Capítulo 3

Coisas para Fazer Antes de Paralelizar [1]

Alfredo Goldman (USP), Vitor Tessari Terra (USP) e Sarita Mazzini Bruschi (USP)

Abstract

In this short course, we will look at some practical things that should be done to improve code performance before thinking about parallelization. For example, on a 32-core machine, the potential gain of speeding up a sequential program with threads can be close to 32 times. On the other hand, by using a proper language, having good data structures and using the cache in the right way, the gains can be much greater. To this end, this short course reviews essential concepts for program evaluation and comparison, including examples and live demonstrations of how to apply them. Divided into two parts, the first part covers general aspects while the second part focuses on the individual analysis of a program. Part of the course was inspired by the Computer Language Benchmarks Game.

Resumo

Nesse minicurso, veremos algumas coisas práticas que devem ser feitas para melhorar o desempenho de código antes de se pensar em paralelização. Por exemplo, em uma máquina com 32 núcleos, o ganho potencial de aceleração de um programa sequencial com threads pode ser próximo a 32 vezes. Por outro lado, ao usar uma linguagem adequada, ter boas estruturas de dados e usar o cache da forma correta, os ganhos podem ser bem maiores. Para isso, este minicurso revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. Dividido em duas partes, a primeira parte aborda os aspectos gerais enquanto a segunda foca na análise individual de um programa. Parte do curso foi inspirada no Computer Language Benchmarks Game.

[1] Esse minicurso teve apoio do processo nº 2019/26702-8, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

3.1. Introdução

Uma das formas de se melhorar o desempenho de aplicações é por meio da paralelização, que é muito usada em computação de alto desempenho (do inglês, *HPC - High Performance Computing*). No entanto, antes de se pensar em como fazer com que partes de um programa possam ser executadas de forma paralela, é essencial saber avaliar o seu desempenho.

Várias técnicas estão disponíveis para paralelizar um código, tais como paralelização manual, paralelização automática na compilação, paralelização de laços, uso de bibliotecas com funções já paralelizadas, uso de programação baseada em diretivas, etc. Dependendo da escolha, o esforço necessário para paralelizar um código é alto. Partindo do pressuposto que já existe um código sequencial, temos como objetivo entender melhor como avaliar e comparar diferentes variações de um mesmo pseudo-código antes de decidir paralelizar o código.

O texto desse minicurso é uma versão atualizada do minicurso "Coisas para saber antes de fazer o seu próprio *Benchmarks Game*", apresentado na Escola Regional de Alto Desempenho - Região Sul, 2022 [Goldman et al. 2022, Capítulo 4]. Inspirado no *Computer Language Benchmarks Game* [2], este minicurso revisa conceitos essenciais para a avaliação e comparação de programas, incluindo exemplos e demonstrações ao vivo de como aplicá-los. Dividido em duas partes, a primeira parte aborda os aspectos gerais, enquanto a segunda foca na análise individual de um programa.

Na primeira parte utilizaremos um conjunto de programas que implementam um mesmo algoritmo (pseudo-código) em linguagens diferentes. Em seguida, será construído um `Makefile` para comparar o desempenho dos programas. Vamos:

- apresentar algumas métricas para avaliação de um programa (tempo, espaço, impacto energético, memória, ...);
- detalhar do que é composto o tempo de execução (tempo real, em espaço de usuário e espaço do sistema operacional);
- mostrar formas de tabular os dados coletados, meios de avaliar, interpretar e visualizar as medições;
- e por fim discutir fatores básicos que afetam o desempenho com base nas medições (linguagem de programação, sistema operacional, hardware e estado de carga do sistema).

Na segunda parte vamos trabalhar nos programas de forma individual para explicar tópicos de análise de desempenho usando o `Makefile` desenvolvido na parte anterior. Vamos abordar:

- os diferentes tipos de relógios que um sistema pode oferecer (incluindo relógios disponibilizados pela arquitetura) usando C;

[2] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

- o impacto de *flags* no tempo de execução e compilação usando o GCC;
- uso de compiladores/interpretadores alternativos e considerações sobre compilação JIT usando o PyPy;
- *profiling* usando a versão em Go;
- verificação de saída e soma de verificação usando Go.

Os códigos apresentados no texto estão disponíveis em `https://github.com/saritabruschi/Minicurso-WSCAD2022`.

3.2. O caso de uma maratona de programação paralela

Um caso que ilustra a necessidade de analisar outros fatores que alterem o desempenho antes de se paralelizar ocorreu na Maratona de Programação Paralela da ERAD-SP [3] 2022. A competição consistia em dois problemas, descritos de forma simplificada a seguir:

- Problema A: a entrada consiste em um texto codificado em ASCII (palavras separadas por espaços e texto terminado por `"\n"`) e um conjunto de palavras a serem removidas; a saída deve ser o texto sem as palavras removidas. Por exemplo:

  Texto: `"06506606503266065082032067068090010"`

  Texto decodificado: `"ABA BAR DEZ\n"`

  Palavras a remover: `"BOA", "BAR"`

  Saída: `"ABA DEZ\n"`

- Problema B: P números primos são divididos em duas partes cada – por exemplo, o número 504155039 pode ser dividido de diversas formas, tais como: 5041 e 55039, ou 504155 e 039. A entrada consiste em uma lista de $2P$ "partes" de números primos, sendo as P primeiras correspondentes à "parte esquerda" de um número primo e as P demais correspondentes à "parte direita"; a saída deve ser a lista dos possíveis números primos formados a partir dessas "partes", em ordem crescente. Por exemplo:

  Entrada: `"8 10 11 03 27 889"`

  Saída: `"827 1103 10889"`

Junto com o enunciado dos problemas, foram fornecidas soluções sequenciais como referência. O time vencedor seria aquele que obtivesse o maior *speedup*, ou seja, a maior redução de tempo de execução em relação à versão sequencial.

