UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL - UFRGS
CIÊNCIA DA COMPUTAÇÃO

PROFILLING E ANÁLISE DO COMPORTAMENTO DA HIERARQUIA DE MEMÓRIA COM DIFERENTES ESTRUTURAS DE DADOS

INF01112 - ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES II
TURMA C

Marcelo Basso
00579239

Porto Alegre - RS
2024/01

SUMÁRIO

INTRODUÇÃO

O projeto implementado [1] visa comparar o desempenho de 3 aplicações no que se refere ao acesso à hierarquia de memória na arquitetura x86_64 com o processador Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz. As aplicações analisadas consistem em:

1. Uma codificação estável do Quicksort para ordenação vetorial;
2. Um algoritmo simples de multiplicação matricial;
3. DFS - *depth first search* - para caminhamento em grafo.

Sobre esses algoritmos, com diferentes tamanhos de entrada, coletou-se uma série de métricas que possibilitam analisar os acessos e o funcionamento da hierarquia de memória na prática, e o desempenho das aplicações no geral. Essas métricas são:

1. Ciclos;
2. Instruções executadas;
3. Acessos à memória cache (*hits* e *misses* na L1, L2 e L3);
4. Acessos à TLB (*misses* e *hits*);
5. Acessos à memória principal.

As métricas coletadas serão melhor discriminadas no capítulo seguinte.

A partir da coleta desses dados, delineou-se uma análise de acessos à cache e à memória principal conforme o comportamento e a implementação de cada aplicação, e após, realizou-se uma análise do porquê as aplicações tiveram tais comportamentos.

[1] O projeto completo pode ser visualizado no link <https://github.com/marcelobasso/INF01112-arch-2/tree/main/cache-tests>

DESCRIÇÃO GERAL DE IMPLEMENTAÇÃO

Para todos os algoritmos, antes da execução do programa principal (como ordenação do vetor ou multiplicação da matriz), as estruturas são populadas com dados aleatórios. Tanto o algoritmo de ordenação quanto o algoritmo de matrizes utiliza inteiros para os testes, com tamanho de 4 bytes. O algoritmo de caminhamento em grafo utiliza nodos com os seguintes dados: um número inteiro, uma string aleatória de tamanho 10 e dois ponteiros para os nodos filhos, totalizando um tamanho de 48 bytes.

Ordenação de vetor

O algoritmo utilizado para ordenação é uma implementação estável do Quicksort, não alterando a ordem de chaves com o mesmo valor. A alocação de memória é feita no momento de execução do algoritmo, fazendo com que o mesmo possua localidade espacial na memória e tornando mais rápida a busca de informações. Os testes realizados variam o tamanho de entrada em ordens de magnitude (potências de 10), e os experimentos executados foram com vetores de tamanho 10^5, 10^6, 10^7 e 10^8, totalizando 4 execuções.

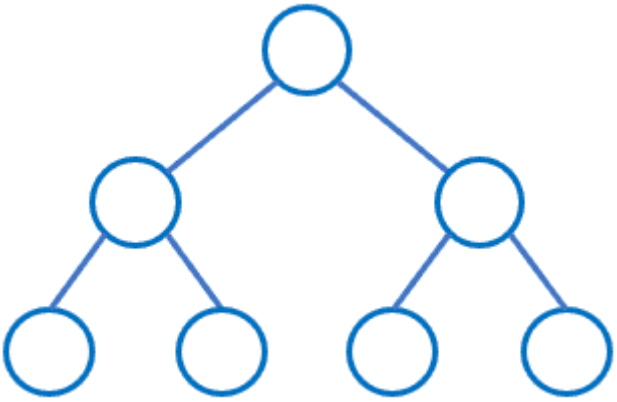
Multiplicação de Matrizes

Assim como o algoritmo de ordenação vetorial, o de multiplicação de matrizes também aloca a memória necessária no início da execução, fazendo com que o algoritmo também possua a propriedade de localidade espacial na memória, ou seja, que os dados estejam próximos um do outro, possibilitando ao processador carregar mais de um dados numa só busca, tornando o algoritmo mais rápido. Os testes realizados foram com matrizes de tamanho 100x100, 250x250, 500x500 e 1000x1000.

