

Memória Cache (TP3) - OC I

Bárbara Pagnocca Andrade - 5061, Lucas Fonseca Sabino Lana - 5105

¹UNIVERSIDADE FEDERAL DE VIÇOSA · CAMPUS FLORESTAL

`barbara.andrade@ufv.br, lucas.lana@ufv.br`

1. Introdução

Este trabalho tem como objetivo demonstrar como as operações de acesso à memória são relevantes no desempenho de um algoritmo. Dessa forma, ao final deste trabalho, esperamos entender melhor como a lógica implementada no algoritmo está facilitando ou não o desempenho da cache do processador.

Entender a forma como os dados são armazenados e acessados pela memória cache é crucial para a otimização de algoritmos. A configuração da cache do processador influencia diretamente no tempo de resposta das operações, e, portanto, sua análise pode revelar oportunidades de melhorar a eficiência do código.

A partir da conclusão deste trabalho, além de avaliar a lógica e a complexidade dos algoritmos, também iremos focar em estratégias que otimizem seu desempenho em relação ao uso da cache, buscando reduzir latências e melhorar a velocidade geral de execução.

***OBS:** A tarefa do collab não foi compreendida claramente pela dupla, além de que o código do grupo indicou inúmeros erros ao ser colocado no campo destinado para isso no collab, apesar de nossos esforços. Acreditamos que o intuito era mostrar como as diferentes configurações de memória cache podem ter desempenhos diferentes para os mesmos códigos. Se for este o objetivo do collab, acreditamos tê-lo contemplado no trabalho, o que será mostrado, analisado e explicado nesta documentação. O PDF gerado pela tarefa será enviado também, mas deixamos claro que não funcionou com o nosso código.*

2. CPU

Para conduzir os experimentos propostos pelo trabalho o computador que será usado é um VivoBook ASUSLaptop X515JA, com um processador Intel® Core™ i5-1035G1. Para termos mais detalhes sobre o processador consultamos o site <http://www.cpu-world.com>, e também os detalhes relacionados ao objetivo do trabalho que o sistema operacional Fedora 40 disponibiliza na parte de sistema do menu de configurações.

A partir das informações fornecidas por ambas as fontes podemos dizer que a organização da cache possui a cache L1 que é dividida entre cache de dados, com 192 KiB distribuídos em 4 instâncias, e cache de instruções, com 128 KiB também distribuídos em 4 instâncias. A cache L2 possui um tamanho de 2 MiB, divididos em 4 instâncias, enquanto a cache L3 é de 6 MiB, com uma única instância.

Embora não sejam fornecidas informações específicas sobre a associatividade, é comum que caches desse tipo, em processadores Intel, utilizem associatividade por conjunto (set-associative cache).

■ Identification results

■ Model number naming conventions - 1 match

Intel 10th generation Core i3, Core i5 and Core i7 mobile CPUs (new naming convention)	
i5	Processor family: Core i5 Mobile
-	
10	Processor generation: 10th generation (Ice Lake)
3	Feature / Performance identifier
5	Thermal Design Power: 15 Watt
G	
1	Integrated graphics: Basic graphics

[\[Identification details\]](#)

■ Processor number i5-1035G1

Processor number	i5-1035G1
Family	Core i5 Mobile
Technology (micron)	0.01
Processor speed (GHz)	1
L2 cache size (KB)	2048
L3 cache size (MB)	6
The number of cores	4
EM64T	Supported
HyperThreading technology	Supported
Virtualization technology	Supported
Enhanced SpeedStep technology	Supported
Execute-Disable bit feature	Supported

[\[Processor numbers of mobile Intel CPUs\]](#)

Please see the ["i5-1035G1" identification page](#) for complete results.

