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

Gabriel Miranda(3857)[1], Felipe Dias(3888)[1], Mariana de Souza(3898)[1]

[1]Instituto de Ciências Exatas e Tecnológicas,
Universidade Federal de Viçosa, Florestal, MG, Brasil

1. Descrição

Neste trabalho prático, foi feito um estudo da utilização da memória cache em algoritmos de ordenação, todavia também foi analisado como podíamos melhorar o uso da memória cache para um algoritmo de Fibonacci. Dessa forma, é possível ressaltar alguns algoritmos que já foram pré-selecionados para essa análise, eles são: **Bubble Sort**, **Radix Sort** e **Quick Sort**, também foi decidido que o grupo escolhesse outro algoritmo para fazer essa análise, o algoritmo escolhido foi o **Merge Sort.** Vale ressaltar também as ferramentas que foram ultilizadas para essa análise, usamos a ferramenta Perf no ubuntu e o software de simulação de cache Valgrind.

2. Introdução

O quarto trabalho prático tem como principal propósito analisar o comportamento dos algoritmos de ordenação na cache do computador utilizado. Nesse sentido, com essa análise feita, é possível realizar melhorias nesses algoritmos, todavia também realizar testes em diversas configurações da memória cache.

Outrossim, foram apresentados resultados divesos para os algoritmos de ordenação e o algoritmo escolhido pelo grupo, o Fibonacci. Nisso, mostramos a quantidade de cache utilizada pelos algoritmos. É possível salientar a otimização feita pelo grupo do Fibonacci que gerou melhores resultados do que o Fibonacci não otimizado.

3. Especificações do computador e sua cache:

- **Memória RAM:** 8 GB DDR4
- **Processador:** AMD Ryzen 3; 4 cores; 4 threads; 4x3600,00 MHz
- **Sistema Operacional:** Pop! OS system 76 Linux
- **Tamanho da Cache Nível 1 (Data):** 4x 32KB, 8-way set-associative, 64 sets
- **Tamanho da Cache Nível 1(Instruction):**4x64KB,4-way set-associative,256 sets
- **Tamanho da Cache Nível 2:** 4x 512, 8-way set-associative, 1024 sets
- **Tamanho da Cache Nível 3:** 1x 4096KB, 16-way set-associative, 4096 sets

4. Algoritmos de ordenação e suas especificações

4.1. Bubble Sort

A ideia base desse algoritmo é que a lista não ordenada seja atribuída a um vetor. Então os números são ordenados por um procedimento que, essencialmente, passa pelo vetor muitas vezes, permutando elementos consecutivos que estão na ordem errada, até

que todos os elementos estejam na ordem correta. Esse procedimento é chamado Bubble Sort, pois os números menores sobem para o topo da lista, como bolhas de ar na água, mas, na verdade, o maior número irá ”afundar” na parte inferior da lista após a primeira passagem, o que o classifica como ”bolha pesada”. Um ponto interessante é que de longe o Bubble é o algoritmo mais fácil de ser implementado, porém se classifica entre os piores quando se trata de ordenação. (Exemplo do Bubble Sort).

4.2. Radix Sort

O Radix Sort é um algoritmo da classe de ordenação de inteiros, e não de comparação entre os elementos de entrada, como é o caso do algoritmo Merge Sort. Suponha n elementos a serem ordenados, cada um representado por uma chave. Algoritmos como o Counting Sort são muito eficientes para ordenação quando esta chave é representada por inteiros relativamente pequenos. Porém, quando essas chaves crescem em tamanho, pode ser interessante ”quebrar” a chave em n pedaços, que chamamos de dígitos, e realizar a ordenação por etapas (dígito a dígito). É essa a ideia principal do Radix Sort.

A representação de cada dígito depende da base escolhida. Por exemplo, podemos utilizar base decimal, binária, hexadecimal, etc. A escolha da base determina os k diferentes valores que um dígito pode assumir (e Radix significa exatamente isso, a base em sistemas de numeração posicional). Apesar de ser da classe de algoritmos de ordenação de inteiros, o Radix Sort pode ser utilizado para ordenação de diversos outros tipos de dados. Por exemplo, para strings: basta, para tal, utilizarmos uma base em que o ’dígito’ represente adequadamente a ordenação desejada para cada caractere.