Caminhamento em grafo com DFS

Para os testes com grafos, definiu-se uma estrutura chamada *Node,* que contém informações acerca dos vizinhos do nodo, além de outros dados. O Grafo é montado como uma árvore binária completa conforme a altura desejada. Por

exemplo, para a entrada 3, o algoritmo construirá a seguinte árvore, retornando um ponteiro para a raiz:



Para percorrer a árvore em profundidade, implementou-se um DFS recursivo, uma vez que os nodos não são altamente conectados, apenas com seus pais e filhos. Os testes executados foram com árvores binárias completas de altura 10, 15, 20 e 25. Como neste algoritmo não há a propriedade de localidade espacial, pois a alocação de memória é dinâmica, os acessos a níveis mais baixos da cache se tornam maiores, e a busca mais lenta.

Coleta de dados

Para a coleta de dados, utilizou-se a ferramenta *perf*[2], passando como parâmetros as métricas desejadas. Os comandos foram executados passando argumentos em blocos de 4, visando evitar conflitos de registradores no processador. Os comandos executados podem ser visualizados abaixo:

perf commands

```
~$ sudo perf stat -e L1-dcache-load-misses,L1-dcache-loads,L2-load-misses ./main <options>
~$ sudo perf stat -e L2-loads,LLC-load-misses,LLC-loads ./main <options>
~$ sudo perf stat -e cycles,instructions,cache-misses,cache-references ./main <options>
~$ sudo perf stat -e dTLB-load-misses,dTLB-store-misses,dTLB-stores ./main <options>
```

[2] perf é uma ferramenta utilizada para medir o desempenho de aplicações por meio de informações de contadores de performance do sistema Linux. <https://perf.wiki.kernel.org/index.php/Main_Page>

TESTES REALIZADOS E DADOS COLETADOS

Os dados coletados[3] foram separados em diferentes tabelas para facilitar a observação. A primeira tabela, em cada algoritmo, mostra informações de tempo de execução, ciclos utilizados, instruções executadas, acessos e *misses* na memória cache. A segunda tabela apresenta informações acerca dos acessos à TLB (*Translation lookaside buffer*) e, por fim, a última tabela apresenta dados mais detalhados de cada nível da memória cache (L1, L2 e L3), com *misses* e hits.

Ademais, vale ressaltar que os testes foram rodados em uma máquina com L1 de tamanho 32K, L2 de tamanho 256K e L3 de tamanho 6144K.

Ordenação de vetor

Elements	seconds	cycles	instructions	cache references	cache misses
10^5	0,04043981	31.222.092	58.810.622	264.318	109.441
10^6	0,218168246	279.182.909	552.794.198	958.220	564.500
10^7	1,524112159	2.750.193.068	5.493.484.287	7.155.769	4.874.570
10^8	14,19071219	27.873.771.837	54.875.452.721	68.989.617	46.679.311

Tabela 1: Dados básicos do algoritmo de ordenação.

Elements	dTLB load misses	dTLB store misses	dTLB stores
10^5	1.129	2.817	7.885.900
10^6	4.090	30.924	74.357.151
10^7	33.664	306.928	739.661.106
10^8	252.640	2.827.103	7.393.237.313

Tabela 2: Dados de acesso à TLB do algoritmo de ordenação.

Elements	L1 misses	L1 loads	L1 miss perc	L2 misses	L2 loads	L2 miss perc
10^5	112.319	17.069.082	0,66%	8.449	30.799	27,43%
10^6	492.238	161.662.645	0,30%	16.884	53.909	31,32%
10^7	4.673.096	1.608.262.322	0,29%	97.362	218.519	44,56%
10^8	45.267.957	16.070.369.066	0,28%	1.331.330	4.213.808	31,59%

[3] Todos os dados podem ser encontrados em formato .csv no repositório do projeto no GitHub. <https://github.com/marcelobasso/INF01112-arch-2/tree/main/cache-tests/data>

Elements	LLC misses	LLC loads	LLC miss perc
10^5	9.155	30.799	29,72%
10^6	17.637	53.909	32,72%
10^7	70.576	218.519	32,30%
10^8	1.288.223	4.213.808	30,57%

Tabela 3: Dados de acesso à memória cache do algoritmo de ordenação.

Para facilitar a observação dos dados e a análise, seguem alguns gráficos importantes sobre os dados coletados.