Para o Problema A, uma possível estratégia de paralelização seria dividir o texto em partes de tamanho próximo (sem quebrar palavras pela metade), de modo que cada *thread* seja responsável por remover as palavras de uma parte do texto. Já para o Problema B, há P^2 candidatos a números primos a serem formados – cada *thread* pode ser

[3] Escola Regional de Alto Desempenho de São Paulo

responsável por testar uma parte desses candidatos e devolver aqueles que forem, de fato, primos.

Surpreendentemente, as melhores soluções para os problemas A e B não aplicaram nenhuma estratégia de paralelização. Dispondo apenas de melhorias para as versões sequenciais, o time vencedor chegou a obter um *speedup* superior a 3000 para o Problema B! As otimizações adotadas foram as seguintes:

- Problema A: as palavras a serem removidas foram armazenadas em uma estrutura de dados mais eficiente para consultas (*trie* [4]). Além disso, a solução foi compilada com uma *flag* de otimização mais agressiva do que a versão inicial: `-Ofast` no lugar de `-O3`. Nesse caso, algumas garantias de corretude para a execução do programa podem ser perdidas [5], embora não tenha afetado a solução deste problema em particular.
- Problema B: a principal melhoria foi em relação ao teste de primalidade utilizado. Na versão sequencial fornecida, para testar se um número N é primo, testa-se a divisibilidade por todos os fatores possíveis até $\sqrt{N}$, de modo que o tempo consumido é $O(\sqrt{N})$, o que é exponencial no número de dígitos de N. Já a solução submetida usou o teste AKS de primalidade, cujo tempo consumido é poli-logarítmico no número de dígitos de do número a ser verificado N [6].

Em outras palavras, muitas vezes é possível obter melhorias significativas de desempenho sem necessariamente paralelizar o código, utilizando estruturas de dados, algoritmos e opções de compilação adequadas, por exemplo.

3.3. Conceitos importantes para avaliar o desempenho

Esta seção apresenta alguns conceitos importantes para que seja possível avaliar o desempenho de programas.

Primeiramente, um ponto importante é definir o que é desempenho. Em termos computacionais, o desempenho pode ser considerado como a quantidade de trabalho útil realizado por um computador. O desempenho deve ser mensurável e, para isso, algumas métricas são utilizadas.

O desempenho pode também ser absoluto ou relativo. Em termos absolutos, temos como exemplo as métricas: tempo de execução, latência, vazão (*throughput*), consumo energético e operações por segundo. Outras métricas podem ser calculadas por meio de operações aritméticas considerando várias métricas absolutas. Nesse caso, essas métricas fazem mais sentido quando são utilizadas para comparação entre sistemas, pois isoladamente elas não possuem significado. Por exemplo, podemos combinar métricas de tempo de execução e de gasto de energia simultaneamente - isso é feito em uma famosa lista de HPC verde [7].

[4] `https://pt.wikipedia.org/wiki/Trie`
[5] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
[6] `https://mathworld.wolfram.com/AKSPrimalityTest.html`
[7] `https://www.top500.org/lists/green500/`

Quando se deseja comparar o desempenho, é necessário que, de todos os fatores que possam influenciar no desempenho, somente um ou poucos fatores sofram variação. Com isso, é possível avaliar a influência da alteração daquele fator (através de simples comparação dos valores obtidos) ou dos poucos fatores que foram alterados (utlizando técnicas de análise de variância ou outras técnicas estatísticas).

Em computação, um *benchmark* é um programa (ou um conjunto de programas), definido especificamente com o objetivo de avaliar o desempenho, podendo ser executado em diferentes configurações. As métricas coletadas nas execuções podem então ser comparadas, sendo possível afirmar qual configuração tem o melhor desempenho sob algum aspecto. Um dos *benchmarks* mais conhecido é o NAS [8], que fornece diversos tipos de código, com diferentes características.

Vários são os fatores que influenciam o desempenho dos programas, incluindo fatores em nível de hardware (por exemplo, hierarquia de memória e níveis de cache, velocidade e conjunto de instruções do processador) e software (por exemplo, estrutura do código, compilador/interpretador e sistema operacional).

Como a análise de desempenho está intimamente ligada a medidas, é essencial também se levar em conta a precisão. Variações em medições de tempo de execução são usuais em computação, e existem várias formas de se procurar encontrar relevância estatística. A regra mais comum, que infelizmente não é usada de forma adequada, é a de se repetir a medição 30 vezes, e apresentar a média e o desvio padrão encontrado. Mas, em muitos casos isso não é suficiente, pois quando o desvio padrão é alto, a média pode deixar de ser significativa.

O tema de relevância estatística foge um pouco do escopo do texto, mas vamos mostrar um exemplo de como esse processo pode ser bem feito; ou seja, em uma medição, vamos mostrar a distribuição dos valores. Com isso podemos verificar se a distribuição obedece um padrão (se espera uma curva em forma de sino), e nesse caso, podemos falar em média e desvio padrão com segurança. Para o leitor interessado sugerimos as leituras de [Jain 1991, Kalibera and Jones 2013].