O Radix Sort pode ser implementado em duas variantes principais: Radix Sort LSD (Least Significant Digit) e o Radix Sort MSD (Most Significant Digit). As versões básicas de cada algoritmo demandam espaço extra de memória durante o processo de ordenação. Assim, como alternativa, será abordada também uma versão ”in-place” do algoritmo MSD. (Exemplo do Radix Sort no Github).

4.3. Quick Sort

Quick Sort é um algoritmo eficiente de ordenação por divisão e conquista. Apesar de ser da mesma classe de complexidade do Merge Sort e do Heap Sort, o Quick Sort é, na prática, o mais veloz deles, pois suas constantes são menores. Contudo, é importante destacar de antemão que, em seu pior caso, o Quick Sort é O(n*n), enquanto que o Merge Sort e o Heap Sort garantem (nlogn) para todos os casos. A boa notícia é que há estratégias simples com as quais podemos minimizar as chances de ocorrência do pior caso.

O funcionamento do Quick Sort baseia-se em uma rotina fundamental cujo nome é particionamento. Particionar significa escolher um número qualquer presente no array, chamado de pivot, e colocá-lo em uma posição tal que todos os elementos a esquerda são menores ou iguais e todos os elementos a direita são maiores. (Exemplo do Quick Sort).

4.4. Merge Sort

O Merge Sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Sua ideia básica é muito fácil: criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original

em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes.

Os três passos úteis dos algoritmos dividir-para-conquistar, ou divide and conquer, que se aplicam ao Merge Sort são:
Dividir: Dividir os dados em subsequências pequenas;
Conquistar: Classificar as duas metades recursivamente aplicando o Merge Sort;
Combinar: Juntar as duas metades em um único conjunto já classificado.

As vantagens do Merge Sort é que esse algoritmo de ordenação é de simples implementação e de fácil entendimento utilizando chamadas recursivas. (Exemplo do Merge Sort).

4.5. Fibonacci

É uma sucessão de números que seguem um padrão, a sequência de Fibonacci nada mais é que a ordem de números inteiros que parte, normalmente, de zero e um no qual cada número seguinte corresponde a soma dos dois algarismos anteriores. Essa continuidade pode ser vista em vários fenômenos da natureza.

A sequência de Fibonacci foi nomeada pelo matemático italiano Leonardo de Pisa, também conhecido como Fibonacci. Em 1202, a partir dessa sequência numérica, o matemático relatou o avanço de uma população de coelhos.

A sucessão de Fibonacci é uma sequência de números inteiros iniciados por zero e um, no qual cada termo subsequente corresponde a soma dos dois números anteriores: 0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584...

Em termos matemáticos, a sequência é definida pela fórmula Fn = fn-1 + Fn-2, sendo o primeiro termo F1 = 1 e os valores iniciais F1 = 1, F2 =1. Esse método é aplicado na análise de mercados financeiros, na teoria de jogos e na ciência da computação, além de configurações biológicas e naturais.

5. Ultilização da ferramenta Perf para medição de desempenho dos algoritmos

5.1. Bubble Sort

A complexidade do Bubble sort faz com que a quantidade de instruções cresça consideravelmente. Quanto maior a sequência sendo ordenada, maior é a taxa de hits e, consequentemente, menos ciclos de clock são gastos, pois são necessários menos ciclos adicionais para acessar os outros níveis da hierarquia de memoria em busca de dados.

Mesmo que gaste pouco menos ciclos de clock por ter uma boa taxa de hits para entradas maiores, isso não causa quase nenhum impacto em seu desempenho geral.

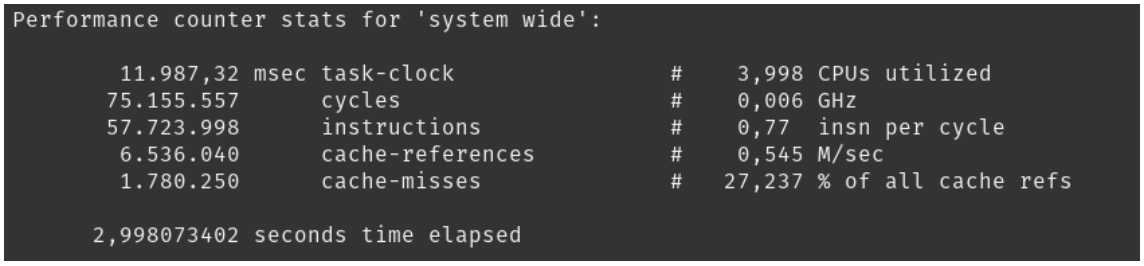

Figura 1. Bubble Sort vetor 1.000 elementos.

