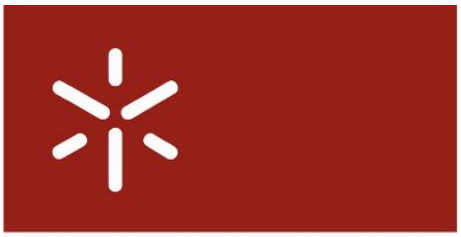
Universidade do Minho

Engenharia de Sistemas de Computação

Mestrado Integrado em Engenharia Informática



Universidade do Minho

Bernardo Silva - a77230

Francisco Lira - a73909

TP4

Análise de algoritmos com recurso à ferramenta perf e FlameGraphs

Conteúdo

1 Parte 1 - Análise de Algoritmos de Sorting

O objetivo desta parte do trabalho prático passa por identificar e retirar informações acerca da maneira como diversos algoritmos de sorting se comportam, quais os algoritmos com melhor performance, etc.

Para esta fase utilizou-se o meu computador pessoal, o qual possúi um processador Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz. Como tal, especialmente nos FlameGraphs poder-se-á reparar na existência de outros processos a correr ao mesmo tempo que o programa que estamos a estudar.

1.1 Obtenção de valores

Para a obtenção de valores escolheram-se vários indicadores, neste caso: Instructions, Cycles e Cache-Misses.

Utilizando o comando:

```
perf stat -e cycles,instructions,cache-misses ./<sort> 1 //
1 100000000
```

Obtiveram-se os seguintes resultados:

Tipo de Sort	Cycles	Instructions	Cache-Misses	Tempo
Quick Sort	72,751 E+9	76,449 E+9	164,972 E+6	20,239 s
Radix Sort	46,950 E+9	81,785 E+9	446,865 E+6	13,400 s
Heap Sort	190,097 E+9	134,587 E+9	447,005 E+6	55,986 s
Merge Sort	113,383 E+9	176,448 E+9	408,424 E+6	35,511 s

Valores obtidos utilizando perf stat

Em relação à razão pela qual os algoritmos obtêm este valor, os algoritmos que obteram melhor performance foram, por esta ordem, Radix Sort > Quick Sort > Merge Sort > Heap Sort.

O Quick Sort é um algoritmo que apesar de não ter um valor muito elevado em relação a instruções por ciclo (1,05), possúi um tempo de execução bastante bom. Isto provávelmente deve-se ao facto da maneira como o algoritmo acede à memória, sendo que os acessos no início do algoritmo são bastante lentos, devido ao facto de este ter de aceder ao array inteiro, mas acelera à medida que o array é subdividido, o que acaba por gerar um número muito reduzido de cache-misses, cuja maior parte se encontra provávelmente no início da execução do programa. Por outro lado o desempenho do mesmo depende da aleatoriedade do conjunto e do quão bem é escolhido o pivot. Se a distribuição de valores não for uniforme, a escolha do pivot poderá não ser indicada para a resolução deste algoritmo, tornando assim o programa bastante mais lento.

O Radix Sort é um algoritmo cuja complexidade é de O(N), obtendo a melhor complexidade de todos os outros algoritmos aqui presentes. Isto faz com que apesar de o número de instruções e cache misses ser relativamente elevado, o facto de o programa não ter de realizar tantos ciclos como nos outros, e as comparações serem apenas dígito a dígito, faz com que as instruções sejam realizadas mais rápidamente.

O Heap Sort foi o algoritmo com os piores resultados, provávelmente pois este depende de alterar uma árvore de pesquisa, o que não é muito bom e termos de acessos à memória e localidade espacial, estas esperas de memória provávelmente foi o que causou um número de ciclos tão elevado, pois este tem de aceder a níveis superiores de cache.

O Merge Sort apesar de ter um número elevado de instruções por ciclo (1,56), e um número muito reduzido de cache-misses, possúi um grande número de instruções o que causa um tempo de execução mais alto do que o Quick Sort. Este elevado número de instruções pode ser causado pelas alocações de memória derivadas da geração do array auxiliar.

1.2 Perfil de Execução

1.2.1 Quick Sort

```
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  .................  ............................
#
    95.58%          1950  sort     sort               [.] sort1
     1.45%            25  sort     sort               [.] ini_vector
     1.09%            30  sort     sort               [.] copy_vector
     0.69%            17  sort     libc-2.23.so       [.] __random
     0.66%            16  sort     libc-2.23.so       [.] __random_r
     0.28%             7  sort     libc-2.23.so       [.] rand
     0.18%            17  sort     [kernel.kallsyms]  [.] native_irq_return_iret
     0.08%             2  sort     sort               [.] rand@plt
```

O Quick Sort pelo que se pode observar, passa o maior tempo a realizar o algoritmo em si e não a aceder a bibliotecas externas. O ini_vector e copy_vector e os acessos às bibliotecas de aleatoriedade são realizados em todos os sorts para a geração do vetor inicial.

1.2.2 Radix Sort

```
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  .................  ............................
#
    92.62%          1214  sort     sort               [.] sort2
     1.57%            27  sort     sort               [.] copy_vector
     1.31%            17  sort     libc-2.23.so       [.] __random
     1.05%            16  sort     libc-2.23.so       [.] __random_r
     1.01%            15  sort     sort               [.] ini_vector
     0.97%             5  sort     sort               [.] rand@plt
```

```
     0.74%            15  sort     [kernel.kallsyms]  [.] native_irq_return_iret
     0.73%            11  sort     libc-2.23.so       [.] rand
```

O Radix Sort, ao contrário do Quick Sort, passou mais consideravelmente mais tempo na função rand@plt, apesar de não ser utilizado qualquer tipo de randomização neste algoritmo. Por outro lado a percentagem de overhead aumentou pois o tempo que se passa a correr o algoritmo em si é menor no Radix Sort do que no Quick Sort.

1.2.3 Heap Sort

```
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  .................  ............................
#
    98.33%          5231  sort     sort               [.] sort3
     0.45%            20  sort     libc-2.23.so       [.] __random
     0.38%            28  sort     sort               [.] copy_vector
     0.37%            22  sort     sort               [.] ini_vector
     0.31%            20  sort     libc-2.23.so       [.] __random_r
     0.08%             5  sort     libc-2.23.so       [.] rand
     0.06%            14  sort     [kernel.kallsyms]  [.] native_irq_return_iret
     0.03%             2  sort     sort               [.] rand@plt
```

O Heap Sort, tal como o Quick Sort, passa quase todo o tempo a utilizar a função do algoritmo em si.