Vejamos os tempos de execução do programa a seguir. Ele aloca um vetor de tamanho `SIZE`, o preenche com números aleatórios e depois calcula a média dos números:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define SIZE  2000000
5
6  int media() {
7      int v[SIZE], sum = 0;
8      for (int i = 0; i < SIZE; i++)
9          v[i] = rand() % 100; // Random number between 0 and 99
10     for (int i = 0; i < SIZE; i++)
11         sum = sum + v[i];
12     return sum/SIZE;
13 }
14
15 void main() {
```

[8] https://www.nas.nasa.gov/software/npb.html

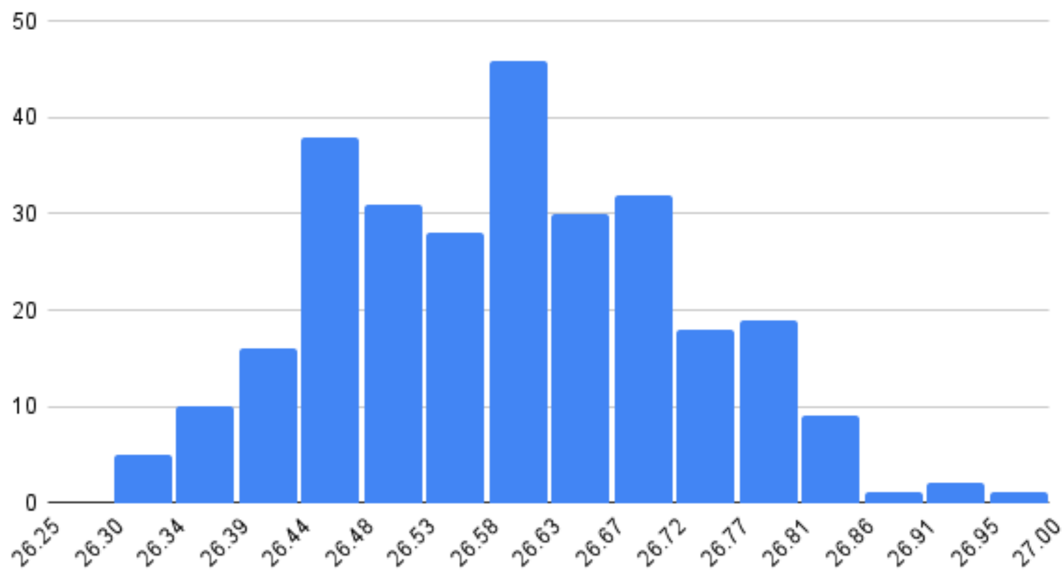


Figura 3.1. Histograma com os valores do tempo de execução da média dos elementos do vetor em milissegundos

```
16      clock_t start, end;
17      double cpu_time_used;
18      int val;
19      for (int i = 0; i < 300; i++){
20          start = clock();
21          val = media();
22          end = clock();
23          cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
24          printf("%g \n",  cpu_time_used);
25      }
26  }
```

Ao fazer um histograma com os valores obtidos em um processador i7 de nona geração (desconsiderando *outliers*, que são valores muito altos - sim, uma das execuções levou mais de 59 milissegundos), podemos observar na Figura 3.1 que a média sim, faz sentido. Dois indícios são importantes nessa distribuição: a forma de sino e o desvio padrão pequeno. Nesse caso, podemos dizer que a média foi de 26.6 milissegundos com desvio padrão 0.3 milissegundos.

No entanto, nem sempre a média representa de forma adequada o tempo de execução de um programa. Considere o programa a seguir, que calcula o produto de duas matrizes quadradas de ordem 2048 contendo valores aleatórios do tipo *double*. A variação dos tempos de execução é devida apenas às mudanças com relação aos tamanhos do blocos usados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
```

```
4  #include <time.h>
5
6
7  void matrix_fill_rand(int n, double *restrict _A)
8  {
9      #define A(i, j) _A[n*(i) + (j)]
10   int i, j;
11
12   for (i = 0; i < n; ++i)
13         for (j = 0; j < n; ++j)
14             A(i, j) = 10*(double) rand() / (double) RAND_MAX;
15
16     #undef A
17 }
18
19 void mmul(int n, double *restrict _C, double *restrict _A, double *
       restrict _B)
20 {
21     // Macros para acesso aos elementos das matrizes
22     #define A(i, j) _A[n*(i) + (j)]
23     #define B(i, j) _B[n*(i) + (j)]
24     #define C(i, j) _C[n*(i) + (j)]
25
26     int min_BLC = 32;
27     int max_BLC = 1024;
28
29     // Tamanho do bloco aleatorio entre min_BLC e max_BLC
30     int BLC = min_BLC + (rand()/(double) RAND_MAX)*(max_BLC - min_BLC);
31
32     int ib, jb, kb, i, j, k;
33     int num_blocks = n/BLC + 1;
34
35     for (ib = 0; ib < num_blocks; ++ib)
36         for (kb = 0; kb < num_blocks; ++kb)
37             for (jb = 0; jb < num_blocks; ++jb)
38                 for (i = ib*BLC; i < (ib+1)*BLC && i < n; ++i)
39                     for (k = kb*BLC; k < (kb+1)*BLC && k < n; ++k)
40                         for (j = jb*BLC; j < (jb+1)*BLC && j < n; ++j)
41                             C(i, j) += A(i, k)*B(k, j);
42
43     #undef A
44     #undef B
45     #undef C
46 }
47
48 int main()
49 {
50     double *restrict A;
51     double *restrict B;
52     double *restrict C;
53
54     int n = 2048;
55
56     // Aloque memoria com alinhamento de 8 bytes
57     A = aligned_alloc(8, n*n*sizeof(*A));
```

```
58      B = aligned_alloc(8, n*n*sizeof(*B));
59      C = aligned_alloc(8, n*n*sizeof(*C));
60
61      // Usar tempo atual como seed para geracao de numeros (pseudo-)
        aleatorios
62      srand(time(0));
63
64      // Matrizes A e B preenchidas com valores aleatorios
65      matrix_fill_rand(n, A);
66      matrix_fill_rand(n, B);
67
68      // Matriz C inicialmente preenchida com zeros
69      memset(C, 0x00, n*n*sizeof(*A));
70
71      clock_t start, end;
72      double cpu_time_used;
73
74      start = clock();
75      // Calcula o produto das matrizes A e B e armazena o
76      // resultado na matriz C
77      mmul(n, C, A, B);
78      end = clock();
79
80      cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
81
82      printf("%lf\n", cpu_time_used);
83
84      free(A);
85      free(B);
86      free(C);
87
88      return 0;
89  }
```

A Figura 3.2 mostra a distribuição dos tempos de 100 execuções do programa em um processador Intel i7-8550U 1.80GHz. Nesse caso, o tempo médio de execução foi de 5.2 segundos e o desvio-padrão de 0.8s, mais significativo do que no exemplo anterior. Apesar do desvio padrão ser relativamente pequeno, ao valor da média é pouco significativo.

