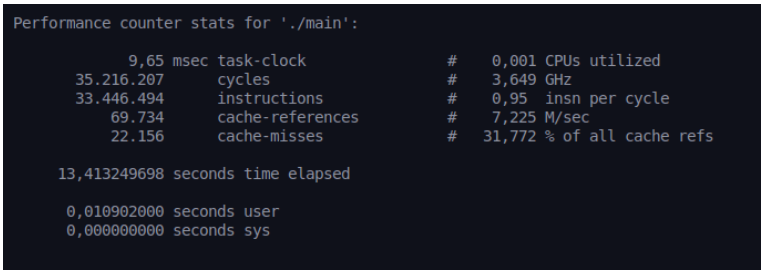

Figura 12: Resultado do Heapsort para dez mil números

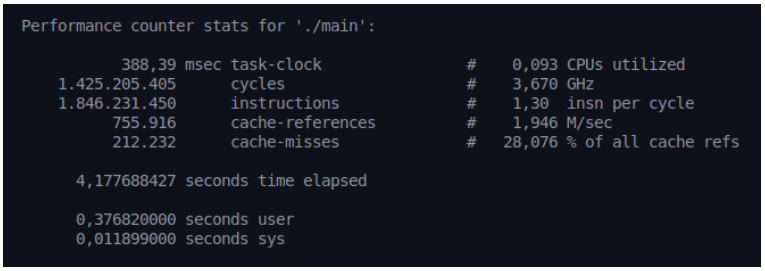

Figura 13: Resultado do Bubblesort para dez mil números

Tabela comparativa dos resultados entre os algoritmos e o quicksort

Resultados	Radixsort	Heapsort	Bubblesort
Tempo	7,01% maior	69,29% maior	6713,85% maior
Ciclos	36,34% menor	70,31% maior	6792,48% maior
Instruções	20,09% menor	5,44% maior	5720,60% maior
Cache-Ref	0,43% menor	20,19% maior	1202,96% maior
Cache-misses	19,37% maior	49,26% maior	1329,84% maior

4.5 Métricas de desempenho para o Valgrind

A priori o valgrind é um sistema para depurar e criar perfis de programas Linux. Com seu conjunto de ferramentas, você pode detectar automaticamente muitos bugs de threading e gerenciamento de memória, evitando horas de busca frustrante de bugs e tornando seus programas mais estáveis. Ademais, o conjunto de ferramentas valgrind fornece várias ferramentas de depuração e criação de perfil que o ajudam a tornar seus programas mais rápidos e corretos.

4.6 Entrada com 1000 números

Em princípio fizemos os testes para uma entrada menor, da mesma maneira feita com o perf, usando uma entrada de 1000 números. Desta forma, foi observado pelo grupo que o algoritmo radixsort se saiu melhor no desempenho com as caches, seguidos pelo quicksort, heapsort e bubblesort por último.

```
==31513==
==31513== I   refs:      2,575,562
==31513== I1  misses:        1,341
==31513== LLi misses:        1,303
==31513== I1  miss rate:      0.05%
==31513== LLi miss rate:      0.05%
==31513==
==31513== D   refs:        989,665  (666,935 rd   + 322,730 wr)
==31513== D1  misses:        3,411  (  2,658 rd   +     753 wr)
==31513== LLd misses:        2,797  (  2,160 rd   +     637 wr)
==31513== D1  miss rate:       0.3% (    0.4%     +     0.2%  )
==31513== LLd miss rate:       0.3% (    0.3%     +     0.2%  )
==31513==
==31513== LL refs:           4,752  (  3,999 rd   +     753 wr)
==31513== LL misses:         4,100  (  3,463 rd   +     637 wr)
==31513== LL miss rate:        0.1% (    0.1%     +     0.2%  )
```