1.2.4 Merge Sort

```
# Overhead       Samples  Command  Shared Object      Symbol
# ........  ............  .......  .................  ............................
#
    86.59%          2976  sort     sort               [.] aux_sort4
     3.10%           105  sort     sort               [.] sort4
     2.11%            72  sort     libc-2.23.so       [.] _int_free
     1.71%            59  sort     libc-2.23.so       [.] _int_malloc
     1.52%            52  sort     libc-2.23.so       [.] malloc
     0.96%            33  sort     libc-2.23.so       [.] free
     0.77%            16  sort     libc-2.23.so       [.] __random_r
     0.67%            23  sort     libc-2.23.so       [.] malloc_consolidate
     0.66%            27  sort     sort               [.] copy_vector
     0.62%            21  sort     libc-2.23.so       [.] __random
     0.53%            27  sort     [kernel.kallsyms]  [.] native_irq_return_iret
     0.44%            15  sort     sort               [.] ini_vector
     0.20%             7  sort     libc-2.23.so       [.] rand
     0.06%             2  sort     sort               [.] rand@plt
     0.03%             1  sort     sort               [.] free@plt
     0.03%             1  sort     sort               [.] malloc@plt
```

Devido ao facto de o Merge Sort possuir duas funções temos de ter em conta a soma do overhead das mesmas, para deduzir os acessos às bibliotecas. Mesmo somando o overhead das duas funções este não chega a tocar nos 90%

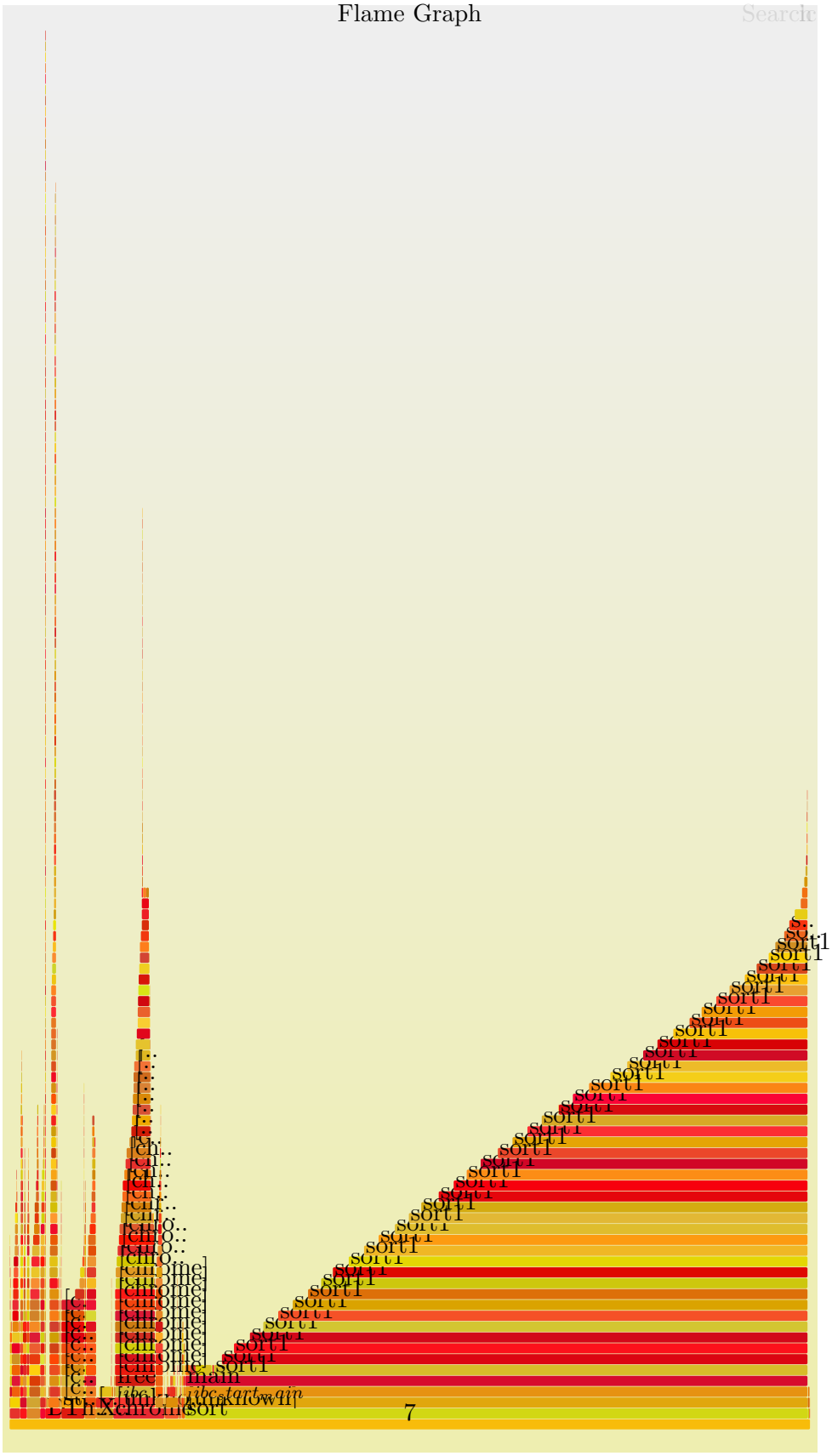
o que quer dizer que este algoritmo recorre bastante a bibliotecas externas, isto devido ao facto de ele ter de alocar um array auxiliar em cada iteração, como tínhamos indicado anteriormente.

1.3 FlameGraphs

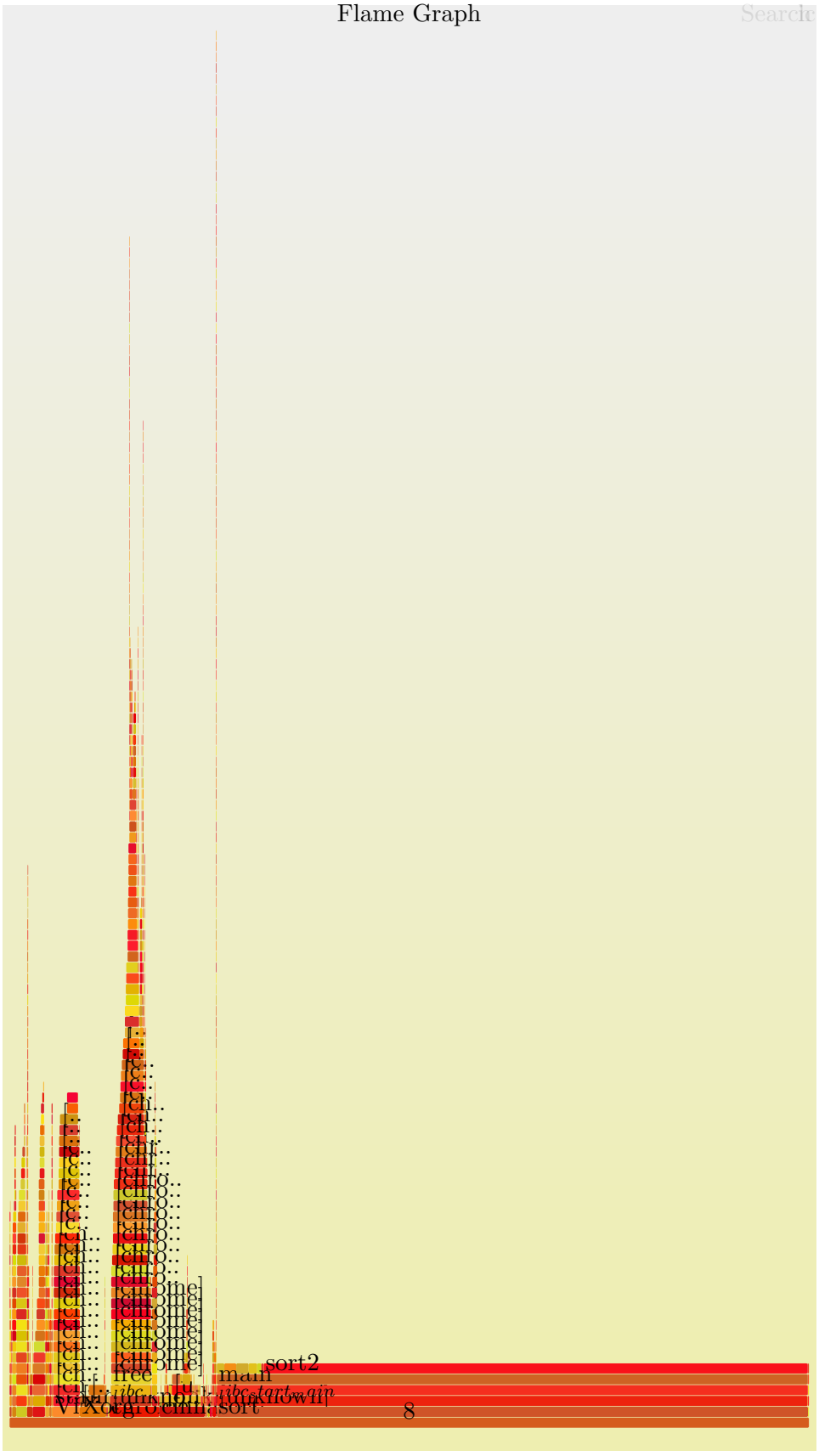
Para a obseração do comportamento de cada um destes algoritmos geraram-se flamegraphs para cada um deles.