3.3.1. Ferramentas utilizadas para medir o desempenho

Esta seção apresenta alguns programas utilizados para medir o desempenho de programas e as métricas que podem ser obtidas a partir deles. Além de programas, também é possível utilizar dispositivos físicos, como medidores de consumo de energia ou processadores auxiliares que obtêm informação fora do dispositivo.

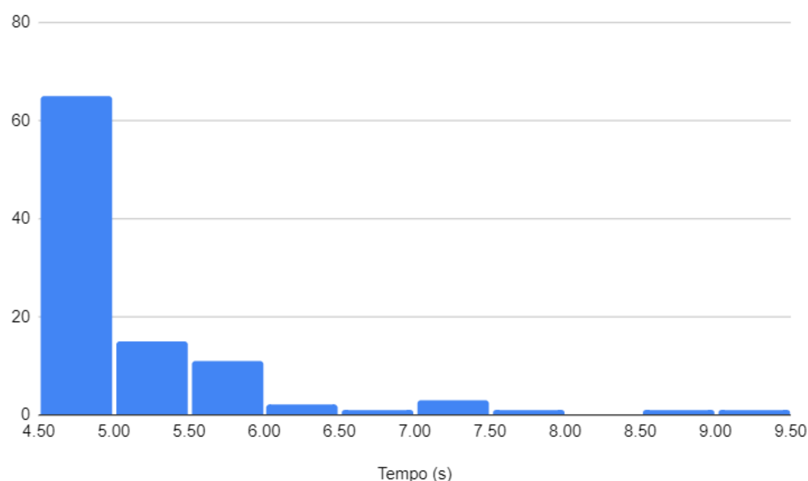


Figura 3.2. Histograma com os valores do tempo de execução da multiplicação de matrizes em segundos

3.3.1.1. Time

O utilitário `time`[9], especificado na ISO/IEC 9945-2:1993, é utilizado para medir a duração da execução de um programa qualquer em sistemas POSIX. Ele pode ser disponibilizado como um programa individual ou como um comando embutido em *shell*s. Sua saída padrão é dependente da implementação, assim, o argumento `-p` pode ser utilizado para emitir o resultado em um formato padronizado.

Na chamada padrão são mostrados três valores: o **real**, que corresponde ao tempo total (ou de relógio) entre o início e o final do programa, o tempo **user**, que é o tempo gasto pelas instruções do programa, e finalmente o **sys**, que o tempo usado em chamadas de sistema (*kernel*) do programa. Mas, há diversas *flgas* que podem ser usadas, por exemplo para obter também informações sobre a memória. A vantagem de se usar o `time` é que ele não é invasivo, mas só mede o programa como um todo.

3.3.1.2. Perf

O `perf`[10] é um comando que utiliza o subsistema de contadores de desempenho do *kernel Linux* para gerar métricas detalhadas do sistema. O `perf` oferece uma quantidade de eventos que podem ser obtidos a partir de diferente fontes. Algumas métricas são do próprio kernel do Linux (troca de contexto, *page-fault*) e outras são obtidas a partir dos contadores disponíveis na *Performance Monitoring Unit* - (PMU), tais como ciclos de processador utilizados e referências à memória cache.

Apesar de ser específico para sistema com o *kernel Linux*, outros sistemas possuem ferramentas com funcionalidades similares, como o *Instruments*[11] para *MacOS*.

[9] `https://pubs.opengroup.org/onlinepubs/009604499/utilities/time.html`
[10] `https://perf.wiki.kernel.org/index.php/Main_Page`
[11] `https://developer.apple.com/xcode/features/`

3.3.1.3. Valgrind

O *Valgrind*[12] é um arcabouço para análise de programas em tempo de execução. Apesar de ser conhecido como ferramenta para a detecção de vazamento de memória, ele também pode ser utilizado para a geração de árvores de execução e análise de uso do *cache* do processador. Devido à utilização de uma técnica de virtualização para a análise dos programas, ele aumenta consideravelmente a duração da execução do programa.

3.3.1.4. Vtune™ Profiler

O *Intel® Vtune™ Profiler*[13] é um ambiente integrado para análises de programas exclusivo para plataformas da *Intel*. Ele permite a análise de diversos tipos de programas, incluindo a análise de programas que rodam em *clusters MPI*, gerando um conjunto diverso de métricas, como contadores de desempenho de hardware, utilização do *cache* e consumo de energia.

3.3.2. Tabulação e análise dos dados

Esta seção apresenta informações sobre a tabulação dos dados coletados e algumas ferramentas para análise deles.

3.3.2.1. Resultados, parâmetros e ambiente

Além do registro dos resultados, é importante registrar informação sobre os parâmetros utilizados e o ambiente de execução. Os parâmetros compreendem as informações passadas que alteram o funcionamento do programa, como variáveis de ambiente que determinam número de *threads* a serem utilizadas ou os argumentos de execução. O ambiente de execução compreende dados sobre o *hardware* e o *software* em que o *benchmark* está sendo executado.

3.3.2.2. Tabulação

Para experimentos onde poucos dados são gerados, o registro manual dos dados é uma opção viável e pode ser feita utilizando uma planilha eletrônica. Para quantidades maiores de dados, o registro automático através de *script*s passam a ser um caminho mais vantajoso pela economia de tempo e menor chance de erros no processo de registro.