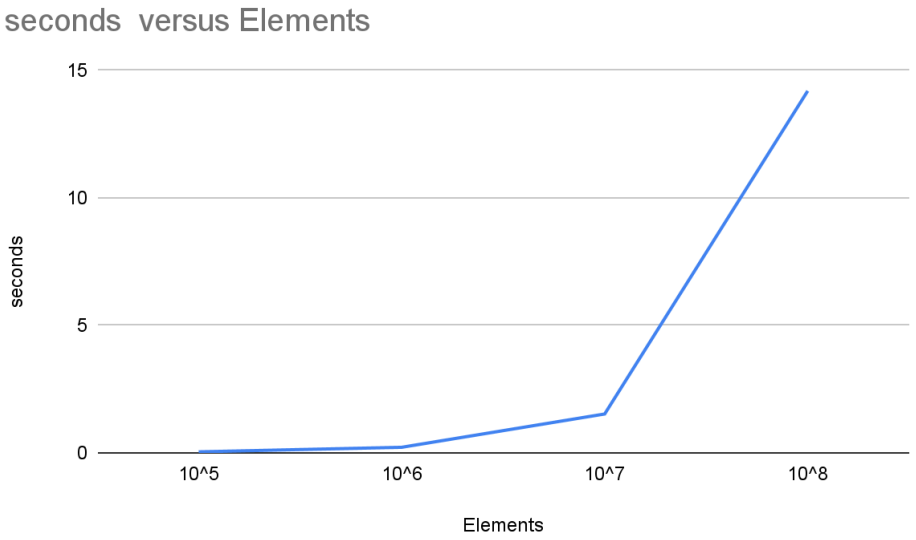


Gráfico 1: tempo de execução vs tamanho do vetor.

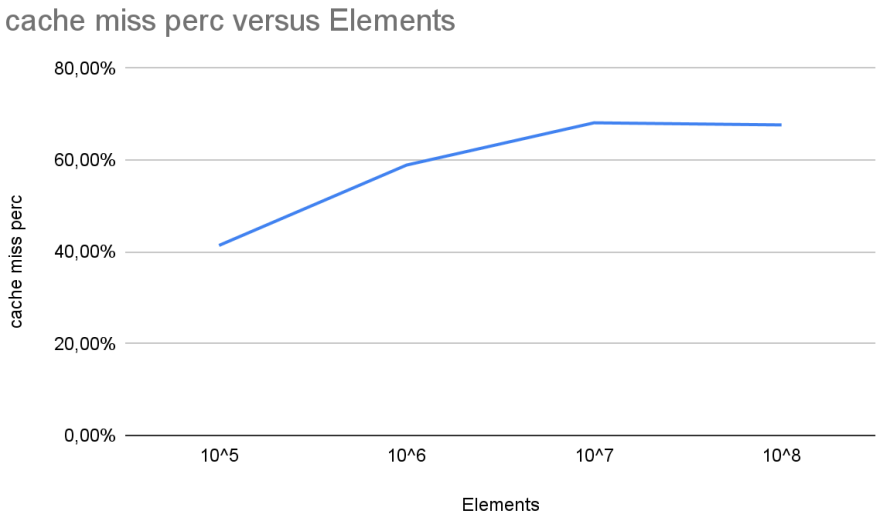


Gráfico 2: Cache misses vs tamanho do vetor.

Multiplicação de Matrizes

Elements	seconds	cycles	instructions	cache references	cache misses
100x100	0,076735309	58.707.452	134.902.073	395.676	101.630
250x250	0,452877204	712.412.604	2.012.031.907	2.938.164	253.347
500x500	3,612868427	6.677.499.226	64.735.779	56.158.840	2.066.072
1000x1000	30,70325	58.043.390.690	127.615.254.323	447.147.697	99.376.378

Tabela 4: Dados básicos do algoritmo de multiplicação de matrizes.

Elements	dTLB load misses	dTLB store misses	dTLB stores
100x100	1.032	691	27.281.718
250x250	2.258	2.585	411.597.295
500x500	8.896	12.711	3.271.798.375
1000x1000	11.244.913	139.717	26.110.254.997

Tabela 5: Acessos à TLB do algoritmo de multiplicação de matrizes.