Figura 1. Informações fornecidas pelo site <http://www.cpu-world.com>

```
Architecture:                x86_64
CPU op-mode(s):              32-bit, 64-bit
Address sizes:                39 bits physical, 48 bits virtual
Byte Order:                   Little Endian
CPU(s):                       8
On-line CPU(s) list:          0-7
Vendor ID:                    GenuineIntel
Model name:                   Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
CPU family:                   6
Model:                        126
Thread(s) per core:           2
Core(s) per socket:           4
Socket(s):                    1
Stepping:                     5
CPU(s) scaling MHz:           29%
CPU max MHz:                  3600,0000
CPU min MHz:                  400,0000
BogoMIPS:                     2380,80
Virtualization:               VT-x
L1d cache:                    192 KiB (4 instances)
L1i cache:                    128 KiB (4 instances)
L2 cache:                     2 MiB (4 instances)
L3 cache:                     6 MiB (1 instance)
NUMA node(s):                 1
NUMA node0 CPU(s):            0-7
```

Figura 2. Informações relacionadas, fornecidas pelo SO

3. Desenvolvimento

3.1. Métodos de Ordenação

Como requisitado na documentação, pesquisamos as implementações para os algoritmos de ordenação; Bubble Sort, Quick Sort, Radix Sort. Para o algoritmo de ordenação, nossa escolha foi o Selection Sort.

Os links das páginas das quais foram retiradas as implementações dos códigos de ordenação estão nas referências deste documento.

3.1.1. Bubble Sort

A cada passagem pelo conjunto de dados, o Bubble Sort verifica se um elemento é maior que o próximo. Se for, eles trocam de lugar. Esse processo continua repetidamente, com o maior elemento "flutuando" para o final da lista em cada passagem. Com o tempo, os elementos maiores se acumulam no final da lista, e a parte ainda não ordenada diminui. Embora seja fácil de entender e implementar, o Bubble Sort não é muito eficiente para grandes conjuntos de dados.

```
void bubbleSort(int *vetor, int tamanho) {
    int auxiliar=0;

    for (int atual = 0; atual < tamanho; atual++) {
        for (int proximo = 0; proximo < tamanho - 1; proximo++) {
            if (vetor[proximo] > vetor[proximo + 1]) {
                auxiliar = vetor[proximo];
                vetor[proximo] = vetor[proximo + 1];
                vetor[proximo + 1] = auxiliar;
            }
        }
    }
}
```

Figura 3. Implementação Bubble Sort

3.1.2. Radix Sort

O Radix Sort, em vez de comparar elementos diretamente, os organiza com base nos seus dígitos individuais. A ordenação começa pelo dígito menos significativo e vai até o mais significativo. Para cada dígito, um algoritmo de ordenação estável, como o Counting Sort, é usado para garantir que a ordem relativa dos elementos com o mesmo dígito seja mantida. Após classificar por todos os dígitos, os números estarão ordenados. O Radix Sort é particularmente eficiente para grandes conjuntos de números inteiros.

```

int getMax(int *arr, int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int *arr, int n, int exp) {
    int output[n];
    int i, count[10] = { 0 };

    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int *arr, int n) {
    int m = getMax(arr, n);

    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

```

Figura 4. Implementação do Radix Sort

3.1.3. Quick Sort

O Quick Sort é um dos algoritmos de ordenação mais utilizados devido à sua eficiência prática. Funciona escolhendo um elemento como pivô e particionando a lista em duas: uma com elementos menores que o pivô e outra com elementos maiores. O pivô é então colocado na sua posição correta. Em seguida, o Quick Sort é recursivamente aplicado nas sub-listas resultantes. A escolha do pivô pode afetar significativamente o desempenho do algoritmo. Acaba se tornando-o mais eficiente para a maioria das situações.

```

void quick_sort(int *a, int left, int right) {
    int i, j, x, y;

    i = left;
    j = right;
    x = a[(left + right) / 2];

    while(i <= j) {
        while(a[i] < x && i < right) {
            i++;
        }
        while(a[j] > x && j > left) {
            j--;
        }
        if(i <= j) {
            y = a[i];
            a[i] = a[j];
            a[j] = y;
            i++;
            j--;
        }
    }

    if(j > left) {
        quick_sort(a, left, j);
    }
    if(i < right) {
        quick_sort(a, i, right);
    }
}