Figura 14: Resultado do Radixsort para mil números no valgrind

```
==31770==
==31770== I   refs:      2,599,488
==31770== I1  misses:        1,331
==31770== LLi misses:        1,294
==31770== I1  miss rate:      0.05%
==31770== LLi miss rate:      0.05%
==31770==
==31770== D   refs:      1,094,515  (736,216 rd   + 358,299 wr)
==31770== D1  misses:        3,384  (  2,656 rd   +     728 wr)
==31770== LLd misses:        2,775  (  2,160 rd   +     615 wr)
==31770== D1  miss rate:       0.3% (    0.4%     +     0.2%  )
==31770== LLd miss rate:       0.3% (    0.3%     +     0.2%  )
==31770==
==31770== LL refs:           4,715  (  3,987 rd   +     728 wr)
==31770== LL misses:         4,069  (  3,454 rd   +     615 wr)
==31770== LL miss rate:        0.1% (    0.1%     +     0.2%  )
```

Figura 15: Resultado do Quicksort para mil números no valgrind

```
==31942==
==31942== I   refs:      3,090,301
==31942== I1  misses:        1,335
==31942== LLi misses:        1,295
==31942== I1  miss rate:      0.04%
==31942== LLi miss rate:      0.04%
==31942==
==31942== D   refs:      1,389,878  (928,125 rd   + 461,753 wr)
==31942== D1  misses:        3,372  (  2,656 rd   +     716 wr)
==31942== LLd misses:        2,772  (  2,160 rd   +     612 wr)
==31942== D1  miss rate:       0.2% (    0.3%     +     0.2%  )
==31942== LLd miss rate:       0.2% (    0.2%     +     0.1%  )
==31942==
==31942== LL refs:           4,707  (  3,991 rd   +     716 wr)
==31942== LL misses:         4,067  (  3,455 rd   +     612 wr)
==31942== LL miss rate:        0.1% (    0.1%     +     0.1%  )
```

Figura 16: Resultado do Heapsort para mil números no valgrind

```
==31374==
==31374== I   refs:      20,098,231
==31374== I1  misses:         1,331
==31374== LLi misses:         1,293
==31374== I1  miss rate:       0.01%
==31374== LLi miss rate:       0.01%
==31374==
==31374== D   refs:      10,625,240  (8,651,477 rd   + 1,973,763 wr)
==31374== D1  misses:         3,370  (    2,656 rd   +       714 wr)
==31374== LLd misses:         2,770  (    2,160 rd   +       610 wr)
==31374== D1  miss rate:        0.0% (      0.0%     +       0.0%  )
==31374== LLd miss rate:        0.0% (      0.0%     +       0.0%  )
==31374==
==31374== LL refs:            4,701  (    3,987 rd   +       714 wr)
==31374== LL misses:          4,063  (    3,453 rd   +       610 wr)
==31374== LL miss rate:         0.0% (      0.0%     +       0.0%  )
```

Figura 17: Resultado do Bubblesort para mil números no valgrind

Resultados	Quicksort	Heapsort	Bubblesort
I refs	0,92% maior	19,98% maior	680,34% maior
I1 miss rate	Igual	20% menor	80% menor
D refs	10,59% maior	40,43% maior	973,61% maior
D1 miss rate	Igual	20% menor	100% menor
LL refs	0,77% menor	0,94% menor	1,03% menor
LL miss rate	Igual	Igual	100% menor

4.7 Entrada com 10000 números

Após fazermos os testes para uma entrada pequena, fizemos os testes para uma entrada grande de dez mil números. Foi notado pelo grupo que assim como nos valores para mil números, a ordem dos algoritmos se manteve a mesma que a do tópico anterior.

```
==31684==
==31684== I   refs:      22,684,903
==31684== I1  misses:         1,343
==31684== LLi misses:         1,305
==31684== I1  miss rate:       0.01%
==31684== LLi miss rate:       0.01%
==31684==
==31684== D   refs:       8,865,349  (5,873,029 rd   + 2,992,320 wr)
==31684== D1  misses:        13,341  (    8,875 rd   +     4,466 wr)
==31684== LLd misses:         3,924  (    2,162 rd   +     1,762 wr)
==31684== D1  miss rate:        0.2% (      0.2%     +       0.1%  )
==31684== LLd miss rate:        0.0% (      0.0%     +       0.1%  )
==31684==
==31684== LL refs:           14,684  (   10,218 rd   +     4,466 wr)
==31684== LL misses:          5,229  (    3,467 rd   +     1,762 wr)
==31684== LL miss rate:         0.0% (      0.0%     +       0.1%  )
```