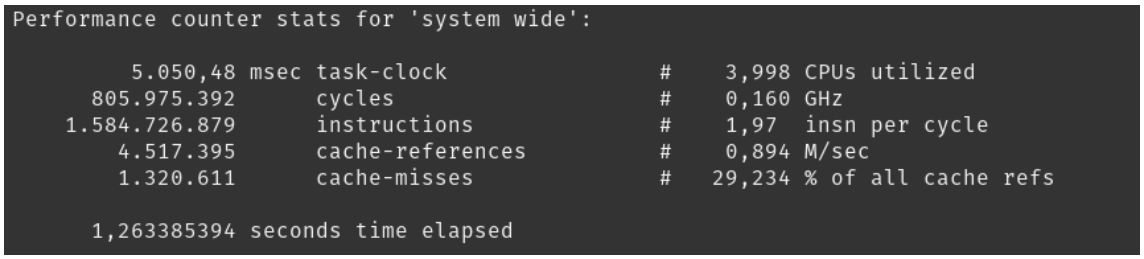

Figura 2. Bubble Sort vetor 10.000 elementos.

5.2. Quick Sort

Para tamanhos de entrada extremamente grandes, no quesito aproveitamento na cache, esse algoritmo demonstra ser extremamente eficiente, pois, já que a divisão do vetor de itens faz parte do seu procedimento de ordenação, as partes da divisão passam a ocupar tamanhos menores ao da cache. Isso faz com que em determinado momento, quando a ordenação passa a ser em outro sub-conjunto de elementos resultantes do vetor pai, todas as informações necessárias para essa tarefa estejam na cache.

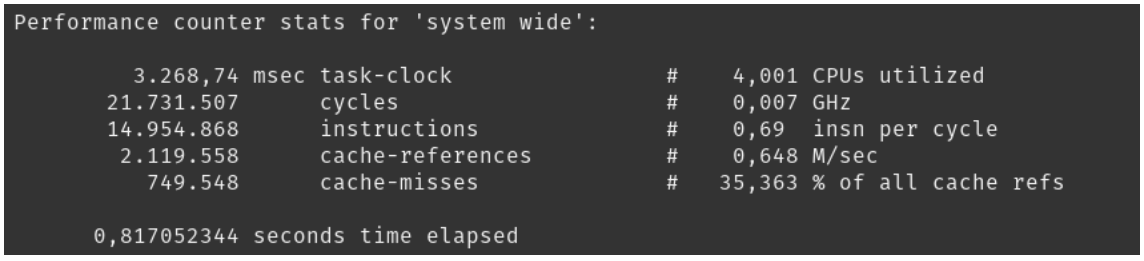

Figura 3. Quick Sort vetor 1.000 elementos.

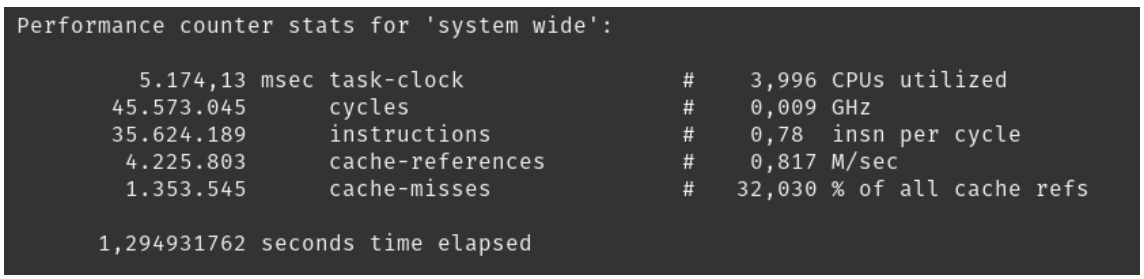

Figura 4. Quick Sort vetor 10.000 elementos.