```

Figura 5. Implementação Quick Sort

3.1.4. Selection Sort

O Selection Sort é um algoritmo que organiza uma lista encontrando o menor elemento e colocando-o na primeira posição. Após cada iteração, o algoritmo ignora a parte já ordenada da lista e continua a buscar o menor elemento nos elementos restantes, repetindo o processo até que a lista esteja totalmente ordenada. Este método se destaca por realizar um número limitado de trocas, o que pode ser útil em situações onde esse critério é relevante. Contudo, devido à sua complexidade de tempo, não é recomendado para listas grandes e é mais comumente usado em pequenos conjuntos de dados ou em situações

onde a memória limitada é uma preocupação.

```
void selection_Sort(int *vetor, int n) {
    for(int i = 0; i < n - 1; i++) {
        int menor = i;
        for(int j = i + 1; j < n; j++) {
            if (vetor[j] < vetor[menor]) menor = j;
        }
        int aux = vetor[i];
        vetor[i] = vetor[menor];
        vetor[menor] = aux;
    }
}
```

Figura 6. Implementação Selection sort

3.2. Entrada de Dados

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10000
#define MIN 1000
#define LIMITE 1024

void preenche_vetore_dados_MAX (int *vetor_MAX){
    srand(time(NULL)); // Inicializa a semente com o tempo atual
    for (int i = 0; i < MAX; i++) {
        vetor_MAX[i] = rand() % (LIMITE + 1); // Limita os valores para ficarem entre 0 e LIMITE
    }
}

void preenche_vetore_dados_MIN (int *vetor_MIN){
    for (int j = 0; j < MIN; j++) {
        vetor_MIN[j] = rand() % (LIMITE + 1); // Limita os valores para ficarem entre 0 e LIMITE
    }
}
```

Figura 7. Funções que preenchem os vetores de dados

A entrada de dados de dois vetores, um contendo 1000 células (entrada pequena) e o outro contendo 10000 células (entrada grande) é feita colocando números inteiros "aleatórios" de 0 à 1024, a partir da função rand.

3.3. Algoritmo Escolhido (Multiplicação de Matriz por Escalar)

Para escolher qual algoritmo iremos utilizar para tentar melhorar o desempenho da memória cache, relembramos o que o professor Nacif comentou em aula sobre programas que envolvem matrizes. Ao acessar uma célula da matriz é carregada automaticamente toda a linha que contém a célula escolhida pelo programa.

O funcionamento é semelhante para matrizes que não cabem por completo na cache, se for o caso a matriz é subdividida em sub-matrizes que agora cabem na cache, e assim funcionam da forma mencionada anteriormente.

Com esta informação em mente, a escolha de uma operação que envolva matriz era o objetivo, sendo que o resultado desta operação deveria ser igual seja realizando linha por linha ou coluna por coluna.

A operação escolhida, então, foi a multiplicação de uma matriz por um escalar. Esta operação é simples e relativamente fácil de modificar para melhorar o desempenho da cache, então foi escolhida para nos ajudar a notar as diferenças de desempenho e entender uma forma de melhorá-lo.

3.3.1. Sem otimização para CACHE

Nesta implementação não se leva em consideração o melhor desempenho da memória cache. A operação de multiplicação de matriz por escalar foi feita multiplicando o conteúdo de cada célula por um escalar, passando de coluna à coluna.

```
void mul_matriz_coluna(int n, int m, int constante, int Matriz[n][m]){
    //Linhas = n
    //Colunas = m
    for(int coluna = 0; coluna < m; coluna++){
        for(int linha = 0; linha < n; linha++){
            Matriz[linha][coluna] = Matriz[linha][coluna] * constante;
        }
    }
}
```

Figura 8. Multiplicação de matriz pela constante feita coluna por coluna

3.3.2. Com otimização para CACHE

Já nesta implementação leva-se em consideração melhorar o desempenho da memória cache. Para isso, a implementação da operação de multiplicação de matriz por escalar será feita multiplicando o conteúdo de cada célula por um escalar, passando de linha a linha.