Figura 18: Resultado do Radixsort para dez mil números no valgrind

```
==31901==
==31901== I   refs:      29,297,458
==31901== I1  misses:         1,333
==31901== LLi misses:         1,296
==31901== I1  miss rate:       0.00%
==31901== LLi miss rate:       0.00%
==31901==
==31901== D   refs:      13,384,574  (9,964,251 rd   + 3,420,323 wr)
==31901== D1  misses:         7,637  (    6,034 rd   +     1,603 wr)
==31901== LLd misses:         3,395  (    2,162 rd   +     1,233 wr)
==31901== D1  miss rate:        0.1% (      0.1%     +       0.0%  )
==31901== LLd miss rate:        0.0% (      0.0%     +       0.0%  )
==31901==
==31901== LL refs:            8,970  (    7,367 rd   +     1,603 wr)
==31901== LL misses:          4,691  (    3,458 rd   +     1,233 wr)
==31901== LL miss rate:         0.0% (      0.0%     +       0.0%  )
```

Figura 19: Resultado do Quicksort para dez mil números no valgrind

```
==32193==
==32193== I   refs:      30,978,532
==32193== I1  misses:         1,337
==32193== LLi misses:         1,297
==32193== I1  miss rate:       0.00%
==32193== LLi miss rate:       0.00%
==32193==
==32193== D   refs:      14,753,306  (9,825,914 rd   + 4,927,392 wr)
==32193== D1  misses:         7,016  (    5,656 rd   +     1,360 wr)
==32193== LLd misses:         3,327  (    2,162 rd   +     1,165 wr)
==32193== D1  miss rate:        0.0% (      0.1%     +       0.0%  )
==32193== LLd miss rate:        0.0% (      0.0%     +       0.0%  )
==32193==
==32193== LL refs:            8,353  (    6,993 rd   +     1,360 wr)
==32193== LL misses:          4,624  (    3,459 rd   +     1,165 wr)
==32193== LL miss rate:         0.0% (      0.0%     +       0.0%  )
```

Figura 20: Resultado do Heapsort para dez mil números no valgrind

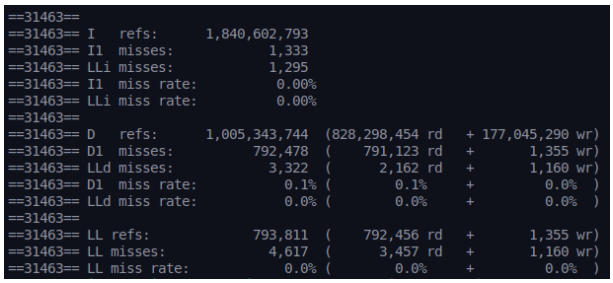

Figura 21: Resultado do Bubblesort para dez mil números no valgrind

Tabela comparativa dos resultados entre os algoritmos e o radixsort

Resultados	Quicksort	Heapsort	Bubblesort
I refs	29,14% maior	36,56% maior	8013,67% maior
I1 miss rate	100% menor	100% menor	100% menor
D refs	57,60% maior	66,41% maior	11240,14% maior
D1 miss rate	50% menor	100% menor	50% menor
LL refs	38,91% menor	43,11% menor	5305,95% maior
LL miss rate	Igual	Igual	Igual

5 Heapsort otimizado

Em primeiro lugar, pode-se dizer que o algoritmo heapsort assim como já exposto anteriormente seleciona o menor item do vetor e em seguida o troca com o item que está na primeira posição, desta forma uma das maneiras de otimizar o algoritmo é, ao contrário do original percorrer o array até o ponto necessário ao invés de verificar item a item até o fim da lista.