Ao automatizar o processo é necessário definir o formato de registro utilizado, como arquivos CSV ou bancos SQLite. Além da facilidade de registro, deve ser considerado o suporte de leitura do formato escolhido pelas ferramentas de análise. No caso de campos não estruturados, representações como JSON podem apresentar a flexibilidade necessária.

[12] `https://valgrind.org`

[13] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`

3.3.2.3. Análise

Após o registro dos dados, a análise pode ser realizada por uma vasta gama de programas. Além de planilhas eletrônicas, pode ser empregado o uso de programas como *Jupyter notebooks*[14] com bibliotecas como a *Matplotlib*[15], ou programas como o *Jamovi*[16].

3.4. Parte 1

Nosso objetivo nesta primeira parte do minicurso é apresentar como obter algumas métricas e utilizá-las para comparação de um mesmo algoritmo implementado em várias linguagens.

Para isso, utilizaremos o algoritmo *binary-trees* como *benchmark*, o qual é uma adaptação do benchmark de Hans Boehm para analisar *Garbage Collection* (GC) [17]. Esse *benchmark* aloca milhões de *short-lived trees* e percorre-as.

As linguagens que iremos comparar são: C, Python e Go. Os três códigos estão disponíveis no *Benchmarks Game* e são as versões que possuem o melhor desempenho em suas respectivas linguagens.

3.4.1. Compilação e Execução

Arquivos `makefile` são utilizados para definir regras de modo que o processo de compilação seja mais automático. Neste minicurso serão definidos dois arquivos de `makefile` para a compilação dos programas em C e Go.

O código em C utilizará o compilador GCC[18], o código em Go utilizará o compilador Go[19] e o código em Python utilizará o interpretador Python[20].

O `Makefile` para o programa em C define o compilador, as *flags* de compilação, os diretórios para os arquivos de *header* e das bibliotecas, o caminho onde é encontrada a biblioteca que será linkada ao arquivo executável, e o nome dos arquivos fonte, objeto e executável:

```
1  # the compiler: gcc for C program
2  CC=gcc
3
4  # compiler flags:
5  CFLAGS=-pipe -Wall -O3 -fomit-frame-pointer -march=ivybridge -fopenmp
6
7  # directory containing hearder files
8  INCLUDES=-I/usr/local/apr/include/apr-1
9
10 # library paths:
11 LFLAGS=-L/usr/local/apr/lib -fopenmp
12
```

[14]`https://jupyter.org`
[15]`https://matplotlib.org/`
[16]`https://www.jamovi.org`
[17]`https://hboehm.info/gc/gc_bench/`
[18]`https://gcc.gnu.org/`
[19]`https://go.dev/`
[20]`https://www.python.org/`

```
13 # library to link into executable
14 LIBS=-lapr-1
15
16 # C source file
17 SRCS=binary-trees.c
18
19 # C object file
20 # chance .c in SRSC by .o
21 OBJS=$(SRCS:.c=.o)
22
23 # executable file
24 MAIN=binary-trees
25
26 .PHONY: depend clean
27
28 all:  $(MAIN)
29   @echo File compiled
30
31 $(MAIN): $(OBJS)
32   $(CC) $(OBJS) -o $(MAIN)  $(LFLAGS) $(LIBS)
33
34 $(OBJS): $(SRCS)
35   $(CC) $(CFLAGS) $(INCLUDES) -c $(SRCS)
36
37 clean:
38   $(RM) *.o *~ $(MAIN)
```

Nesse exemplo do *benchmark binary-trees*, é necessário instalar o *Apache Portable Runtime*[21], o qual irá fazer o gerenciamento de alocação de memória do programa.

Para compilação do código em Go, basta ter instalado o compilador e executar o seguinte `Makefile`:

```
1 build:
2   go build -o binary-trees.go_run binarytrees.go-2.go
```

No caso da linguagem Python, como se trata de uma linguagem interpretada, não é necessário compilar o código, sendo que o comando será executado diretamente na execução.

O esforço de deixar claros todos os parâmetros usados, como ambiente, compilador e *flags* é essencial para permitir a reprodutibilidade dos experimentos.

Para facilitar a repetição da execução dos programas, como descrito em 3.3, foi elaborado um *script shell* que repete a execução dos três programas que serão comparados, coletando o tempo de execução de cada repetição e armazenando um arquivo texto:

```
1 # ===================
2 # ||    CLEANUP    ||
3 # ===================
4
5 rm results/*.out
6
7 # ===================
```

[21] `https://apr.apache.org/`

```
8  # ||    EXECUTION    ||
9  # ===================
10
11 SOURCE="binary-trees"
12 PARAM="15"
13
14
15 for i in $(seq 1 30)
16 do
17   (time C/$SOURCE $PARAM) 2>> results/C.out
18   (time python3 -OO Python/$SOURCE.py $PARAM) 2>> results/Python.out
19   (time Go/$SOURCE.go_run $PARAM) 2>> results/Go.out
20 done
```

3.4.2. Métricas

Em termos de métricas a serem analisadas, as ferramentas descritas em 3.3.1 podem fornecer muitas métricas, tais como: tempo de execução, quantidade de troca de contexto, *page-fault*, quantidade de ciclos de máquina utilizados, quantidade de instruções, *branchs* (desvios) e *branch-misses* (previsão errada no desvio). A saída abaixo é resultado da execução do `perf` com o seguinte comando:

```
1 sudo perf stat -o ../results/perf.out ./binary-trees 15
2        ../results/saida.out
```

```
1  # started on Fri Sep 30 18:24:23 2022
2
3
4   Performance counter stats for './binary-trees 15 saida.out':
5
6              43,72 msec task-clock          #    2,824 CPUs utilized
7                 85      context-switches    #    1,944 K/sec
8                  3      cpu-migrations      #   68,625 /sec
9                784      page-faults         #   17,934 K/sec
10       139.404.625      cycles              #    3,189 GHz
11       331.406.108      instructions        #    2,38  insn per cycle
12        58.226.381      branches            #    1,332 G/sec
13            98.349      branch-misses       #    0,17% of all branches
14
15       0,015479133 seconds time elapsed
16
17       0,044337000 seconds user
18       0,000000000 seconds sys
```

Se o teste estiver sendo executado em uma máquina virtual ou em um *container*, alguns resultados podem aparecer como `<not supported>` pois o `perf` não consegue coletar algumas métricas de hardware.