```
void mul_matriz_linha(int n, int m, int constante, int Matriz[n][m]){
    //Linhas = n
    //Colunas = m
    for(int linha = 0; linha < n; linha++){
        for(int coluna = 0; coluna < m; coluna++){
            Matriz[linha][coluna] = Matriz[linha][coluna] * constante;
        }
    }
}
```

Figura 9. Multiplicação de matriz pela constante feita linha por linha

4. Resultados

Tivemos alguns problemas para conseguir executar da forma desejada as ferramentas Perf e Valgrind. Para utilizar a ferramenta Perf foi necessário a edição de um arquivo do kernel do sistema, para que o comando conseguisse acessar as informações da memória cache. Houve também uma leve dificuldade de encontrar o comando correto para mostrar apenas as informações desejadas, em ambas ferramentas.

A configuração da memória cache simulada foi a cache de instrução (I1), a cache de dados (D1) e a cache de último nível (LL). A cache de instrução (I1) tem um tamanho de 16 KB, com uma associatividade de 16 vias e um tamanho de linha de cache de 64 bytes. A cache de dados (D1) também possui 16 KB de tamanho, com a mesma associatividade de 16 vias e linhas de cache de 64 bytes. Já a cache de último nível (LL) é maior, com 256 KB de tamanho, 16 vias de associatividade e linhas de cache de 64 bytes.

Cada algoritmo de ordenação foi sujeito a dois casos de teste, um teste com uma entrada de dados menor, e outro com uma entrada de dados maior. Em outras palavras, cada algoritmo de ordenação terá 2 secções de resultados. Uma secção sendo para as imagens utilizando a ferramenta perf, e a outra utilizando a ferramenta Valgrind.

Cada uma destas secções tem duas imagens, uma imagem com a utilização da ferramenta da secção sendo executada no conjunto de dados menor (vetor preenchido com números aleatórios de 1000 posições) e uma imagem sendo executada no conjunto de dados maior (vetor preenchido com números aleatórios de 10000 posições).

4.1. Multiplicação de Matriz por escalar

Ao realizar a mudança no algoritmo a fim de melhorar a performance no acesso da memória cache, utilizamos a ferramenta Perf para comprovar a melhora de desempenho.

Para termos maior certeza dos dados, foi realizado o teste de performance em dois conjuntos de dados, sendo um destes casos a operação com uma matriz menor, quadrada de 100 por 100 e multiplicada por um escalar de valor 12. No outro caso foi realizada a operação com uma matriz maior, quadrada de 1000 por 1000, sendo o valor do escalar igual à 124.

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

          95.884      cache-references      #    10,433 M/sec
          32.110      cache-misses         #    33,49% of all cache refs
           9,19 msec  task-clock            #    0,001 CPUs utilized
          9.335.221    cycles                #    1,016 GHz
         19.713.859    instructions          #    2,11  insn per cycle

10,968818118 seconds time elapsed

 0,005573000 seconds user
 0,004471000 seconds sys
```

Figura 10. Perf da Multiplicação de matriz coluna a coluna (Matriz menor)


```

Performance counter stats for './main':

      87.191      cache-references      #    10,801 M/sec
      28.884      cache-misses          #    33,13% of all cache refs
       8,07 msec  task-clock            #    0,001 CPUs utilized
    8.171.395     cycles                #    1,012 GHz
    19.685.957    instructions          #    2,41  insn per cycle

15,433871173 seconds time elapsed

  0,006077000 seconds user
  0,003004000 seconds sys

```

Figura 11. Perf da Multiplicação de matriz linha a linha (Matriz menor)

```

Performance counter stats for './main':

    3.009.031      cache-references      #    6,602 M/sec
      869.691      cache-misses          #    28,90% of all cache refs
       455,76 msec task-clock            #    0,037 CPUs utilized
    705.088.895    cycles                #    1,547 GHz
    1.845.375.812  instructions          #    2,62  insn per cycle

12,164788640 seconds time elapsed

  0,359615000 seconds user
  0,091726000 seconds sys