Ademais, o ***heapify*** é a função responsável por manter a propriedade da estrutura heap em cada subárvore enraizada com o nó ***i*** do array. Outrossim, o grupo optou por utilizar o algoritmo ***build-max*** que é um procedimento executado em tempo linear produzindo um 'heap' máximo a partir de um vetor de entrada não ordenada.

```c
void build_max_heap(int *arr, int length)
{
    for (int i = 1; i < length; i++)
    {
        int f = i + 1;
        while (f > 1 && arr[f / 2] < arr[f])
        {
            swap(&arr[f / 2], &(arr[f]));
            f /= 2;
        }
    }
}
```

Figura 22: Função buildmax Heapsort otimizado

5.1 Comparação dos valores antes e depois pelo perf

A priori, após fazermos os testes entre cada algoritmo iremos comparar o código de ordenação heapsort antes e depois da otimização. A princípio iremos mostrar os experimentos utilizando a ferramenta perf e com a entrada menor (1000 números) e a maior (10000 números). Do mesmo modo vale dizer que a primeira e a terceira imagem se referem ao algoritmo sem otimização e a segunda e quarta imagem se referem ao código já otimizado, sendo que as duas primeiras se tratam do valor menor e as duas últimas do valor maior.

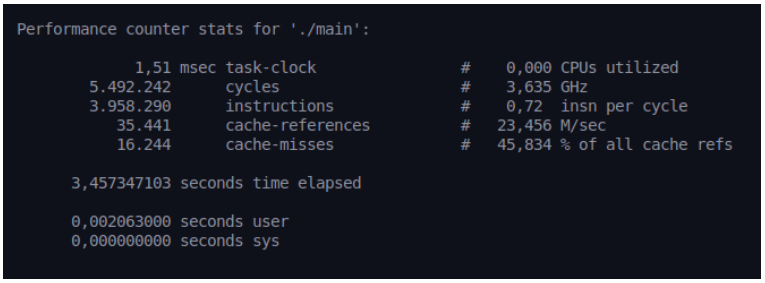

Figura 23: Resultado do Heapsort para mil números no perf

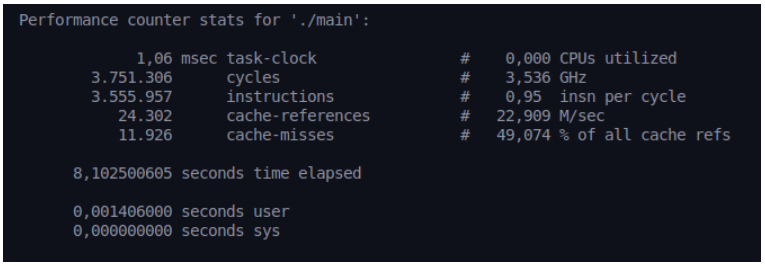

Figura 24: Resultado do Heapsort otimizado para mil números no perf

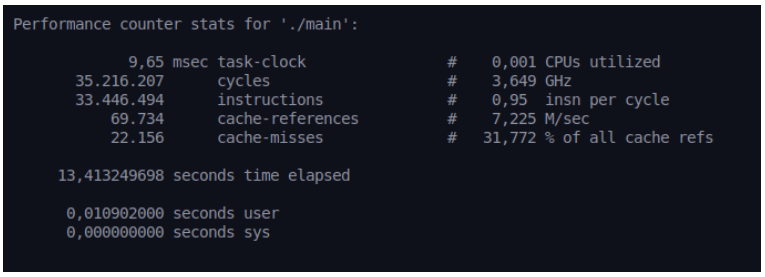

Figura 25: Resultado do Heapsort para dez mil números no perf

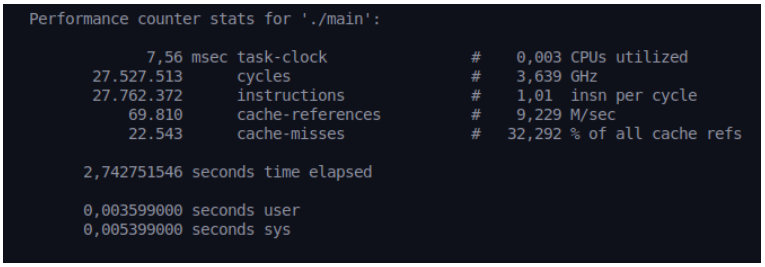

Figura 26: Resultado do Heapsort otimizado para dez mil números no perf

Heapsort antes e depois da otimização com os dados do perf (1000 números)