5.3. Radix Sort

A maior fraqueza do Radix sort são suas referências a memória, que inevitavelmente dão misses na cache em grandes quantidades, principalmente com um array de entrada muito grande. Isso se deve as pilhas, que são armazenadas de forma aleatória, muitas vezes distribuídas pessimamente, na memória principal, mesmo os vetores em si ficando ordenados, ferindo bastante o princípio da localidade.

```
Performance counter stats for 'system wide':

         2.051,57 msec task-clock                #    3,996 CPUs utilized
       34.242.393      cycles                    #    0,017 GHz
       27.967.621      instructions              #    0,82  insn per cycle
        3.259.061      cache-references          #    1,589 M/sec
        1.049.078      cache-misses              #   32,190 % of all cache refs

      0,513359965 seconds time elapsed
```

Figura 5. Radix Sort vetor 1.000 elementos.

```
Performance counter stats for 'system wide':

         4.118,90 msec task-clock                #    3,987 CPUs utilized
       43.820.785      cycles                    #    0,011 GHz
       34.803.742      instructions              #    0,79  insn per cycle
        4.280.726      cache-references          #    1,039 M/sec
        1.341.078      cache-misses              #   31,328 % of all cache refs

      1,033046805 seconds time elapsed
```

Figura 6. Radix Sort vetor 10.000 elementos.

5.4. Merge Sort

Uma classifição Merge sort ascendente irá gerar execuções classificadas com tamanhos de potências de 2, o que pode ser um pouco mais amigável a cache. As vantagens do Merge sort ainda se mostram ao final da fase de divisão, onde grande parte dos dados necessários para a fase de combinação já estão disponíveis na cache. Com isso, requerem um número menor de acessos á memória.

```
Performance counter stats for 'system wide':

         3.708,74 msec task-clock                #    3,994 CPUs utilized
       30.184.898      cycles                    #    0,008 GHz
       24.526.500      instructions              #    0,81  insn per cycle
        3.009.665      cache-references          #    0,812 M/sec
        1.025.917      cache-misses              #   34,087 % of all cache refs

      0,928489836 seconds time elapsed
```

Figura 7. Merge Sort vetor 1.000 elementos.

```
Performance counter stats for 'system wide':

         5.697,47 msec task-clock                #    4,000 CPUs utilized
        58.682.631      cycles                   #    0,010 GHz
        52.991.977      instructions             #    0,90  insn per cycle
         4.825.301      cache-references         #    0,847 M/sec
         1.459.302      cache-misses             #   30,243 % of all cache refs

      1,424288984 seconds time elapsed
```

Figura 8. Merge Sort vetor 10.000 elementos.

6. Simulação Valgrind

Simulação com o Valgrind, analisando os algoritmos Quick Sort, Bubble Sort, Merge Sort e Radix Sort, para um vetor de ordenação de 1.000 elementos. Analisando o Quick Sort com o tamanho de cache de 256, 512 e 1024 bits, e para a associatividade 2, 4 e 8, com o tamanho do bloco de palavras de 128 bytes.

6.1. Quick Sort

Tamanho de cache: o algoritmo melhora muito conforme e aumentado a cache, pois evita, na divisão, a perda de dados da primeira parte do vetor pai em vetores filhos.
Associatividade: quanto mais superior, melhor o algoritmo fica, pois, assim como o tamanho de cache, tem potencial para acabar com a perda dos dados da parte A do vetor pai durante a formação de um vetor B.
Tamanho do bloco: faz bem para o algoritmo se aumentar o tamanho do bloco, porém a influência disso no desempenho é muito menor do que se aumentar associatividade ou a cache.

```
==12959== I   refs:      1,339,816
==12959== I1  misses:        1,419
==12959== LLi misses:        1,409
==12959== I1  miss rate:      0.11%
==12959== LLi miss rate:      0.11%
==12959==
==12959== D   refs:        537,777  (386,838 rd   + 150,939 wr)
==12959== D1  misses:      224,332  (175,151 rd   +  49,181 wr)
==12959== LLd misses:        2,480  (  1,938 rd   +     542 wr)
==12959== D1  miss rate:      41.7% (   45.3%     +    32.6%  )
==12959== LLd miss rate:       0.5% (    0.5%     +     0.4%  )
==12959==
==12959== LL refs:         225,751  (176,570 rd   +  49,181 wr)
==12959== LL misses:         3,889  (  3,347 rd   +     542 wr)
==12959== LL miss rate:        0.2% (    0.2%     +     0.4%  )
```