```

Figura 12. Perf da Multiplicação de matriz coluna a coluna (Matriz maior)

```

Performance counter stats for './main':

    3.210.735      cache-references      #    6,617 M/sec
      820.494      cache-misses          #    25,55% of all cache refs
       485,20 msec task-clock            #    0,029 CPUs utilized
    748.727.348    cycles                #    1,543 GHz
    1.846.780.555  instructions          #    2,47  insn per cycle

16,959902644 seconds time elapsed

  0,388638000 seconds user
  0,092593000 seconds sys

```

Figura 13. Perf da Multiplicação de matriz linha a linha (Matriz maior)

4.2. Bubble Sort

4.2.1. *Perf*:

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

      56.075      cache-references      #    4,957 M/sec
      26.823      cache-misses          #   47,83% of all cache refs
       11,31 msec task-clock            #    0,002 CPUs utilized
    12.045.277     cycles                #    1,065 GHz
    29.885.259     instructions          #    2,48  insn per cycle

5,498630546 seconds time elapsed

0,009521000 seconds user
0,002719000 seconds sys
```

Figura 14. Perf do Bubble sort para a entrada de dados pequena

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

    130.066      cache-references      #   355,278 K/sec
     50.138      cache-misses          #   38,55% of all cache refs
     366,10 msec task-clock            #    0,050 CPUs utilized
    970.387.460   cycles                #    2,651 GHz
   2.722.223.650   instructions          #    2,81  insn per cycle

7,373215888 seconds time elapsed

0,363844000 seconds user
0,002185000 seconds sys
```

Figura 15. Perf do Bubble sort para a entrada de dados grande

4.2.2. Valgrind:

```
==23801==
==23801== I refs:      28,661,901
==23801== I1 misses:    1,471
==23801== LLi misses:   1,416
==23801== I1 miss rate:  0.01%
==23801== LLi miss rate: 0.00%
==23801==
==23801== D refs:      13,120,738 (12,112,764 rd + 1,007,974 wr)
==23801== D1 misses:    1,731 ( 1,240 rd + 491 wr)
==23801== LLD misses:   1,541 ( 1,071 rd + 470 wr)
==23801== D1 miss rate:  0.0% ( 0.0% + 0.0% )
==23801== LLD miss rate: 0.0% ( 0.0% + 0.0% )
==23801==
==23801== LL refs:      3,202 ( 2,711 rd + 491 wr)
==23801== LL misses:    2,957 ( 2,487 rd + 470 wr)
==23801== LL miss rate:  0.0% ( 0.0% + 0.0% )
```

Figura 16. Valgrind do Bubble sort (Config padrão)

```
==26022==
==26022== I refs:      28,661,901
==26022== I1 misses:    1,725
==26022== LLi misses:   1,416
==26022== I1 miss rate:  0.01%
==26022== LLi miss rate: 0.00%
==26022==
==26022== D refs:      13,120,738 (12,112,764 rd + 1,007,974 wr)
==26022== D1 misses:    2,075 ( 1,538 rd + 537 wr)
==26022== LLD misses:   1,541 ( 1,071 rd + 470 wr)
==26022== D1 miss rate:  0.0% ( 0.0% + 0.1% )
==26022== LLD miss rate: 0.0% ( 0.0% + 0.0% )
==26022==
==26022== LL refs:      3,800 ( 3,263 rd + 537 wr)
==26022== LL misses:    2,957 ( 2,487 rd + 470 wr)
==26022== LL miss rate:  0.0% ( 0.0% + 0.0% )
```

Figura 17. Valgrind do Bubble sort (Config simulada)

4.3. Randix Sort

4.3.1. *Perf*:

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

      60.149      cache-references      #    23,370 M/sec
      28.945      cache-misses          #    48,12% of all cache refs
           2,57 msec task-clock          #    0,001 CPUs utilized
    2.560.079      cycles                #    0,995 GHz
    3.419.209      instructions          #    1,34 insn per cycle

4,968555993 seconds time elapsed

0,002565000 seconds user
0,000977000 seconds sys
```

Figura 18. Perf do Randix sort para a entrada de dados pequena

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

      85.206      cache-references      #    7,669 M/sec
      29.325      cache-misses          #    34,42% of all cache refs
      11,11 msec task-clock          #    0,001 CPUs utilized
    11.174.969      cycles                #    1,006 GHz
    23.917.910      instructions          #    2,14 insn per cycle

9,927420040 seconds time elapsed

0,008153000 seconds user
0,004106000 seconds sys
```