Elements	L1 misses	L1 loads	L1 miss perc	L2 misses	L2 loads	L2 miss perc
100x100	192.410	46.201.505	0,42%	11.630	26.379	44,09%
250x250	3.358.050	695.722.793	0,48%	38.692	245.074	15,79%
500x500	228.567.583	5.533.132.167	4,13%	1.306.391	8.219.709	15,89%
1000x1000	1.452.598.696	44.164.722.877	3,29%	37.688.286	234.381.393	16,08%

Elements	LLC misses	LLC loads	LCC miss perc
100x100	9.666	26.379	36,64%
250x250	45.777	245.074	18,68%
500x500	1.062.556	8.219.709	12,93%
1000x1000	39.086.337	234.381.393	16,68%

Tabela 6: Dados de acesso à memória cache do algoritmo de multiplicação de matrizes.

Seguem alguns gráficos para facilitar a visualização e interpretação dos dados coletados.

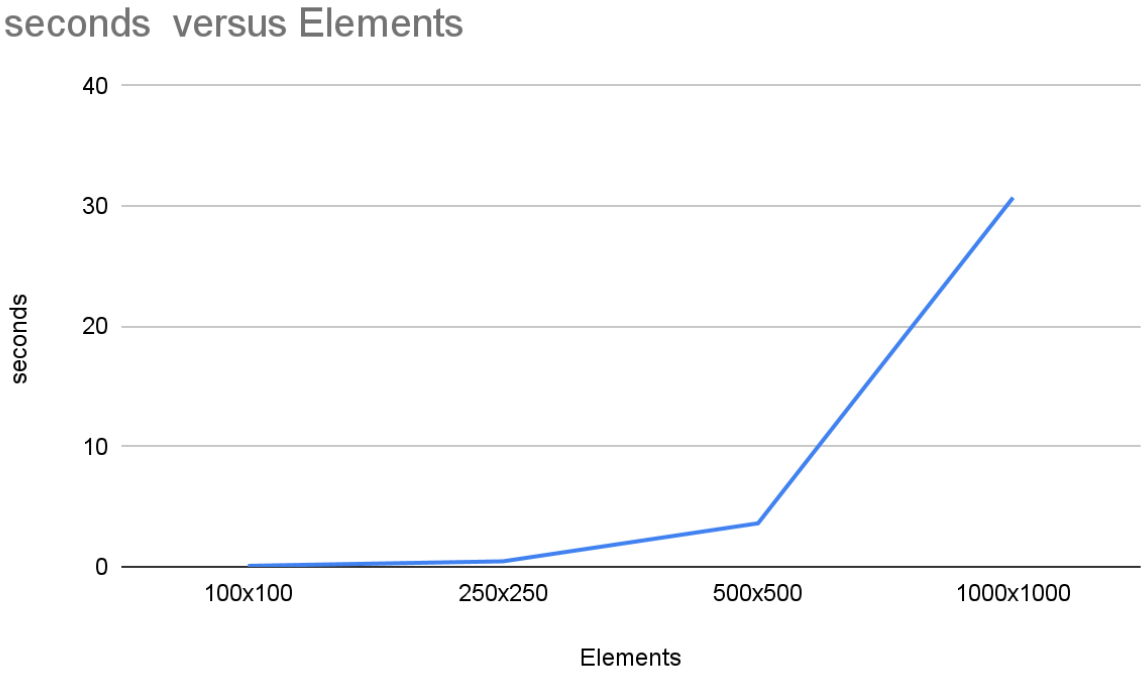


Gráfico 3: tempo de execução vs tamanho da matriz.