Figura 13. Associatividade 2, tamanho de 256 bits.

```
==13577== I   refs:      1,339,816
==13577== I1  misses:        1,419
==13577== LLi misses:        1,409
==13577== I1  miss rate:      0.11%
==13577== LLi miss rate:      0.11%
==13577==
==13577== D   refs:        537,777  (386,838 rd   + 150,939 wr)
==13577== D1  misses:      116,974  ( 86,807 rd   +  30,167 wr)
==13577== LLd misses:        2,248  (  1,789 rd   +     459 wr)
==13577== D1  miss rate:      21.8% (   22.4%     +    20.0%  )
==13577== LLd miss rate:       0.4% (    0.5%     +     0.3%  )
==13577==
==13577== LL refs:         118,393  ( 88,226 rd   +  30,167 wr)
==13577== LL misses:         3,657  (  3,198 rd   +     459 wr)
==13577== LL miss rate:        0.2% (    0.2%     +     0.3%  )
```

Figura 14. Associatividade 4, tamanho de 512 bits.

```
==13648== I   refs:      1,339,797
==13648== I1  misses:        1,416
==13648== LLi misses:        1,406
==13648== I1  miss rate:      0.11%
==13648== LLi miss rate:      0.10%
==13648==
==13648== D   refs:        537,772  (386,836 rd   + 150,936 wr)
==13648== D1  misses:       74,563  ( 57,465 rd   +  17,098 wr)
==13648== LLd misses:        1,916  (  1,471 rd   +     445 wr)
==13648== D1  miss rate:      13.9% (    14.9%     +    11.3%  )
==13648== LLd miss rate:       0.4% (     0.4%     +     0.3%  )
==13648==
==13648== LL refs:          75,979  ( 58,881 rd   +  17,098 wr)
==13648== LL misses:         3,322  (  2,877 rd   +     445 wr)
==13648== LL miss rate:        0.2% (     0.2%     +     0.3%  )
```

Figura 15. Associatividade 8, tamanho de 1024 bits.

6.2. Bubble Sort

Para os demais algoritmos de ordenação foram utilizados o tamanho de cache de 1024 bits, associatividade 2, 4 e 8, tamanho de bloco de palavras de 128 bytes.
Tamanho de cache: se o futuro vetor ordenado em questao couber na cache, já serve. Seu tamanho não tem tanto impacto em desempenho.
Associatividade: também não faz muita diferença para o Bubble sort, uma vez que se trata somente de um vetor contíguo para ordenar.
Tamanho do bloco: se aumentado o tamanho da memória cache, é causada a diminuição do miss rate, já que, um por um, o algoritmo percorre todo o vetor em processo diversas vezes em ordem crescente.

```
==3719== I   refs:      16,715,299
==3719== I1  misses:         1,444
==3719== LLi misses:         1,433
==3719== I1  miss rate:       0.01%
==3719== LLi miss rate:       0.01%
==3719==
==3719== D   refs:       8,465,748  (7,568,692 rd   + 897,056 wr)
==3719== D1  misses:       203,008  (  167,040 rd   +  35,968 wr)
==3719== LLd misses:         2,060  (    1,618 rd   +     442 wr)
==3719== D1  miss rate:        2.4% (      2.2%     +     4.0%  )
==3719== LLd miss rate:        0.0% (      0.0%     +     0.0%  )
==3719==
==3719== LL refs:          204,452  (  168,484 rd   +  35,968 wr)
==3719== LL misses:          3,493  (    3,051 rd   +     442 wr)
==3719== LL miss rate:         0.0% (      0.0%     +     0.0%  )
```