Pode-se observar que tanto o RadixSort e HeapSort usam apenas a função de sorting em si, sendo assim difícil distinguí-los apartir do gráfico.

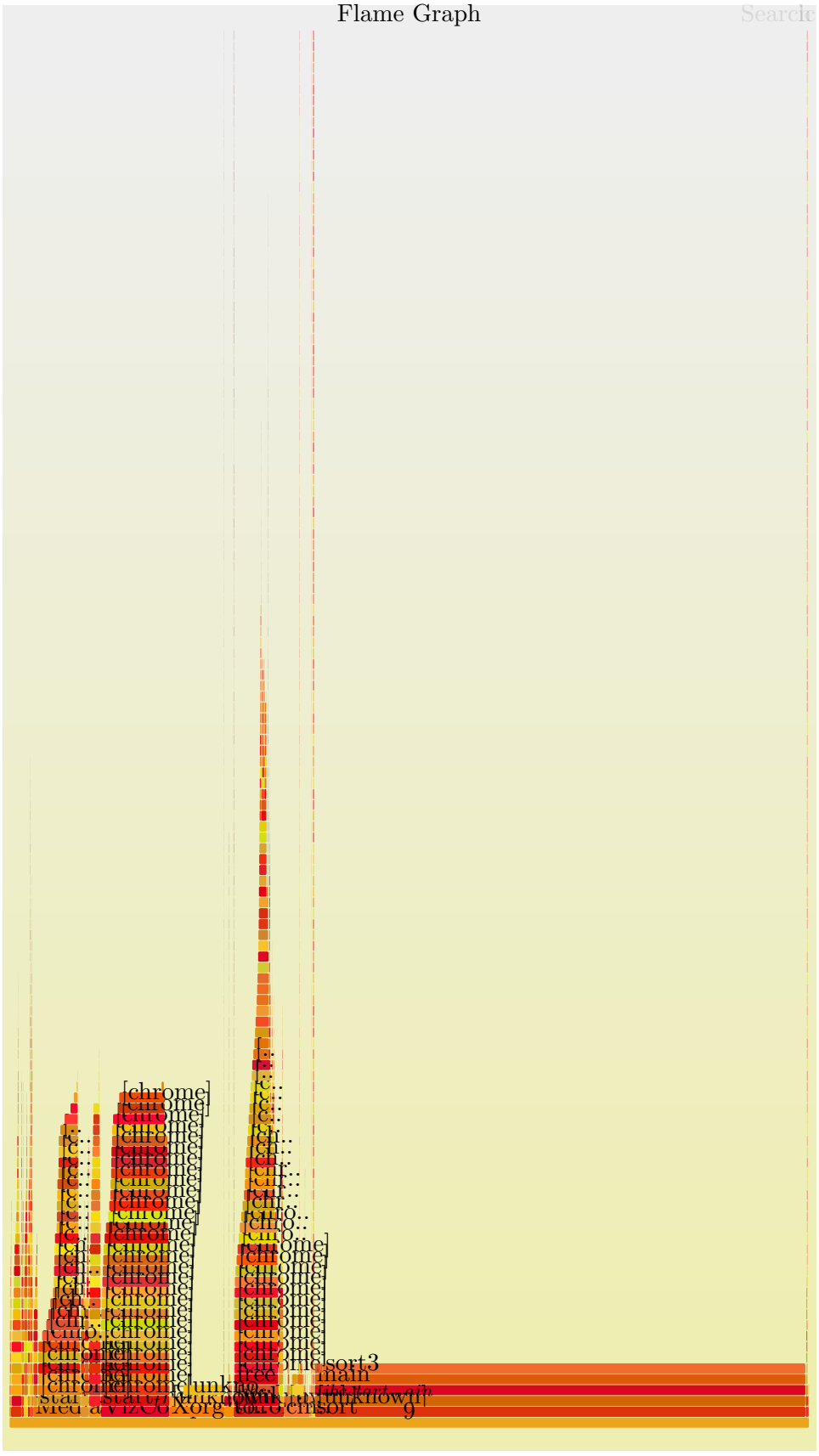
Pelo contrário, o QuickSort e o MergeSort resultam de recursão por isso é possível observar o todas as funções que foram chamadas para esse efeito. A grande diferença entre os dois é que o MergeSort além de chamar a função sort4 também chama a função auxiliar ao algoritmo o que garante assim a distinção entre os dois.