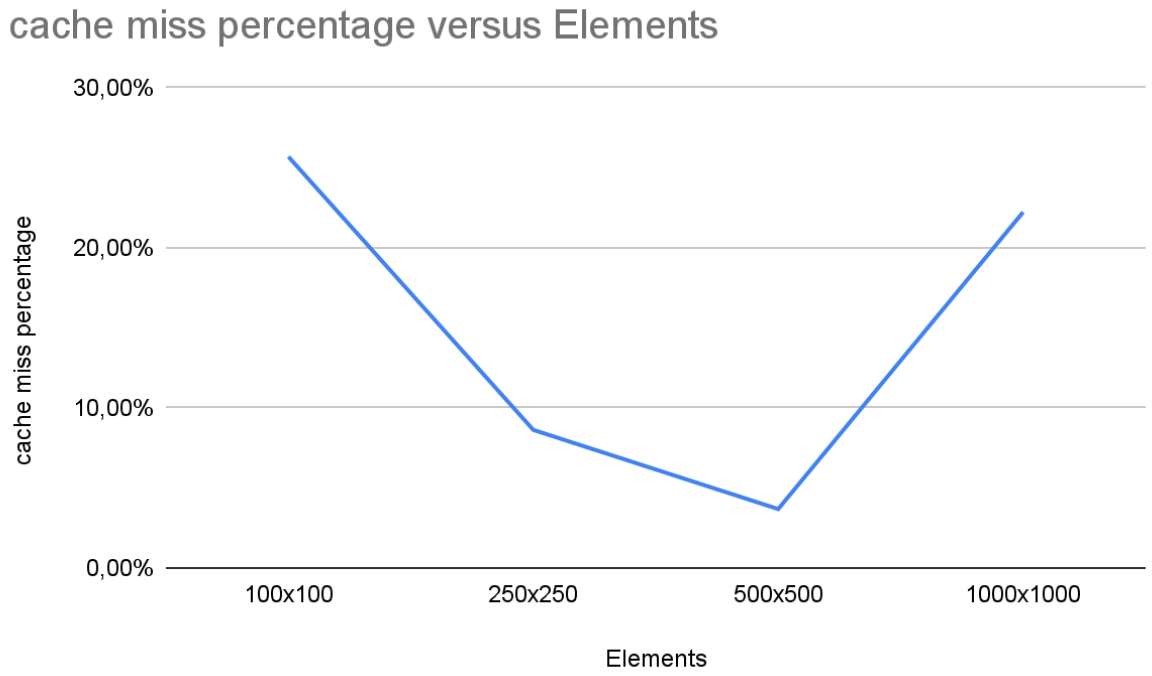


Gráfico 4: cache misses vs tamanho da matriz.

Caminhamento em grafo com DFS

height	seconds	cycles	instructions	cache references	cache misses
10	0,003285485	4.201.397	5.571.770	176.345	60.558
15	0,030361303	34.073.314	61.463.969	308.343	158.002
20	0,279720566	1.015.338.756	1.847.742.149	3.710.707	3.070.482
25	8,283629439	31.951.999.536	59.052.401.365	110.729.969	90.917.927

Tabela 7: Dados básicos do algoritmo de caminhamento em grafo.

height	dTLB load misses	dTLB store misses	dTLB stores
10	770	489	779.599
15	1.152	3.527	10.769.540
20	19.295	100.163	100.163
25	555.107	3.203.122	10.557.647.600

Tabela 8: Acessos à TLB do algoritmo de caminhamento em grafo.

height	L1 misses	L1 loads	L1 miss perc	L2 misses	L2 loads	L2 miss perc
10	45.828	1.465.647	3,13%	4.706	14.263	32,99%
15	139.047	17.157.613	0,81%	11.332	34.010	33,32%
20	2.745.399	518.744.545	0,53%	346.060	439.788	78,69%
25	87.257.238	16.587.174.084	0,53%	10.232.113	15.062.252	67,93%

height	*LLC misses*	*LLC loads*	*LLC miss perc*
10	*4.205*	*14.263*	*29,48%*
15	*10.060*	*34.010*	*29,58%*
20	*343.552*	*439.788*	*78,12%*
25	*11.019.305*	*15.062.252*	*73,16%*

Tabela 9: Dados de acesso à memória cache do algoritmo de caminhamento em grafo.