Figura 16. Bubble Sort para Associatividade 2.

```
==3787== I   refs:      16,715,309
==3787== I1  misses:         1,444
==3787== LLi misses:         1,433
==3787== I1  miss rate:       0.01%
==3787== LLi miss rate:       0.01%
==3787==
==3787== D   refs:       8,465,752  (7,568,693 rd   + 897,059 wr)
==3787== D1  misses:        88,002  (   74,085 rd   +  13,917 wr)
==3787== LLd misses:         1,969  (    1,528 rd   +     441 wr)
==3787== D1  miss rate:        1.0% (      1.0%     +     1.6%  )
==3787== LLd miss rate:        0.0% (      0.0%     +     0.0%  )
==3787==
==3787== LL refs:            89,446  (   75,529 rd   +  13,917 wr)
==3787== LL misses:          3,402  (    2,961 rd   +     441 wr)
==3787== LL miss rate:         0.0% (      0.0%     +     0.0%  )
```

Figura 17. Bubble Sort para Associatividade 4.

```
==3818== I   refs:      16,715,266
==3818== I1  misses:         1,442
==3818== LLi misses:         1,431
==3818== I1  miss rate:       0.01%
==3818== LLi miss rate:       0.01%
==3818==
==3818== D   refs:       8,465,734  (7,568,685 rd   + 897,049 wr)
==3818== D1  misses:        87,249  (   75,489 rd   +  11,760 wr)
==3818== LLd misses:         1,922  (    1,479 rd   +     443 wr)
==3818== D1  miss rate:        1.0% (      1.0%     +     1.3%  )
==3818== LLd miss rate:        0.0% (      0.0%     +     0.0%  )
==3818==
==3818== LL refs:           88,691  (   76,931 rd   +  11,760 wr)
==3818== LL misses:          3,353  (    2,910 rd   +     443 wr)
==3818== LL miss rate:         0.0% (      0.0%     +     0.0%  )
```

Figura 18. Bubble Sort para Associatividade 8.

6.3. Merge Sort

Tamanho de cache: devido a menor necessidade de solicitações de leitura/escrita, a carga injetada diminui com o aumento do tamanho da cache. Pode-se ver que tamanhos maiores de cache resultam em execução mais eficiente do algoritmo, quando considerados apenas os parâmetros de desempenho, ou seja, tempo de resposta de opereções de leitura e escrita.
Associatividade: grande associatividade garante menos perda de dados, garantindo uma maior taxa de acertos que, por sua vez, otimiza em desempenho.
Tamanho do bloco: tamanhos de bloco maiores providenciam sim uma melhoria no algoritmo, porém mínima se comparada ás que são proporcionadas pelo tamanho da cache ou pela associatividade.

```
==3953== I   refs:      1,717,384
==3953== I1  misses:        1,401
==3953== LLi misses:        1,392
==3953== I1  miss rate:      0.08%
==3953== LLi miss rate:      0.08%
==3953==
==3953== D   refs:        746,965  (560,666 rd   + 186,299 wr)
==3953== D1  misses:       80,494  ( 65,575 rd   +  14,919 wr)
==3953== LLd misses:        2,056  (  1,610 rd   +     446 wr)
==3953== D1  miss rate:      10.8% (   11.7%     +     8.0%  )
==3953== LLd miss rate:       0.3% (    0.3%     +     0.2%  )
==3953==
==3953== LL refs:          81,895  ( 66,976 rd   +  14,919 wr)
==3953== LL misses:         3,448  (  3,002 rd   +     446 wr)
==3953== LL miss rate:        0.1% (    0.1%     +     0.2%  )
```

Figura 19. Merge Sort para Associatividade 2.

```
==3992== I   refs:      1,717,384
==3992== I1  misses:        1,401
==3992== LLi misses:        1,392
==3992== I1  miss rate:      0.08%
==3992== LLi miss rate:      0.08%
==3992==
==3992== D   refs:        746,965  (560,666 rd   + 186,299 wr)
==3992== D1  misses:       73,377  ( 59,189 rd   +  14,188 wr)
==3992== LLd misses:        1,965  (  1,520 rd   +     445 wr)
==3992== D1  miss rate:       9.8% (   10.6%     +     7.6%  )
==3992== LLd miss rate:       0.3% (    0.3%     +     0.2%  )
==3992==
==3992== LL refs:          74,778  ( 60,590 rd   +  14,188 wr)
==3992== LL misses:         3,357  (  2,912 rd   +     445 wr)
==3992== LL miss rate:        0.1% (    0.1%     +     0.2%  )
```