Resultados	Antes	Depois
Tempo	1,51 msec	1,06 msec
Ciclos	5.492.242	3.751.306
Instruções	3.958.290	3.555.957
Cache-Ref	35.441	24.302
Cache-misses	16.244	11.926

Heapsort antes e depois da otimização com os dados do perf (10000 números)

Resultados	Antes	Depois
Tempo	9,65 msec	7,56 msec
Ciclos	35.216.207	27.527.513
Instruções	33.446.494	27.762.372
Cache-Ref	69.734	69.810
Cache-misses	22.156	22.543

5.2 Comparação dos valores antes e depois pelo valgrind

A posteriori, faremos agora as comprovações para a ferramenta valgrind usando valores menores (1000 números) e maiores (10000 números) para a entrada. Posto isso, é válido ressaltar que a sequência de ordens e valores das imagens seguem a mesma organização explicada na seção anterior.

```
==31942==
==31942== I   refs:      3,090,301
==31942== I1  misses:        1,335
==31942== LLi misses:        1,295
==31942== I1  miss rate:      0.04%
==31942== LLi miss rate:      0.04%
==31942==
==31942== D   refs:      1,389,878  (928,125 rd   + 461,753 wr)
==31942== D1  misses:        3,372  (  2,656 rd   +     716 wr)
==31942== LLd misses:        2,772  (  2,160 rd   +     612 wr)
==31942== D1  miss rate:       0.2% (    0.3%     +     0.2%  )
==31942== LLd miss rate:       0.2% (    0.2%     +     0.1%  )
==31942==
==31942== LL refs:           4,707  (  3,991 rd   +     716 wr)
==31942== LL misses:         4,067  (  3,455 rd   +     612 wr)
==31942== LL miss rate:        0.1% (    0.1%     +     0.1%  )
```