Figura 19. Perf do Randix sort para a entrada de dados grande

4.3.2. Valgrind:

```
==24898==
==24898== I refs:          2,212,549
==24898== I1 misses:       1,481
==24898== LLi misses:      1,424
==24898== I1 miss rate:    0.07%
==24898== LLi miss rate:   0.06%
==24898==
==24898== D refs:          815,084 (533,452 rd + 281,632 wr)
==24898== D1 misses:       1,757 ( 1,240 rd +   517 wr)
==24898== LLd misses:      1,567 ( 1,071 rd +   496 wr)
==24898== D1 miss rate:    0.2% ( 0.2% + 0.2% )
==24898== LLd miss rate:   0.2% ( 0.2% + 0.2% )
==24898==
==24898== LL refs:          3,238 ( 2,721 rd +   517 wr)
==24898== LL misses:       2,991 ( 2,495 rd +   496 wr)
==24898== LL miss rate:    0.1% ( 0.1% + 0.2% )
```

Figura 20. Valgrind do Randix sort (Config padrão)

```
==26387==
==26387== I refs:          2,212,549
==26387== I1 misses:       1,741
==26387== LLi misses:      1,424
==26387== I1 miss rate:    0.08%
==26387== LLi miss rate:   0.06%
==26387==
==26387== D refs:          815,084 (533,452 rd + 281,632 wr)
==26387== D1 misses:       2,111 ( 1,545 rd +   566 wr)
==26387== LLd misses:      1,567 ( 1,071 rd +   496 wr)
==26387== D1 miss rate:    0.3% ( 0.3% + 0.2% )
==26387== LLd miss rate:   0.2% ( 0.2% + 0.2% )
==26387==
==26387== LL refs:          3,852 ( 3,286 rd +   566 wr)
==26387== LL misses:       2,991 ( 2,495 rd +   496 wr)
==26387== LL miss rate:    0.1% ( 0.1% + 0.2% )
```

Figura 21. Valgrind do Randix sort (Config simulada)

4.4. Quick Sort

4.4.1. *Perf*:

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

      57.516      cache-references      #    21,295 M/sec
      28.679      cache-misses          #    49,86% of all cache refs
       2,70 msec  task-clock            #    0,000 CPUs utilized
    2.618.776     cycles                #    0,970 GHz
    3.195.856     instructions          #    1,22  insn per cycle

11,644417824 seconds time elapsed

 0,000986000 seconds user
 0,002905000 seconds sys
```

Figura 22. Perf do Quick sort para a entrada de dados pequena

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

      71.300      cache-references      #    6,643 M/sec
      27.484      cache-misses          #    38,55% of all cache refs
      10,73 msec  task-clock            #    0,002 CPUs utilized
    10.895.175     cycles                #    1,015 GHz
    22.556.209     instructions          #    2,07  insn per cycle

 6,427932193 seconds time elapsed

 0,008702000 seconds user
 0,002926000 seconds sys
```

Figura 23. Perf do Quick sort para a entrada de dados grande

4.4.2. Valgrind:

```
==24611==
==24611== I refs:      2,018,684
==24611== I1 misses:    1,474
==24611== LLi misses:   1,418
==24611== I1 miss rate:  0.07%
==24611== LLi miss rate: 0.07%
==24611==
==24611== D refs:      816,527 (536,050 rd + 280,477 wr)
==24611== D1 misses:    1,731 ( 1,240 rd +   491 wr)
==24611== LLd misses:   1,541 ( 1,071 rd +   470 wr)
==24611== D1 miss rate:  0.2% (  0.2% +   0.2% )
==24611== LLd miss rate: 0.2% (  0.2% +   0.2% )
==24611==
==24611== LL refs:      3,205 ( 2,714 rd +   491 wr)
==24611== LL misses:    2,959 ( 2,489 rd +   470 wr)
==24611== LL miss rate:  0.1% (  0.1% +   0.2% )
```