Figura 20. Merge Sort para Associatividade 4.

```
==4021== I   refs:      1,717,384
==4021== I1  misses:        1,401
==4021== LLi misses:        1,392
==4021== I1  miss rate:      0.08%
==4021== LLi miss rate:      0.08%
==4021==
==4021== D   refs:        746,965  (560,666 rd   + 186,299 wr)
==4021== D1  misses:       72,429  ( 60,418 rd   +  12,011 wr)
==4021== LLd misses:        1,918  (  1,471 rd   +     447 wr)
==4021== D1  miss rate:       9.7% (   10.8%     +     6.4%  )
==4021== LLd miss rate:       0.3% (    0.3%     +     0.2%  )
==4021==
==4021== LL refs:          73,830  ( 61,819 rd   +  12,011 wr)
==4021== LL misses:         3,310  (  2,863 rd   +     447 wr)
==4021== LL miss rate:        0.1% (    0.1%     +     0.2%  )
```

Figura 21. Merge Sort para Associatividade 8.

6.4. Radix Sort

Tamanho de cache: é inútil em relação a utilização da cache por dissipar diversos vetores, para ordenação dos elementos, na memória principal.
Associatividade: quanto maior associatividade, mais hits, ja que existem vetores distribuídos na cache guardados na memória principal, e pelo Radix Sort não seguir o princípio da localidade temporal.
Tamanho do bloco: por justamente não seguir o princípio da localidade temporal, o miss rate e atrapalhado ao se aumentar o tamanho do bloco.

```
==3914== I   refs:      1,690,878
==3914== I1  misses:        1,401
==3914== LLi misses:        1,392
==3914== I1  miss rate:      0.08%
==3914== LLi miss rate:      0.08%
==3914==
==3914== D   refs:        676,842  (513,416 rd   + 163,426 wr)
==3914== D1  misses:       89,571  ( 72,510 rd   +  17,061 wr)
==3914== LLd misses:        2,085  (  1,610 rd   +     475 wr)
==3914== D1  miss rate:      13.2% (   14.1%     +    10.4%  )
==3914== LLd miss rate:       0.3% (    0.3%     +     0.3%  )
==3914==
==3914== LL refs:          90,972  ( 73,911 rd   +  17,061 wr)
==3914== LL misses:         3,477  (  3,002 rd   +     475 wr)
==3914== LL miss rate:        0.1% (    0.1%     +     0.3%  )
```

Figura 22. Radix Sort para Associatividade 2.

```
==3885== I   refs:      1,690,878
==3885== I1  misses:        1,401
==3885== LLi misses:        1,392
==3885== I1  miss rate:      0.08%
==3885== LLi miss rate:      0.08%
==3885==
==3885== D   refs:        676,842  (513,416 rd   + 163,426 wr)
==3885== D1  misses:       76,621  ( 59,453 rd   +  17,168 wr)
==3885== LLd misses:        1,998  (  1,520 rd   +     478 wr)
==3885== D1  miss rate:      11.3% (   11.6%     +    10.5%  )
==3885== LLd miss rate:       0.3% (    0.3%     +     0.3%  )
==3885==
==3885== LL refs:          78,022  ( 60,854 rd   +  17,168 wr)
==3885== LL misses:         3,390  (  2,912 rd   +     478 wr)
==3885== LL miss rate:        0.1% (    0.1%     +     0.3%  )
```

Figura 23. Radix Sort para Associatividade 4.

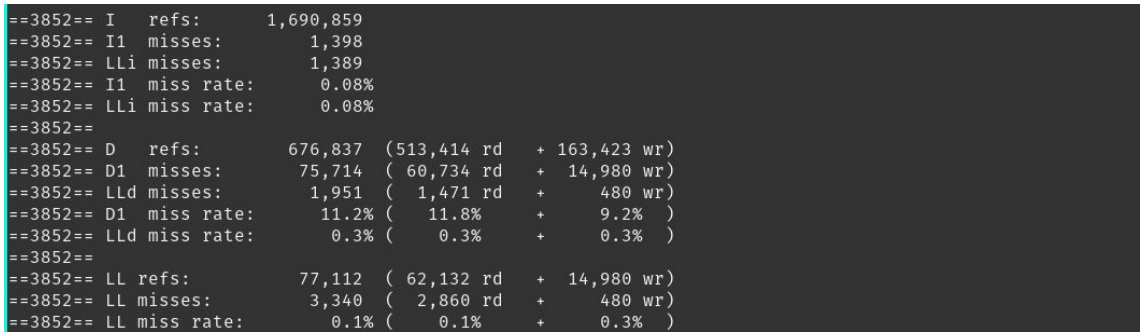

Figura 24. Radix Sort para Associatividade 8.