Para essa parte do minicurso, utilizaremos a ferramenta `time`, como já observado na Seção 3.4.1 e é com base nesses valores que faremos a comparação das linguagens.

3.4.3. Tabulando os dados

Para a tabulação dos dados, foi elaborado um programa em Python que lê os arquivos de saída gerados (C.out, Python.out e Go.out), trata os valores, insere-os em um *dataframe*,

salva o *dataframe* em um arquivo .csv e gera os gráficos para cada linguagem e cada métrica analisada (tempos real, user e sys).

```
import os, math
import subprocess
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

NUM_EXPERIMENTS = 30


def proccess_file(df, lang):
    data  = { 'language':lang }

    f     = open('results/{}.out'.format(lang))

    line = f.readline()
    line = f.readline()

    for i in range (NUM_EXPERIMENTS):
      if 'real' in line:
        # Trata a linha com a informacao do tempo real
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['real'] = time2
        # Trata a linha com a informacao do tempo user
        line = f.readline()
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['user'] = time2
        # Trata a linha com a informacao do tempo sys
        line = f.readline()
        line = line.strip().split()
        time = line[1].split("m")
        seg = time[1].split("s")
        seg = seg[0].replace(',', '.')
        time2 = float(time[0])*60 + float(seg)
        data['sys'] = time2
        line = f.readline()
        line = f.readline()
        df = df.append(data, ignore_index=True)

    f.close()

    return df

def proccess_data():
    languages = ['C', 'Python', 'Go']

```

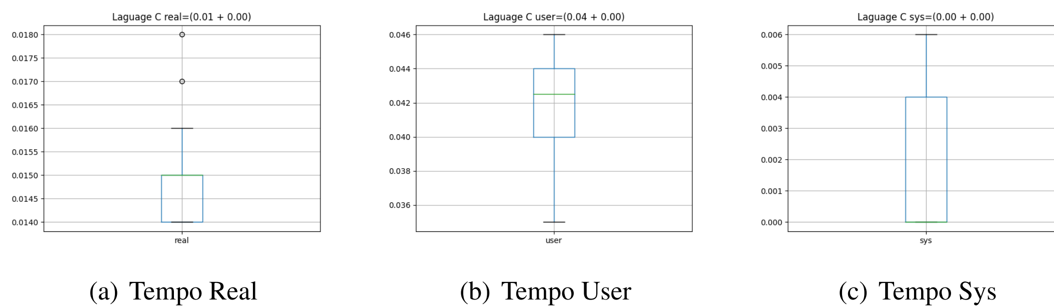
```
54      # Criando o DataFrame a ser populado
55      column_names = [’real’, ’user’, ’sys’]
56      df = pd.DataFrame(columns = column_names)
57
58      # Insere as informacoes de cada arquivo de resultado
59      for lang in languages:
60        df = proccess_file(df,lang)
61
62      return df
63
64
65  def processs_results(df):
66
67      languages = [’C’, ’Python’, ’Go’]
68      metrics = [’real’, ’user’, ’sys’]
69
70      # Gerando Box plot dos graficos de maneira simples e calculando o
       intervalo de confianca
71      for lang in languages:
72          if lang == ’C’:
73            str_option = ’C’
74          elif lang == ’Python’:
75            str_option = ’Python’
76          elif lang == ’Go’:
77            str_option = ’Go’
78
79          print("Gerando Imagens para Linguagem {}".format(str_option))
80
81          temp_df = df[df[’language’] == lang]
82          print(temp_df)
83          for c in metrics:
84              # Calculando o intervalo de confianca a 95% com t-student
85              conf_interval = (2.5096 * temp_df[c].std()) / math.sqrt(
       float(NUM_EXPERIMENTS))
86
87              plt.title("Laguage {} {}=({:.2f} + {:.2f})".format(
       str_option, c, temp_df[c].mean(), conf_interval ))
88              temp_df.boxplot(column=c)
89              plt.savefig(’figures/fig_{}_{}’.format(lang, c))
90              plt.close(’all’)
91
92
93  if __name__ == ’__main__’:
94
95      print("Processamento os arquivos de resutlados")
96      df = proccess_data()
97      print("Salvando resultados em disco (results/results.csv)...")
98      df.to_csv(’results/results.csv’, index=False, header=True)
99      print("Resultados salvos em um Dataframe {}.".format(df.shape))
100
101     print("Processando a Analise dos dados")
102     processs_results(df)
103     print("Analise finalizada")
```