Figura 24. Valgrind do Quick sort (Config padrão)

```
==26203==
==26203== I refs:      2,018,684
==26203== I1 misses:    1,725
==26203== LLi misses:   1,418
==26203== I1 miss rate:  0.09%
==26203== LLi miss rate: 0.07%
==26203==
==26203== D refs:      816,527 (536,050 rd + 280,477 wr)
==26203== D1 misses:    2,076 ( 1,539 rd +   537 wr)
==26203== LLd misses:   1,541 ( 1,071 rd +   470 wr)
==26203== D1 miss rate:  0.3% (  0.3% +   0.2% )
==26203== LLd miss rate: 0.2% (  0.2% +   0.2% )
==26203==
==26203== LL refs:      3,801 ( 3,264 rd +   537 wr)
==26203== LL misses:    2,959 ( 2,489 rd +   470 wr)
==26203== LL miss rate:  0.1% (  0.1% +   0.2% )
```

Figura 25. Valgrind do Quick sort (Config simulada)

4.5. Selection Sort

4.5.1. *Perf*:

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

        62.888      cache-references      #    9,456 M/sec
        29.533      cache-misses          #    46,96% of all cache refs
         6,65 msec  task-clock             #    0,002 CPUs utilized
       7.041.237     cycles                 #    1,059 GHz
      12.019.089     instructions           #    1,71 insn per cycle

4,420984145 seconds time elapsed

0,004926000 seconds user
0,002889000 seconds sys
```

Figura 26. Perf do Selection sort para a entrada de dados pequena

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

       71.300      cache-references      #    6,643 M/sec
       27.484      cache-misses          #    38,55% of all cache refs
       10,73 msec  task-clock             #    0,002 CPUs utilized
      10.895.175     cycles                 #    1,015 GHz
      22.556.209     instructions           #    2,07 insn per cycle

6,427932193 seconds time elapsed

0,008702000 seconds user
0,002926000 seconds sys
```

Figura 27. Perf do Selection sort para a entrada de dados grande

4.5.2. Valgrind:

```
==25315==
==25315== I refs:          10,793,429
==25315== I1 misses:       1,472
==25315== LLi misses:      1,415
==25315== I1 miss rate:    0.01%
==25315== LLi miss rate:   0.01%
==25315==
==25315== D refs:          5,192,876 (4,916,981 rd + 275,895 wr)
==25315== D1 misses:       1,731 ( 1,240 rd + 491 wr)
==25315== LLd misses:      1,541 ( 1,071 rd + 470 wr)
==25315== D1 miss rate:    0.0% ( 0.0% + 0.2% )
==25315== LLd miss rate:   0.0% ( 0.0% + 0.2% )
==25315==
==25315== LL refs:         3,203 ( 2,712 rd + 491 wr)
==25315== LL misses:       2,956 ( 2,486 rd + 470 wr)
==25315== LL miss rate:    0.0% ( 0.0% + 0.2% )
```

Figura 28. Valgrind do Selection sort (Config padrão)

```
==26556==
==26556== I refs:          10,793,429
==26556== I1 misses:       1,721
==26556== LLi misses:      1,415
==26556== I1 miss rate:    0.02%
==26556== LLi miss rate:   0.01%
==26556==
==26556== D refs:          5,192,876 (4,916,981 rd + 275,895 wr)
==26556== D1 misses:       2,075 ( 1,538 rd + 537 wr)
==26556== LLd misses:      1,541 ( 1,071 rd + 470 wr)
==26556== D1 miss rate:    0.0% ( 0.0% + 0.2% )
==26556== LLd miss rate:   0.0% ( 0.0% + 0.2% )
==26556==
==26556== LL refs:         3,796 ( 3,259 rd + 537 wr)
==26556== LL misses:       2,956 ( 2,486 rd + 470 wr)
==26556== LL miss rate:    0.0% ( 0.0% + 0.2% )
```

Figura 29. Valgrind do Selection sort (Config simulada)

5. Comparações

5.1. Perf's

5.1.1. Multiplicação de Matriz por Escalar

Ao analisarmos e compararmos os resultados dos testes fornecidos pela ferramenta Perf, podemos perceber que, como falado e explicado pelo professor da disciplina, Nacif, operações que são feitas linha por linha de uma matriz vão ter um melhor desempenho

comparadas à mesma operação executada coluna por coluna, devido ao funcionamento padrão de acesso de uma memória cache.