7. Algoritmo Fibonacci

7.1. Fibonacci recursivo

Nesta seção, iremos demonstrar o que foi feito para a otimização do Fibonacci recursivo. Primeiramente, falaremos sobre como o algoritmo Fibonacci recursivo pode ser ineficiente. Nesse sentido, analisamos como a sequência de Fibonacci gera números que crescem muito rapidamente. Com isso, como o algoritmo é recursivo, cada chamada recursiva vai criar outras chamadas recursivas que, por sua vez, criam outras chamadas recursivas e assim por diante. No final de cada chamada recursiva e produzido um resultado 0 ou 1. Logo, é possível perceber que com o passar do tempo, o Fibonacci recursivo não vai mostrar mais os resultados da sequência.

```
int fib(int n){
    if (n == 1) return 1;
    else
      if (n == 2) return 1;
      else return fib(n - 1) + fib(n - 2);
}
```

Figura 25. Observe que, da forma como está escrito, cada chamada da função irá chamar outras duas funções, o que pode ser um problema.

7.2. Fibonacci otimizado

Foram analisados alguns fatores no código e percebemos que um algoritmo iterativo seria mais relevante de ser ultilizado, pois basicamente ele possui a estrutura de repetição, como o for. Esta seria a forma mais “normal” de se programar um comportamento repetitivo. Neste sentido, conseguimos obter um melhor desempenho do algoritmo iterativo em relação a memória.

```
int fib(int n){

  int i, fib1 = 1, fib2 = 1, soma;
  for (i = 3; i <= n; i = i + 1){

    soma = fib1 + fib2;
    fib1 = fib2;
    fib2 = soma;

  }
     return fib2;
}
```

Figura 26. A primeira vista, parece que ficou mais robusto e longo em relação ao recursivo, mas se mostrou melhor na análise.

8. Conclusão

Neste último trabalho, foram utilizados quatro algoritmos de ordenação e o algoritmo Fibonacci para serem analisados pela ferramenta Perf e Valgrind no ubuntu, o intuito inicial do trabalho foi analisar e descrever esses algoritmos para uma melhor compreensão da utilização de memória. Diante disso, foi perceptível a importância de se extrair o máximo da estrutura de memória de um computador e como a mal otimização de um algoritmo pode tornar o processo de compilação lento. Nesse sentido, é notório que todo o profissional de TI tenha uma boa carga, como boas práticas de programação e saiba otimizar bem os algoritmos e também utilizar bem a cache do computador que será usado para fins de progamação.

Portanto, indubitavelmente, é observável que um bom uso da memória cache pelos algoritmos pode acelerar a execução dos códigos para problemas maiores. Destarte, e possível salientar os aumentos e quedas dos numeros de instruções e misses que indicam como a memória cache está lidando com os valores essenciais para a execução dos algoritmos. Logo, o acréscimo desses algoritmos devem ser refletidos tanto no quesito complexidade, quanto no quesito de uso de memória cache, e que a austeridade desses elementos tragam consigo um melhor resultado obtido no processo de compilação.

9. Referências:

• Perf
• Valgrind
• Instalação Linux Valgrind
• Comandos Perf Ubuntu
• Livro Computer Organization and Design RISC-V Edition: The Hardware Software Interface (The Morgan Kaufmann Series)
• Exemplo do Merge Sort

- Exemplo do Quick Sort
- Exemplo do Radixsort no Github
- Exemplo do Bubble Sort
- Especificação do Bubble Sort
- Especificação do Radix Sort
- Especificação do Quick Sort
- Especificação do Merge Sort
- Especificação do Fibonacci