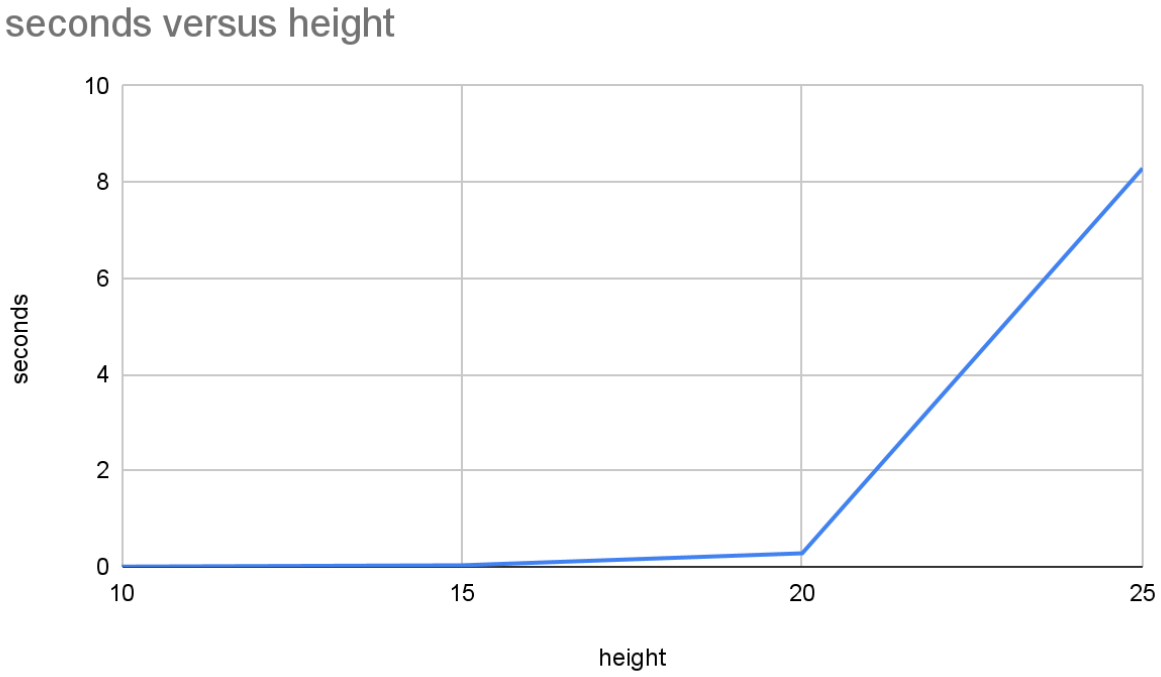


Gráfico 5: tempo de execução vs altura da árvore binária completa.

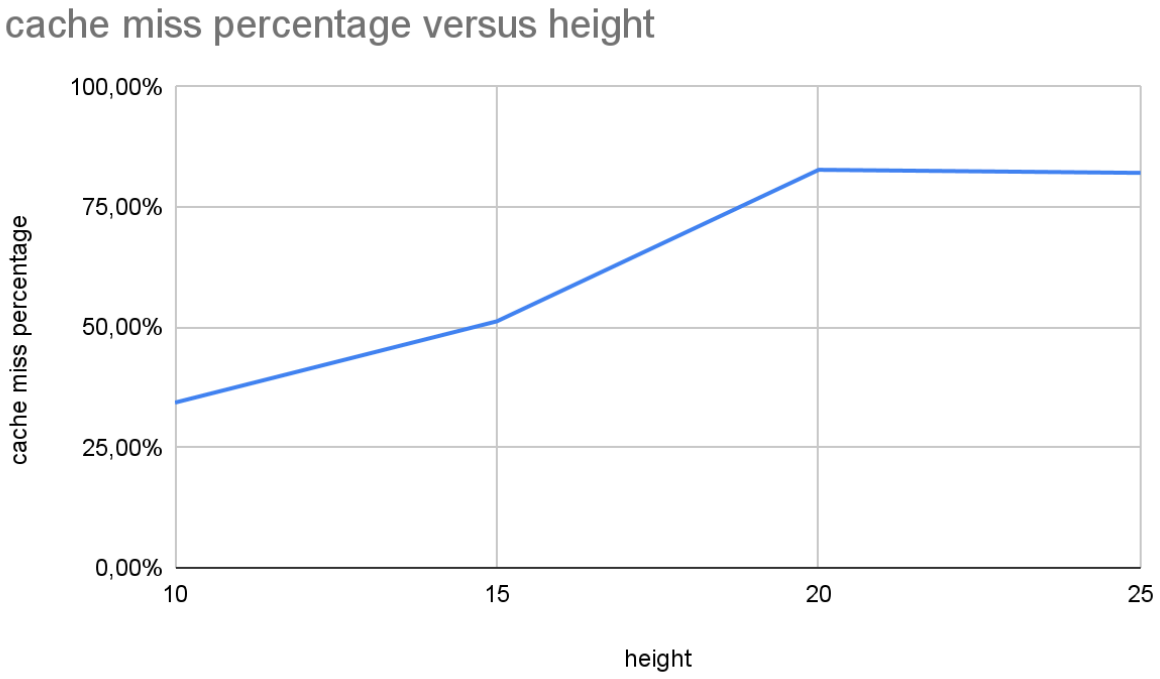


Gráfico 6: cache misses vs altura da árvore binária completa.