Listagem 3.1. Código para ler os resultados e gerar os gráficos

3.4.4. Analisando os resultados

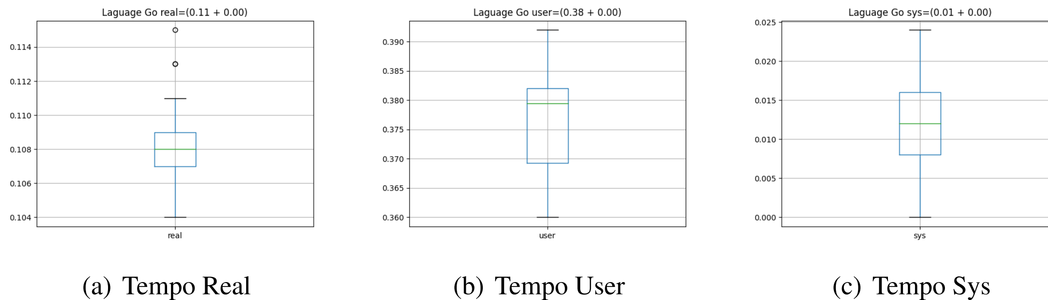
Os gráficos gerados pelo código da Listagem 3.1 podem ser observados nas Figuras 3.3, 3.5 e 3.4, as quais apresentam boxplots com 30 resultados para cada código para cada métrica. Além disso, são apresentados também os valores referentes à média e a metade do valor do intervalo de confiança com nível de confiança de 95%.

A linguagem que obteve o melhor desempenho (menor tempo de execução) é a linguagem C, seguida pela linguagem Go. Python obteve o pior desempenho. Esses resultados confirmam os resultados apresentados na página com os *benchmarks*.



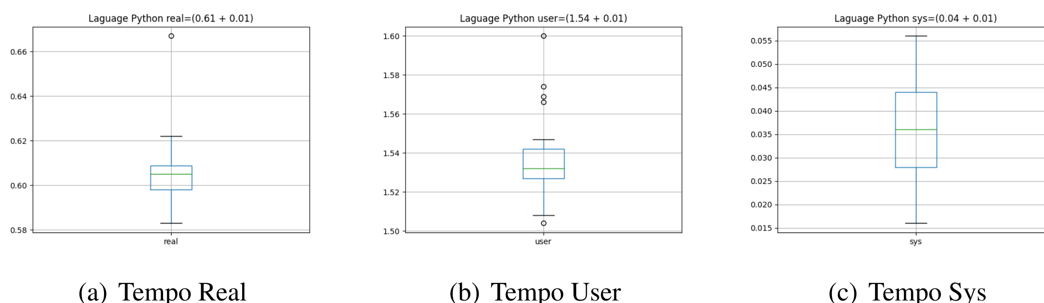
(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.3. Tempos para linguagem C



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.4. Tempos para linguagem Go



(a) Tempo Real (b) Tempo User (c) Tempo Sys

Figura 3.5. Tempos para linguagem Python

É interessante também ressaltar que uma outra métrica que pode ser usada é o valor mínimo, ou seja, entre todas a medidas a de menor valor. Dessa forma, temos que o

tempo real em C é 0.014 segundos, em Go, 0.1 segundos. O menor tempo em Python foi de 0.58 segundos.

Pode-se observar também que o tempo **real** é menor do que o tempo **user**, quando o que se esperava era que fosse o contrário. Isso significa que o código é CPU Bound e que aproveitou o paralelismo de execução em vários núcleos.

3.5. Parte 2

Nesta parte são explorados tópicos de análise de desempenho de programas. Utilizaremos o cálculo do conjunto Mandelbrot[22] para realizar as análises. Primeiro será apresentada a definição geral do programa, com implementações disponíveis para as linguagens C[23], Python[24] e Go[25]. Posteriormente são abordados os diferentes tipos de relógios que um sistema pode oferecer utilizando a versão em C; impacto de *flags* no tempo de execução e compilação usando o GCC; uso de compiladores/interpretadores alternativos usando o PyPy e considerações sobre compilação JIT; *profiling* usando a versão em Go; e verificação de saída e soma de verificação usando a versão em Go.

3.5.1. Conjunto Mandelbrot

O conjunto Mandelbrot consiste nos números complexos c tais que a função $f_c(z) = z^2 + c$ não diverge quando aplicada de forma recursiva: $f_c(f_c(...f_c(0)))$. Para fins computacionais, consideramos que um número complexo diverge quando, em algum momento da aplicação recursiva da função, a norma euclidiana do valor intermediário é maior que 2, e consideramos que um número complexo converge quando, após um número pré-estabelecido de iterações, nenhum de seus valores intermediários apresenta norma euclidiana maior que 2. O programa gera uma imagem 2D onde o eixo das abscissas representa a parte real e o eixo das ordenadas a parte imaginária.

O programa recebe como argumentos de entrada a largura (w) e altura (h) da imagem a ser gerada e os valores em ponto flutuante x_r, x_i, y_r e y_i que serão utilizados para definir o conjunto de pontos analisados. Para cada pixel da imagem (x,y), sendo $(0,0)$ o ponto superior esquerdo e $(w-1,h-1)$ o ponto inferior esquerdo, é atribuído o número complexo $(x/w*(y_r - x_r) + x_r) + (y/h*(x_i - y_i) + y_i)i$. Caso o número atribuído faça parte do conjunto Mandelbrot ele é pintado de branco, caso contrário, de preto. Por fim, o programa deve escrever na saída padrão a imagem utilizando o formato Netpbm P1[26].

3.5.2. Relógios do sistema

Sistemas operacionais modernos possuem diferentes fontes de tempo disponíveis para os processos. Essas fontes avançam com base em critérios diferentes e permitem obter diferentes métricas para análise.