É possível observar tal afirmação ao olharmos para a quantidade de vezes que o processador tentou acessar a cache (*cache references*) a quantidade de vezes que a informação desejada não estava na memória cache (*cache misses*), quantidade de ciclos (*cycles*) necessários para realizar a tarefa e a quantidade de instruções (*instructions*) necessárias para realizar a tarefa.

Todas essas informações citadas nas operações de multiplicação de matriz por um escalar linha por linha têm os menores valores. Ou seja, são mais eficientes que a mesma operação realizada coluna por coluna. A diferença ao comparar com os dados da matriz menor não é tão grande, embora evidente. Observarmos essa diferença com mais clareza ao compararmos os dados da matriz maior, na qual as diferenças são maiores e mais notáveis.

5.1.2. Sort's

Quando comparamos os códigos sorts percebemos que há uma melhora no desempenho da memória cache no conjunto de dados maior, provavelmente consequência de haverem mais números repetidos, já que há um intervalo de números que são preenchidos no vetor.

Embora tendo sendo necessário mais tempo para executar o conjunto de dados maior (todos os algoritmos) sua eficiência na utilização da memória cache é visível no conjunto de dados maior. Outra coisa possível de notar é que o escalonamento da quantidade de instruções realizadas é diferente para cada algoritmo.

Mesmo que o conjunto de dados maior seja x vezes maior que o conjunto de dados menor, o número de instruções não é multiplicado por 10x ao compararmos a quantidade de instruções entre a tarefa realizada com o conjunto de dados menor e a realizada com o conjunto de dados maior. Sendo o que menos escala o que menos aumenta a quantidade de instruções ao realizar a operação com o conjunto de dados maior o Selection Sort.

Além disso, é possível perceber a eficiência dos métodos de ordenação ao compará-los, pois notamos que, no geral, o tempo necessário para o Quick sort terminar a tarefa é menor que todos os outros, tanto nos testes com o conjunto de dados menor quanto no conjunto de dados maior.

5.2. Valgrind

Utilizando o Valgrind podemos notar que, para a execução de todos os algoritmos de ordenação usados neste trabalho (estudo), a configuração (arquitetura) de memória cache do computador usado para os testes é mais eficiente que a arquitetura que foi simulada. A cache simulada está detalhada no segundo parágrafo dos resultados.

Essa diferença de desempenho reforça a importância de uma cache bem dimensionada em sistemas que executam algoritmos intensivos, destacando a relevância de um bom balanceamento entre tamanho de cache, associatividade, e a natureza das tarefas executadas.

6. Conclusão

Após muitos testes e análises, podemos concluir que o objetivo proposto pelo trabalho foi concluído com sucesso. Conseguimos observar, relatar e entender os diferentes desempenhos da memória cache, seja por ser um algoritmo diferente ou pela quantidade de dados tratados ser diferente.

Conseguimos também mostrar que uma mesma tarefa pode ter um melhor desempenho de acesso a memória cache dependendo da lógica de programação implementada para realizar essa tarefa. Além disso, conseguimos desenvolver uma mesma tarefa com diferentes lógicas de programação, e foi possível comprovar que uma operação realizada em uma matriz possui melhor desempenho se a operação for realizada na ordem de linha a linha, ao invés de coluna a coluna.

7. Referências

Apenas os códigos de ordenação foram retirados da internet, o restante dos códigos utilizados para executar o trabalho foram de autoria da própria dupla.

Repositório do Trabalho: Git Hub

Código do Bubble Sort

Código do Quick Sort

Código do Radix Sort

Código do Selection Sort

Aprender a utilizar do Perf

Aprender a utilizar do Valgrind