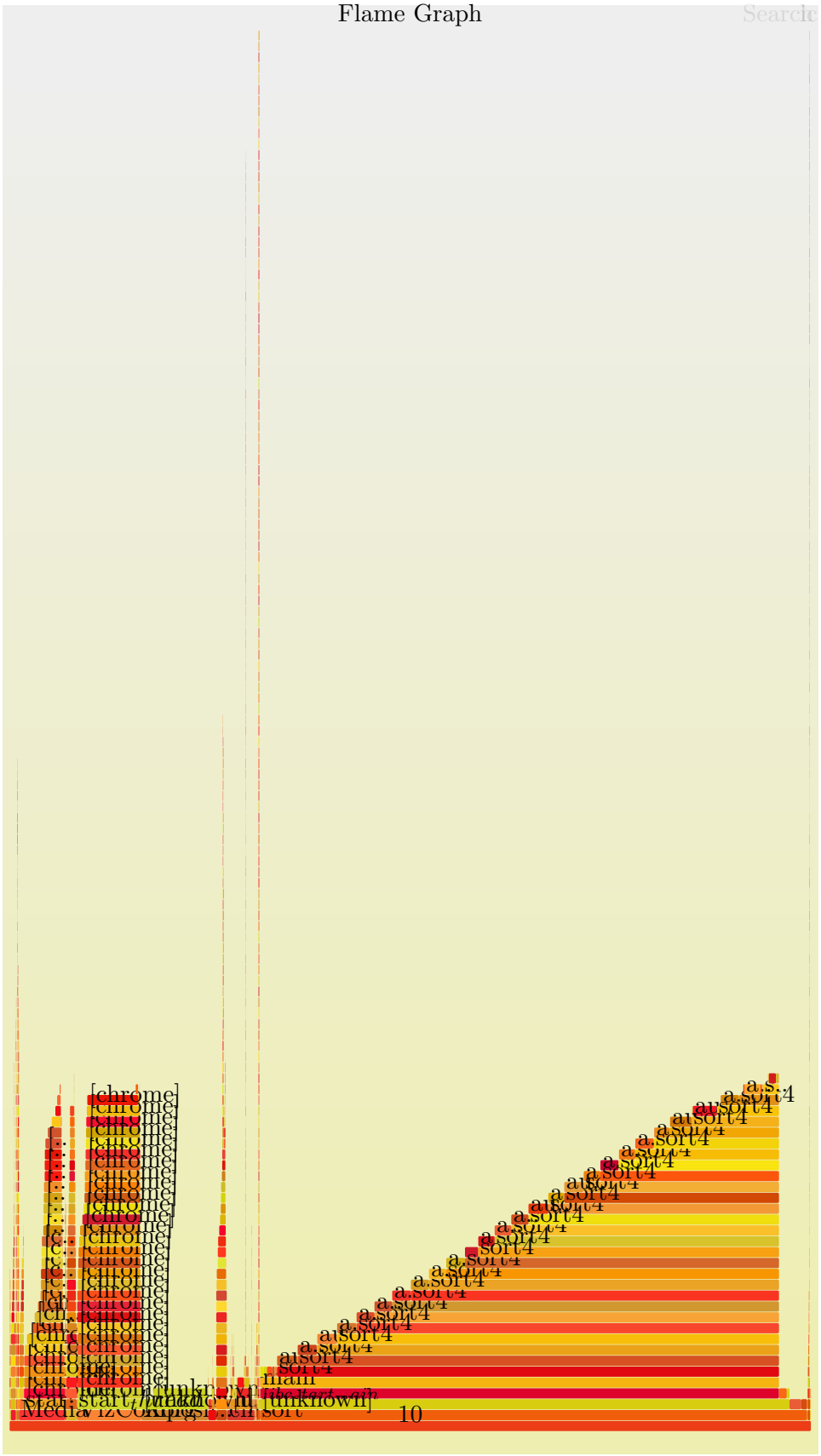
QuickSort FlameGraph



RadixSort FlameGraph



HeapSort FlameGraph



MergeSort FlameGraph

2 Parte 2 - Análise de Algoritmos de Multiplicação de Matrizes

Com ajuda dos tutoriais fornecidos pelo professor, foi-nos incumbida a tarefa de encontrar os pontos quentes de um programa de multiplicação de matrizes. Para tal tarefa definimos o tamanho de matriz como 2000x2000.

2.1 Encontrar os pontos quentes da execução

De forma a encontrar os pontos quentes da execução em primeiro lugar executou-se o comando perf stat de forma a obter alguns valores iniciais sobre o programa, tais como page-faults e tempo de execução.

```
> perf stat -e cpu-clock,faults ./naive

 Performance counter stats for './naive':

       5705.389200      cpu-clock (msec)
             1,083      faults

       5.712866600 seconds time elapsed
```

De seguida utilizou-se o comando perf record para obter o profile do programa em si, de forma a gerar reports mais tarde e obter o overhead das funções do programa.

```
> perf report --stdio --sort comm,dso

# Samples: 28K of event 'cpu-clock'
# Event count (approx.): 28939
#
# Overhead  Command      Shared Object
# ........  .......  .................
#
    98.67%    naive  naive
     0.78%    naive  libc-2.12.so
     0.54%    naive  [kernel.kallsyms]
     0.00%    naive  ld-2.12.so


# Samples: 44  of event 'faults'
# Event count (approx.): 1086
#
# Overhead  Command      Shared Object
```

```
# ........  .......  ..................
#
    84.07%    naive  naive
    15.65%    naive  ld-2.12.so
     0.28%    naive  [kernel.kallsyms]
```

De forma a nos focarmos nos shared objects de interesse utilizou-se a flag -dsos de forma a limitar os Shared Objects de interesse.

```
> perf report --stdio --dsos=naive,libc-2.13.so

# Overhead  Command  Shared Object          Symbol
# ........  .......  .............  ..............
#
    98.62%    naive  naive          [.] main
     0.36%    naive  libc-2.12.so   [.] __random
     0.35%    naive  libc-2.12.so   [.] __random_r
     0.07%    naive  libc-2.12.so   [.] rand
     0.05%    naive  naive          [.] rand@plt


# Samples: 44  of event 'faults'
# Event count (approx.): 1086
#
# Overhead  Command  Shared Object    Symbol
# ........  .......  .............  ........
#
    84.07%    naive  naive          [.] main
```