Figura 27: Resultado do Heapsort para mil números no valgrind

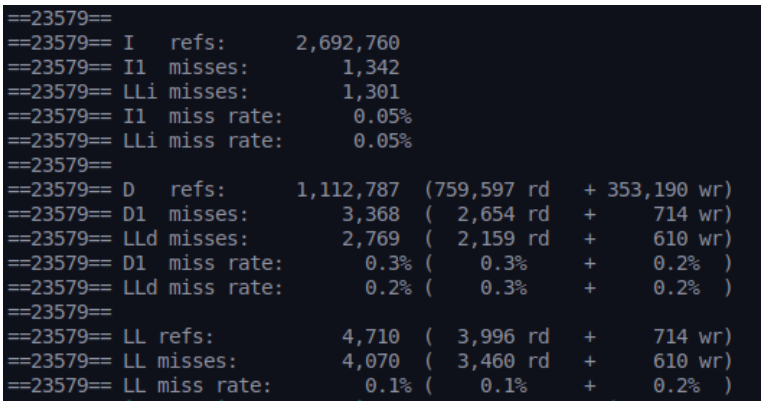

Figura 28: Resultado do Heapsort otimizado para mil números no valgrind

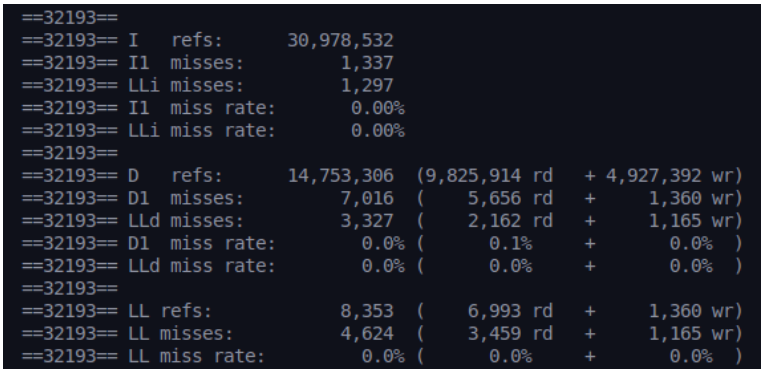

Figura 29: Resultado do Heapsort para dez mil números no valgrind

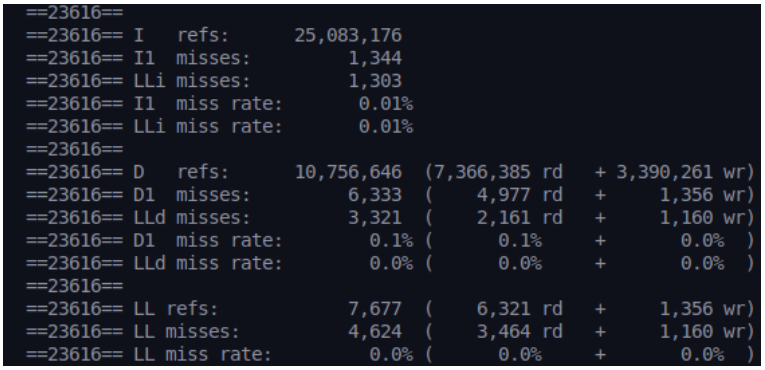

Figura 30: Resultado do Heapsort otimizado para dez mil números no valgrind

Heapsort antes e depois da otimização com os dados do valgrind (1000 números)

Resultados	Antes	Depois
I refs	3.090.301	2.692.720
I1 miss rate	0.04%	0.05%
D refs	1.339.878	1.112.787
D1 miss rate	0.2%	0.3%
LL refs	4.707	4.710
LL miss rate	0.1%	0.1%

Heapsort antes e depois da otimização com os dados do valgrind (10000 números)

Resultados	Antes	Depois
I refs	30.968.532	25.083.176
I1 miss rate	0.0%	0.01%
D refs	14.753.306	10.756.646
D1 miss rate	0.0%	0.1%
LL refs	8.353	7.677
LL miss rate	0.0%	0.0%

6 Conclusão

Desta forma, podemos concluir que o trabalho em questão foi desenvolvido conforme o esperado, atingindo todas as especificações requeridas na descrição do mesmo, já que o intuito principal do trabalho foi atingido, o estudo e aplicação da memória cache.

Por conseguinte, vale ressaltar que os sites oficiais dos softwares perf ([6] e [5]) e valgrind ([10] e [9]) foram de suma importância para o desenvolvimento do projeto. Outrossim, a utilização do perf e do valgrind auxiliaram na consolidação dos conteúdos trabalhados na disciplina de Algoritmos e Estrutura de Dados I [11], em que foi mostrado a eficiência de algoritmos como o quicksort e heapsort. Em adição, foi comprovado pelos testes a validez dos algoritmos supracitados, haja vista que pode ser visualizado os resultados obtidos a partir de cada experimento.

Posto isso, é válido dizer que apesar das dificuldades na implementação do código, o grupo foi capaz de superar e corrigir quaisquer erros no desenvolvimento dos algoritmos. Haja vista que o trabalho foi executado conforme o planejado, sendo tratado tudo que foi pedido pelo professor e pelos monitores. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar os algoritmos de ordenação e demonstrar como as operações de acesso à memória são relevantes no desempenho geral de um algoritmo.

Referências

[1] bubbleweb:https://www.geeksforgeeks.org/bubble-sort/.

[2] cache:https://www.geeksforgeeks.org/cache-memory-in-computer-organization/#:~:text=Cache%20memory%20is%20an%20extremely,data%20from%20the%20Main%20memory..

[3] heapweb:https://www.geeksforgeeks.org/heap-sort/.

[4] infoescola:https://www.infoescola.com/informatica/memoria-cache/.

[5] perfweb2:https://developers.redhat.com/blog/2014/03/10/determining-whether-an-application-has-poor-cache-performance-2/.

[6] perfweb:https://en.wikipedia.org/wiki/Perf_(Linux)#cite_note-haas-16.

[7] quickweb:https://www.geeksforgeeks.org/quick-sort/.

[8] radixweb:https://www.geeksforgeeks.org/radix-sort/.

[9] valgrindweb2:https://tools.kali.org/reverse-engineering/valgrind.

[10] valgrindweb:https://www.valgrind.org/docs/manual/quick-start.html.

[11] Nivio Ziviani. *Projeto de algoritmos com implementação em Pascal e C, 3ª edição*. 2017.