CONCLUSÃO E ANÁLISE DE DADOS

Com o grande volume de informações coletadas e comportamentos observados, é possível analisar diversos fatores acerca do comportamento da hierarquia de memória de acordo com diferentes estruturas de dados e níveis de ocupação da memória.

Tempo de execução

Observa-se um comportamento muito similar em todos os testes executados: o tempo de execução para estruturas de diferentes tamanhos não cresce de forma acentuada, até que, a partir do momento em que a memória cache é preenchida e os dados passam a ser buscados da memória principal - principalmente nas execuções com grande volume de itens processados - ocorre um crescimento destacável do tempo de execução. Isso se deve ao fato de o carregamento de dados da memória principal ser muito mais lento do que o de dados da memória cache, por necessitar de controladores e ter que atravessar o barramento da comunicação.

Graças à localidade espacial, que está presente nos dois primeiros algoritmos analisados, o carregamento de informações é feito mais rapidamente, pelo fato dos dados estarem armazenados de forma contígua na memória. No entanto, no terceiro algoritmo, por utilizar alocação dinâmica de memória, os dados encontram-se de forma mais esparsa na memória e isso gera um atraso maior no tempo de execução.

Cache misses

Nota-se que o algoritmo de caminhamento em grafo apresenta uma maior quantidade de misses na memória cache. Isso se deve, provavelmente, ao fato desse algoritmo trabalhar com alocação dinâmica de memória, propriedade que gera um efeito de cascata para a métrica analisada acima e para as demais. Enquanto isso, o algoritmo de ordenação vetorial ficou em segundo lugar na quantidade de misses, enquanto a multiplicação de matrizes apresentou a menor quantidade percentual de forma visível.

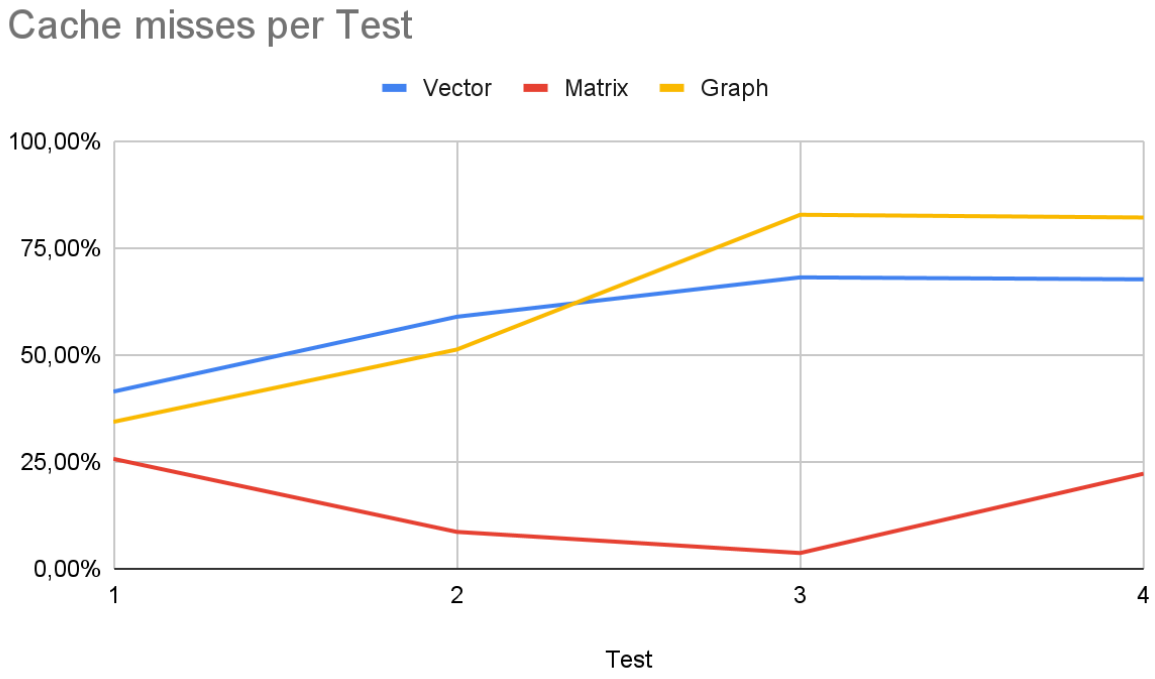


Gráfico 7: comparação do percentual de cache misses em cada teste para os algoritmos analisados.