Para encontrar os locais do programa onde o overhead é maior pode-se usar o comando perf annotate de forma a obter um perfil mais específico do programa em si.

```
> perf annotate --stdio --dsos=naive --symbol=main

 Percent | Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------
         :
         :
         :
         : Disassembly of section .text:
         :
         : 00000000004005f0 <main>:
```

```
      :     }
      :   }
      : }
      :
      : int main(int argc, char* argv[])
      : {
 0.00 :   4005f0:       push   %r14
 0.00 :   4005f2:       xor    %r14d,%r14d
 0.00 :   4005f5:       push   %r13
 0.00 :   4005f7:       push   %r12
 0.00 :   4005f9:       push   %rbp
 0.00 :   4005fa:       push   %rbx
      :       matrix_r[i][j] = sum ;
      :     }
      :   }
      : }
      :
      : int main(int argc, char* argv[])
 0.00 :   4005fb:       movslq %r14d,%rbp
 0.00 :   4005fe:       xor    %ebx,%ebx
 0.00 :   400600:       imul   $0x1f40,%rbp,%rbp
 0.00 :   400607:       lea    0x2485a20(%rbp),%r13
 0.00 :   40060e:       lea    0x1543620(%rbp),%r12
 0.00 :   400615:       add    $0x601220,%rbp
 0.00 :   40061c:       nopl   0x0(%rax)
      : {
      :   int i, j ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
 0.02 :   400620:       callq  4003c8 <rand@plt>
 0.02 :   400625:       cvtsi2ss %eax,%xmm0
      : void initialize_matrices()
      : {
      :   int i, j ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
 0.09 :   400629:       add    $0x1,%ebx
      :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
 0.00 :   40062c:       mulss  0x1c4(%rip),%xmm0        # 4007f8 <__dso_handle+0x
 0.06 :   400634:       movss  %xmm0,0x0(%r13)
      : void initialize_matrices()
```

```
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.03 :   40063a:       add    $0x4,%r13
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   40063e:       callq  4003c8 <rand@plt>
0.04 :   400643:       cvtsi2ss %eax,%xmm0
     :       matrix_r[i][j] = 0.0 ;
0.07 :   400647:       movl   $0x0,0x0(%rbp)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.00 :   40064e:       add    $0x4,%rbp
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   400652:       mulss  0x19e(%rip),%xmm0        # 4007f8 <__dso_handle+0x
0.10 :   40065a:       movss  %xmm0,(%r12)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.02 :   400660:       add    $0x4,%r12
0.00 :   400664:       cmp    $0x7d0,%ebx
0.00 :   40066a:       jne    400620 <main+0x30>
     :
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   40066c:       add    $0x1,%r14d
0.00 :   400670:       cmp    $0x7d0,%r14d
0.00 :   400677:       jne    4005fb <main+0xb>
     :     for (j = 0 ; j < MSIZE ; j++) {
     :       float sum = 0.0 ;
     :       for (k = 0 ; k < MSIZE ; k++) {
```

```
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      :       }
      :       matrix_r[i][j] = sum ;
 0.00 :   400679:       xorps  %xmm2,%xmm2
      :
      : void initialize_matrices()
      : {
      :   int i, j ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   40067c:       mov    $0x601220,%edi
 0.00 :   400681:       mov    $0x2485a20,%esi
      :
      : void multiply_matrices()
      : {
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   400686:       xor    %ebx,%ebx
 0.00 :   400688:       nopl   0x0(%rax,%rax,1)
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      :       }
      :       matrix_r[i][j] = sum ;
 0.00 :   400690:       xorps  %xmm1,%xmm1
 0.00 :   400693:       lea    0x1543620(%rbx),%rcx
 0.00 :   40069a:       mov    %rsi,%rdx
 0.00 :   40069d:       xor    %eax,%eax
 0.00 :   40069f:       nop
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
11.13 :   4006a0:       movaps %xmm2,%xmm0
 0.01 :   4006a3:       movlps (%rdx),%xmm0
 0.49 :   4006a6:       movhps 0x8(%rdx),%xmm0
 1.57 :   4006aa:       add    $0x4,%rdx
10.36 :   4006ae:       shufps $0x0,%xmm0,%xmm0
 0.29 :   4006b2:       mulps  (%rcx,%rax,1),%xmm0
56.69 :   4006b6:       add    $0x1f40,%rax
```

```
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
 7.00 :   4006bc:       cmp    $0xf42400,%rax
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
 0.01 :   4006c2:       addps  %xmm0,%xmm1
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
11.95 :   4006c5:       jne    4006a0 <main+0xb0>
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      :       }
      :       matrix_r[i][j] = sum ;
 0.00 :   4006c7:       addps  %xmm2,%xmm1
 0.02 :   4006ca:       movaps %xmm1,(%rdi,%rbx,1)
 0.01 :   4006ce:       add    $0x10,%rbx
 0.00 :   4006d2:       cmp    $0x1f40,%rbx
 0.00 :   4006d9:       jne    400690 <main+0xa0>
 0.00 :   4006db:       add    $0x1f40,%rsi
 0.00 :   4006e2:       add    $0x1f40,%rdi
      :
      : void multiply_matrices()
      : {
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   4006e9:       cmp    $0x33c7e20,%rsi
 0.00 :   4006f0:       jne    400686 <main+0x96>
      : int main(int argc, char* argv[])
      : {
      :   initialize_matrices() ;
      :   multiply_matrices() ;
      :   return( EXIT_SUCCESS ) ;
      : }
 0.00 :   4006f2:       pop    %rbx
 0.00 :   4006f3:       pop    %rbp
 0.00 :   4006f4:       pop    %r12
 0.00 :   4006f6:       pop    %r13
```

```
    0.00 :   4006f8:       xor    %eax,%eax
    0.00 :   4006fa:       pop    %r14
    0.00 :   4006fc:       retq
```

Com este comando pode-se observar que o overhead provém quase completamente da seguinte linha, a qual corresponde a aproximadamente 78% de todo o overhead do programa:

```
sum = sum + (matrix_a[i][k] * matrix_b[k][j]);
```

De seguida vamos aumentar a frequência de sampling, de forma a aumentar a precisão do profile.

```
> perf record -e cpu-clock --freq=8000 ./naive

> perf report --stdio --show-nr-samples --dsos=naive

# dso: naive
# Samples: 58K of event 'cpu-clock'
# Event count (approx.): 58075
#
# Overhead       Samples  Command        Symbol
# ........  ............  .......  ............
#
    98.55%         57232    naive  [.] main
     0.04%            24    naive  [.] rand@plt
```

Com todos estes resultados pode-se observar que muito provávelmente o slowdown provém de acessos lentos à memória.

2.2 Eventos de desempenho de hardware

Nesta segunda fase, tendo identificado o hot spot do programa em análise, decidiu-se realizar uma segunda implementação do mesmo, no qual a memória é acedida de maneira diferente. Particularmente alterou-se a ordem dos ciclos de i>j>k para i>k>j.

De seguida com a ajuda do comando perf stat obtiveram-se os seguintes valores:

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

    20,482,364,019      instructions              #    1.07  insns per cycle
    19,085,647,497      cycles                    [62.47%]
     6,147,935,536      L1-dcache-loads
```

```
     2,131,206,220      L1-dcache-load-misses     #   34.67% of all L1-dcache hits
           99,460      dTLB-load-misses
       504,698,080      cache-references
         1,056,138      branch-misses             #    0.05% of all branches
     2,128,656,484      branch-instructions

       7.231122611 seconds time elapsed

Performance counter stats for './naive2':

    14,502,947,179      instructions              #    1.65  insns per cycle
     8,777,473,363      cycles                    [62.52%]
     4,145,861,213      L1-dcache-loads
       503,302,464      L1-dcache-load-misses     #   12.14% of all L1-dcache hits
            68,514      dTLB-load-misses
        78,895,754      cache-references
         4,036,699      branch-misses             #    0.19% of all branches
     2,129,233,291      branch-instructions

       3.336764310 seconds time elapsed
```

Apartir destes valores tornou-se possível a obtenção das seguintes métricas:

Métricas	naive	naive2
Instructions per cycle	1.07	1.65
L1 cache miss ratio	34.67%	12.14%
L1 cache miss rate PTI	104.05	34.7
Data TLB miss ratio	0.0001	0.0008
Data TLB miss rate PTI	0.005	0.005
Branch mispredict ratio	0.05%	0.19%
Branch mispredict rate PTI	0.05	0.27

Com estes valores pode-se observar que a segunda implementação possúi, além do óbvio melhoramento no tempo de execução, mais instruções por ciclo, e um número muito inferior de cache misses, sendo esta provávelmente a principal razão do slowdown.

2.3 Amostragem de eventos de desempenho de hardware

Esta fase consistia em aumentar a quantidade de samples de forma a não obter um overhead muito superior a 5%. Após alguns testes, cheguei a um número muito similar ao obtido no tutorial, 100000 samples. Apartir deste valor testei um perf report em cada um dos programas:

```
# =======
```

```
# captured on: Fri Jul  3 13:23:00 2020
# hostname : compute-431-8
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 3.16.3-1.el6.elrepo.x86_64
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 49551752 kB
# cmdline : /usr/bin/perf record -e cpu-cycles -c 100000 ./naive
# event : name = cpu-cycles, type = 0, config = 0x0, config1 = 0x0, config2 = 0x0, e
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# ========
#
# Samples: 211K of event 'cpu-cycles'
# Event count (approx.): 21102000000
#
# Overhead       Samples  Command      Shared Object                              Symb
# ........  ............  .......  .................  ..............................
#
    98.76%        208393    naive  naive              [.] main
     0.36%           767    naive  libc-2.12.so       [.] __random
     0.31%           646    naive  libc-2.12.so       [.] __random_r
     0.10%           212    naive  [kernel.kallsyms]  [k] clear_page_c


# ========
# captured on: Fri Jul  3 13:30:17 2020
# hostname : compute-431-8
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 3.16.3-1.el6.elrepo.x86_64
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 49551752 kB
# cmdline : /usr/bin/perf record -e cpu-cycles -c 100000 ./naive2
# event : name = cpu-cycles, type = 0, config = 0x0, config1 = 0x0, config2 = 0x0, e
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
```

```
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# ========
#
# Samples: 108K of event 'cpu-cycles'
# Event count (approx.): 10889700000
#
# Overhead       Samples  Command      Shared Object                              S
# ........  ............  .......  .................  ...............................
#
    97.62%        106308   naive2  naive2             [.] main
     0.72%           781   naive2  libc-2.12.so       [.] __random
     0.60%           656   naive2  libc-2.12.so       [.] __random_r
     0.19%           208   naive2  [kernel.kallsyms]  [k] clear_page_c
     0.14%           152   naive2  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.12%           136   naive2  libc-2.12.so       [.] rand
```

Após este feito procedi a analisar de novo o overhead no código com perf annotate:

```
 Percent | Source code & Disassembly of naive for cpu-cycles
------------------------------------------------------------------
         :
         :
         :
         : Disassembly of section .text:
         :
         : 00000000004005f0 <main>:
         :     }
         :   }
         : }
         :
         : int main(int argc, char* argv[])
         : {
    0.00 :   4005f0:       push   %r14
    0.00 :   4005f2:       xor    %r14d,%r14d
    0.00 :   4005f5:       push   %r13
    0.00 :   4005f7:       push   %r12
    0.00 :   4005f9:       push   %rbp
    0.00 :   4005fa:       push   %rbx
         :       matrix_r[i][j] = sum ;
         :     }
         :   }
         : }
         :
```

```
     : int main(int argc, char* argv[])
0.00 :   4005fb:       movslq %r14d,%rbp
0.00 :   4005fe:       xor    %ebx,%ebx
0.00 :   400600:       imul   $0x1f40,%rbp,%rbp
0.00 :   400607:       lea    0x2485a20(%rbp),%r13
0.00 :   40060e:       lea    0x1543620(%rbp),%r12
0.00 :   400615:       add    $0x601220,%rbp
0.00 :   40061c:       nopl   0x0(%rax)
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
0.01 :   400620:       callq  4003c8 <rand@plt>
0.02 :   400625:       cvtsi2ss %eax,%xmm0
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.04 :   400629:       add    $0x1,%ebx
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
0.00 :   40062c:       mulss  0x1c4(%rip),%xmm0        # 4007f8 <__dso_handle+0x
0.07 :   400634:       movss  %xmm0,0x0(%r13)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.01 :   40063a:       add    $0x4,%r13
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   40063e:       callq  4003c8 <rand@plt>
0.02 :   400643:       cvtsi2ss %eax,%xmm0
     :       matrix_r[i][j] = 0.0 ;
0.04 :   400647:       movl   $0x0,0x0(%rbp)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
```

```
     :     for (j = 0 ; j < MSIZE ; j++) {
0.00 :   40064e:       add    $0x4,%rbp
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   400652:       mulss  0x19e(%rip),%xmm0        # 4007f8 <__dso_handle+0x
0.06 :   40065a:       movss  %xmm0,(%r12)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.02 :   400660:       add    $0x4,%r12
0.00 :   400664:       cmp    $0x7d0,%ebx
0.00 :   40066a:       jne    400620 <main+0x30>
     :
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   40066c:       add    $0x1,%r14d
0.00 :   400670:       cmp    $0x7d0,%r14d
0.00 :   400677:       jne    4005fb <main+0xb>
     :     for (j = 0 ; j < MSIZE ; j++) {
     :       float sum = 0.0 ;
     :       for (k = 0 ; k < MSIZE ; k++) {
     :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
     :       }
     :       matrix_r[i][j] = sum ;
0.00 :   400679:       xorps  %xmm2,%xmm2
     :
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   40067c:       mov    $0x601220,%edi
0.00 :   400681:       mov    $0x2485a20,%esi
     :
     : void multiply_matrices()
     : {
     :   int i, j, k ;
     :
```

```
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   400686:       xor    %ebx,%ebx
 0.00 :   400688:       nopl   0x0(%rax,%rax,1)
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      :       }
      :       matrix_r[i][j] = sum ;
 0.01 :   400690:       xorps  %xmm1,%xmm1
 0.00 :   400693:       lea    0x1543620(%rbx),%rcx
 0.00 :   40069a:       mov    %rsi,%rdx
 0.00 :   40069d:       xor    %eax,%eax
 0.01 :   40069f:       nop
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
 9.73 :   4006a0:       movaps %xmm2,%xmm0
 0.00 :   4006a3:       movlps (%rdx),%xmm0
 0.46 :   4006a6:       movhps 0x8(%rdx),%xmm0
 1.52 :   4006aa:       add    $0x4,%rdx
 9.44 :   4006ae:       shufps $0x0,%xmm0,%xmm0
 0.27 :   4006b2:       mulps  (%rcx,%rax,1),%xmm0
60.16 :   4006b6:       add    $0x1f40,%rax
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
 6.19 :   4006bc:       cmp    $0xf42400,%rax
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
 0.00 :   4006c2:       addps  %xmm0,%xmm1
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (j = 0 ; j < MSIZE ; j++) {
      :       float sum = 0.0 ;
      :       for (k = 0 ; k < MSIZE ; k++) {
11.90 :   4006c5:       jne    4006a0 <main+0xb0>
      :         sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
```

```
      :       }
      :       matrix_r[i][j] = sum ;
 0.00 :   4006c7:       addps  %xmm2,%xmm1
 0.02 :   4006ca:       movaps %xmm1,(%rdi,%rbx,1)
 0.00 :   4006ce:       add    $0x10,%rbx
 0.00 :   4006d2:       cmp    $0x1f40,%rbx
 0.00 :   4006d9:       jne    400690 <main+0xa0>
 0.00 :   4006db:       add    $0x1f40,%rsi
 0.00 :   4006e2:       add    $0x1f40,%rdi
      :
      : void multiply_matrices()
      : {
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   4006e9:       cmp    $0x33c7e20,%rsi
 0.00 :   4006f0:       jne    400686 <main+0x96>
      : int main(int argc, char* argv[])
      : {
      :   initialize_matrices() ;
      :   multiply_matrices() ;
      :   return( EXIT_SUCCESS ) ;
      : }
 0.00 :   4006f2:       pop    %rbx
 0.00 :   4006f3:       pop    %rbp
 0.00 :   4006f4:       pop    %r12
 0.00 :   4006f6:       pop    %r13
 0.00 :   4006f8:       xor    %eax,%eax
 0.00 :   4006fa:       pop    %r14
 0.00 :   4006fc:       retq