Acessos à memória principal

Os acessos à memória principal foram analisados a partir das *misses* da LLC, uma vez que a próxima memória a ser consultada será a principal. Isso se deve ao fato de as métricas de mem-access e mem-loads não retornarem resultados válidos na arquitetura e configuração utilizadas.

Novamente, graças à ocupação esparsa da memória, o líder de acessos à memória principal foi o algoritmo de caminhamento em grafo. Os algoritmos de ordenação vetorial e multiplicação matricial apresentaram um comportamento semelhante. Nota-se uma quantidade maior de acessos dos testes de ordenação, mas talvez isso se deva ao tamanho das estruturas geradas, que nos testes vetoriais foram maiores do que nos testes matriciais.

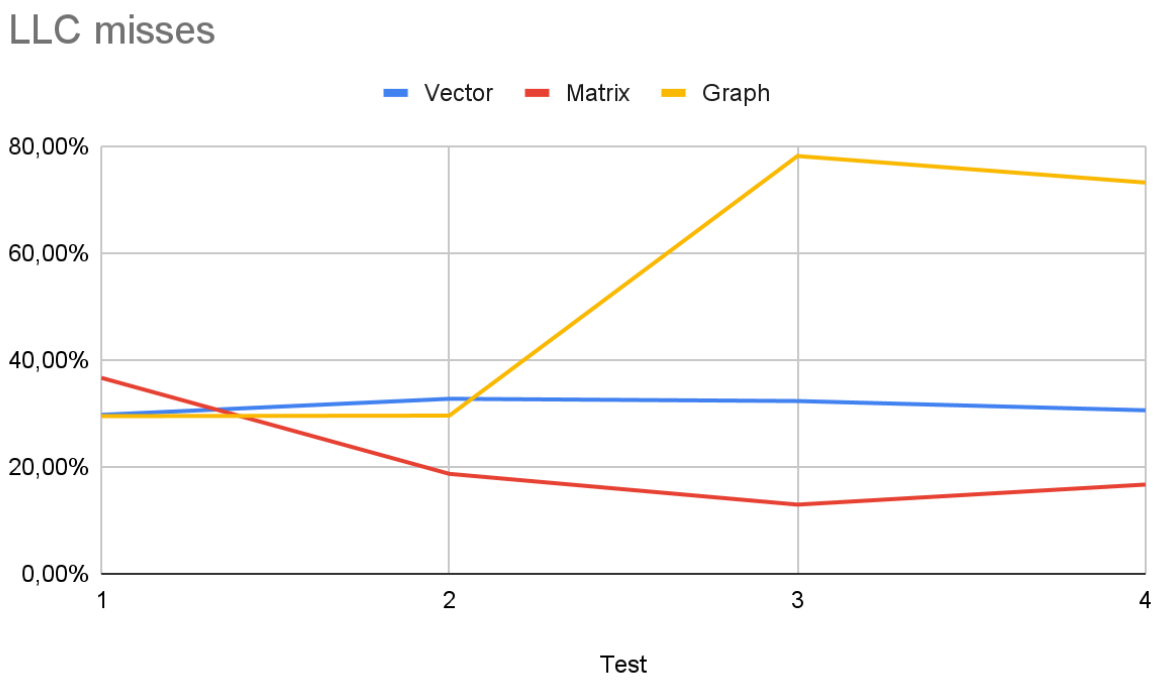


Gráfico 8: comparação de misses na LLC em cada testes para os algoritmos analisados.

Considerações finais

A partir dos dados coletados, pode-se observar o impacto das estruturas de dados utilizadas no desempenho das aplicações e o comportamento da hierarquia de memória. Tornou-se clara a defasagem na performance de programas que necessitam de acesso à memória principal, como nos testes com grandes volumes de dados, através das métricas de segundos transcorridos, ciclos e instruções executadas, por exemplo, que apresentam um crescimento não linear a partir do momento que os dados estouram a capacidade da memória cache.

A única estrutura que apresentou dados “curiosos” foram as matrizes que, por algum motivo, tiveram queda nas métricas de memória (mas não de performance) conforme aumentaram-se os tamanhos de entrada. Acredita-se que isso se deva a alguma otimização própria do compilador ou talvez da própria CPU, uma vez que computadores comumente lidam com operações matriciais em grande quantidade.