 Percent | Source code & Disassembly of naive2 for cpu-cycles
-----------------------------------------------------------------------
      :
      :
      :
      : Disassembly of section .text:
      :
      : 00000000004005f0 <main>:
      :     }
      :   }
      : }
      :
      : int main(int argc, char* argv[])
```

```
     : {
0.00 :   4005f0:       push   %r14
0.00 :   4005f2:       xor    %r14d,%r14d
0.00 :   4005f5:       push   %r13
0.00 :   4005f7:       push   %r12
0.00 :   4005f9:       push   %rbp
0.00 :   4005fa:       push   %rbx
     :
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   4005fb:       movslq %r14d,%rbp
0.00 :   4005fe:       xor    %ebx,%ebx
0.00 :   400600:       imul   $0x1f40,%rbp,%rbp
0.00 :   400607:       lea    0x2485a20(%rbp),%r13
0.00 :   40060e:       lea    0x1543620(%rbp),%r12
0.00 :   400615:       add    $0x601220,%rbp
0.00 :   40061c:       nopl   0x0(%rax)
     :     for (j = 0 ; j < MSIZE ; j++) {
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
0.03 :   400620:       callq  4003c8 <rand@plt>
0.04 :   400625:       cvtsi2ss %eax,%xmm0
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.09 :   400629:       add    $0x1,%ebx
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
0.00 :   40062c:       mulss  0x1c4(%rip),%xmm0        # 4007f8 <__dso_handle+0x
0.12 :   400634:       movss  %xmm0,0x0(%r13)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.04 :   40063a:       add    $0x4,%r13
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   40063e:       callq  4003c8 <rand@plt>
```

```
0.03 :   400643:       cvtsi2ss %eax,%xmm0
     :       matrix_r[i][j] = 0.0 ;
0.10 :   400647:       movl   $0x0,0x0(%rbp)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.00 :   40064e:       add    $0x4,%rbp
     :       matrix_a[i][j] = (float) rand() / RAND_MAX ;
     :       matrix_b[i][j] = (float) rand() / RAND_MAX ;
0.00 :   400652:       mulss  0x19e(%rip),%xmm0        # 4007f8 <__dso_handle+0x
0.12 :   40065a:       movss  %xmm0,(%r12)
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
     :     for (j = 0 ; j < MSIZE ; j++) {
0.03 :   400660:       add    $0x4,%r12
0.00 :   400664:       cmp    $0x7d0,%ebx
0.00 :   40066a:       jne    400620 <main+0x30>
     :
     : void initialize_matrices()
     : {
     :   int i, j ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   40066c:       add    $0x1,%r14d
0.00 :   400670:       cmp    $0x7d0,%r14d
0.00 :   400677:       jne    4005fb <main+0xb>
0.00 :   400679:       mov    $0x601220,%edx
0.00 :   40067e:       xor    %esi,%esi
     :
     : void multiply_matrices()
     : {
     :   int i, j, k ;
     :
     :   for (i = 0 ; i < MSIZE ; i++) {
0.00 :   400680:       movslq %esi,%rbx
0.00 :   400683:       mov    $0x1543620,%ecx
0.00 :   400688:       imul   $0x1f40,%rbx,%rbx
0.00 :   40068f:       add    $0x2485a20,%rbx
```

```
 0.00 :   400696:       nopw   %cs:0x0(%rax,%rax,1)
      :     for (k = 0 ; k < MSIZE ; k++) {
 0.04 :   4006a0:       movss  (%rbx),%xmm1
 0.02 :   4006a4:       xor    %eax,%eax
 0.00 :   4006a6:       shufps $0x0,%xmm1,%xmm1
 0.02 :   4006aa:       nopw   0x0(%rax,%rax,1)
      :       for (j = 0 ; j < MSIZE ; j++) {
      :         matrix_r[i][j] = matrix_r[i][j] + (matrix_a[i][k] * matrix_b[k][j
20.55 :   4006b0:       movaps (%rcx,%rax,1),%xmm0
35.26 :   4006b4:       mulps  %xmm1,%xmm0
10.85 :   4006b7:       addps  (%rdx,%rax,1),%xmm0
27.71 :   4006bb:       movaps %xmm0,(%rdx,%rax,1)
 4.88 :   4006bf:       add    $0x10,%rax
 0.02 :   4006c3:       cmp    $0x1f40,%rax
 0.02 :   4006c9:       jne    4006b0 <main+0xc0>
 0.04 :   4006cb:       add    $0x1f40,%rcx
 0.01 :   4006d2:       add    $0x4,%rbx
      : void multiply_matrices()
      : {
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
      :     for (k = 0 ; k < MSIZE ; k++) {
 0.00 :   4006d6:       cmp    $0x2485a20,%rcx
 0.00 :   4006dd:       jne    4006a0 <main+0xb0>
      :
      : void multiply_matrices()
      : {
      :   int i, j, k ;
      :
      :   for (i = 0 ; i < MSIZE ; i++) {
 0.00 :   4006df:       add    $0x1,%esi
 0.00 :   4006e2:       add    $0x1f40,%rdx
 0.00 :   4006e9:       cmp    $0x7d0,%esi
 0.00 :   4006ef:       jne    400680 <main+0x90>
      : int main(int argc, char* argv[])
      : {
      :   initialize_matrices() ;
      :   multiply_matrices() ;
      :   return( EXIT_SUCCESS ) ;
      : }
 0.00 :   4006f1:       pop    %rbx
 0.00 :   4006f2:       pop    %rbp
 0.00 :   4006f3:       pop    %r12
```

```
0.00 :   4006f5:       pop    %r13
0.00 :   4006f7:       xor    %eax,%eax
0.00 :   4006f9:       pop    %r14
0.00 :   4006fb:       retq
```

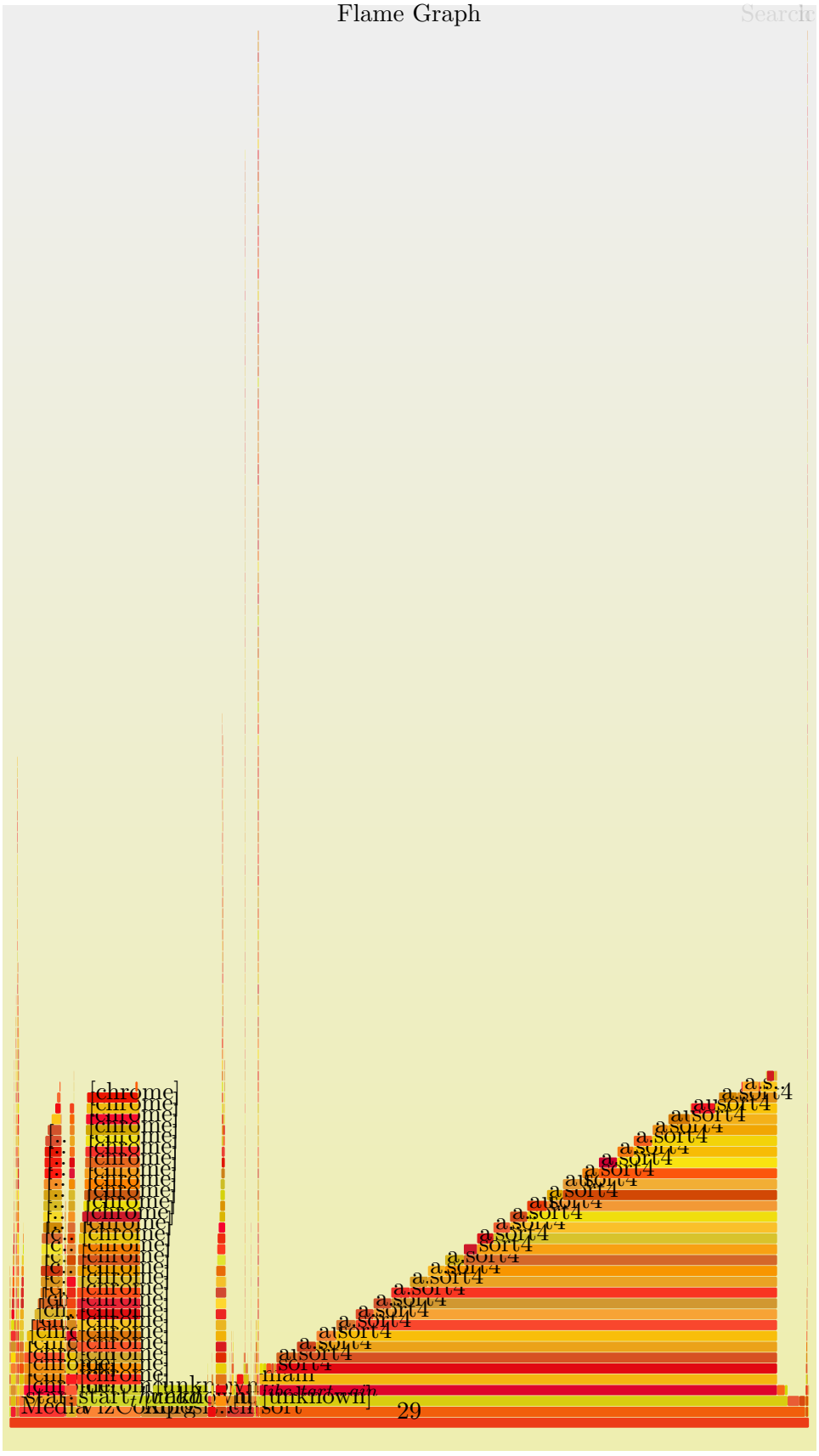
Aqui observa-se com garantia de uma melhor avaliação o overhead recebido por cada linha de código, devido a um valor de sampling superior.

2.4 Geração de FlameGraphs

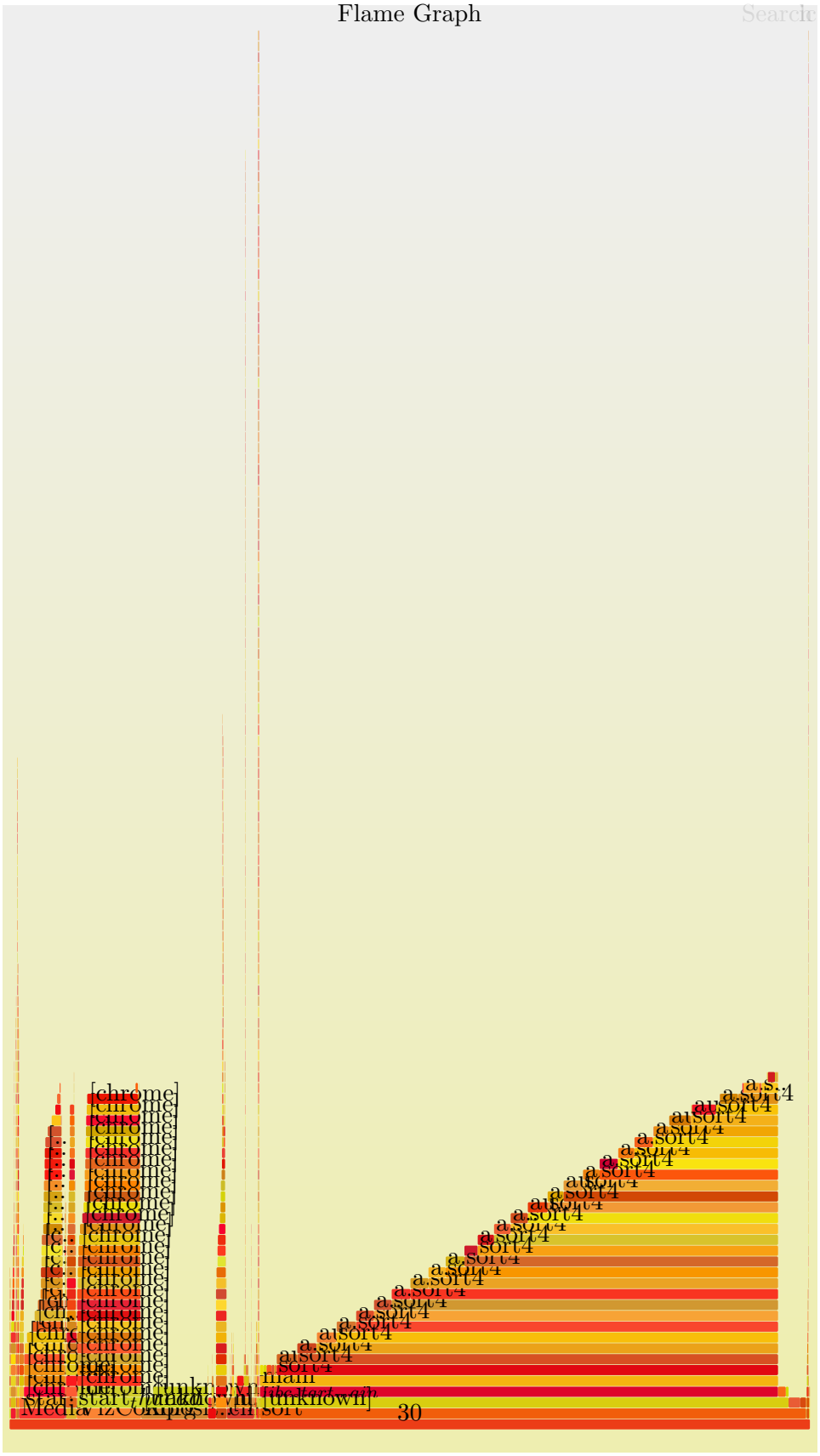
Com a ajuda da ferramenta FlameGraphs geraram-se os seguintes gráficos de desempenho de cada um dos programas:

3 Conclusão

Com este trabalho prático foi possível observarmos o poder e a utilidade da ferramenta perf na geração de relatórios de performance e na identificação dos hot spots dos programas, devido a tornar fácil a obtenção de valores específicos em relação aos acessos a bibliotecas externas e a linhas de código.


Flame Graph
Search
[chrome]
a.sort4
[unknown]
__libc_start_main
main
std::sort
29

Naive



Naive 2