[22] `https://en.wikipedia.org/wiki/Mandelbrot_set`

[23] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.c`

[24] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.py`

[25] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot.go`

[26] `https://en.wikipedia.org/wiki/Netpbm`

Fonte	Segundos	Pontos Convergentes
thread 1	0.022489s	5406
thread 2	0.022460s	5525
thread 3	0.112023s	219960
thread 4	0.112285s	220777
mono	0.116688s	-
proc	0.271308s	-

Tabela 3.1. Dados de tempo e pontos convergentes para o programa Mandelbrot em C com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

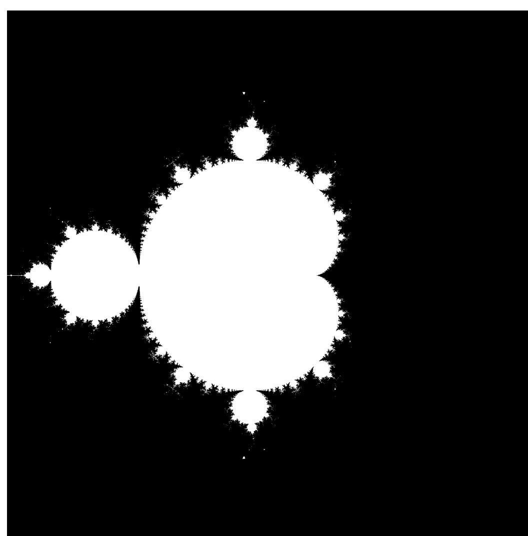


Figura 3.6. Exemplo de imagem gerada pelos programas Mandelbrot

Um modo portátil para ter acesso a um conjunto útil de fontes é por meio da função `clock_gettime`[27], disponível em sistemas POSIX.

Sistemas GNU/Linux costumam disponibilizar 3 fontes de interesse para análise de programas: CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_-THREAD_CPUTIME_ID. Elas representam, respectivamente, uma fonte de tempo que incrementa de forma consistente, uma fonte de tempo que incrementa de forma consistente sempre que o processo está em execução e uma fonte de tempo que incrementa de forma consistente sempre que a *thread* que invocou a função está em execução.

A versão em C do programa faz uso das três fontes de tempo diferentes e escreve na saída padrão de erro a duração de execução para cada *thread*, para o processo como um todo (proc) e o tempo decorrido no mundo real (mono). No caso das *threads*, também é escrito a quantidade de pontos dentre os processados por cada *thread* que convergiram.

A Tabela 3.1 apresenta um exemplo dos dados gerados para uma execução utilizando 4 *threads* em um computador 4-core comum para gerar a Figura 3.6. É possível

[27] `https://linux.die.net/man/3/clock_gettime`

Otimização	Média (compile)	Média (runtime)
-O0	0.04s	0.14s
-O1	0.07s	0.064s
-O2	0,07s	0,060s
-O3	0,07s	0,060s

Tabela 3.2. Tempos de compilação e execução para a versão em C do programa

observar que o tempo de execução do programa (mono) é dominado em boa parte por duas das *threads*, enquanto as outras completam suas partes da imagem antes, indicando uma má distribuição da carga de trabalho. Como cada *thread* processa uma quantidade similar de pontos, a diferença de carga está na quantidade de iterações para determinar se um ponto diverge ou não.

3.5.3. Impacto de flags no *runtime* e *compile time*

Um forma de otimizar a execução de um programa é através da alteração das *flags* de otimização passadas ao compilador. Geralmente, ao ativar uma *flag* acontece uma troca, onde tenta se reduzir o tempo de execução de um programa por meio de um maior tempo de análise e transformações durante a compilação.

Todavia, a tentativa de reduzir o tempo de execução nem sempre é alcançada, por isso é importante analisar o impacto do uso de *flags* tanto no tempo de execução quanto no de compilação por ser possível que o tempo extra gasto na compilação não seja vantajoso em determinadas partes do estudo.

A Tabela 3.2 apresenta o tempo de compilação e execução médio com 10 amostras para as flags -O0, -O1, -O2 e -O3 e com os mesmos argumentos de execução da Tabela 3.1 em um computador 4-core comum utilizando o compilador *Clang* (a variância foi omitida pois todas foram inferiores a 10^{-4}). É possível observar que o melhor tempo de execução está com as otimizações -O2 e -O3.

É importante ressaltar que o espaço de busca por *flags* adequadas vai muito além das otimizações padrão, pois certas combinações de *flags* fora do padrão -O"podem ser benéficas, ou mesmo levar a erros na execução. Quando se pensa em desempenho uma busca pelas melhores *flags* não pode ser descartada, mas só lembrando o espaço de busca pode ser exponencial. Para o leitor interessado sugerirmos um artigo clássico [Hoos 2012] e um mais recente que mostra o potencial de uso de *flags* para GPUs [Bruel et al. 2017].

3.5.4. Compiladores e interpretadores alternativos

Uma outra alternativa para otimizar o programa é através do uso de compiladores ou interpretadores alternativos. Eles podem explorar otimizações não implementadas no projeto principal ou utilizarem como alvo aceleradores disponíveis no sistema, como GPUs ou TPUs.

Um caso em que é comum observar uma grande diferença nos números observados é em linguagens interpretadas como no caso do Python. Utilizando a versão em Python do programa e os mesmos argumentos da Tabela 3.1 (exceto pela omissão dos

workers, já que esta é uma versão sequencial), com o interpretador padrão da linguagem (Python 3.10.14), para 10 amostras, foi observado um tempo de execução médio de 7.43 segundos com variância de 0,04 e com o interpretador alternativo (Pypy 7.3.9), para 10 amostras, foi observado um tempo de execução médio de 0,53 segundos com variância de 0.00021. Esse salto acontece pelo uso da técnica de compilação *just-in-time* utilizada pelo Pypy.

3.5.5. Considerações sobre interpretadores *just-in-time*

Uma forma de se aumentar o desempenho de linguagens interpretadas como Python, ou Julia é o uso da compilação *just-in-time* (JIT). Usualmente, o código é interpretado para ser executado e o tempo gasto com a interpretação pode ser considerável. Algo semelhante acontece em Java, onde após a pré-compilação em bytecode (javac), o código é interpretado (java), dessa forma os processos de compilação e execução ficam explícitos.

Em casos que a perca de tempo interpretando é grande o bastante, a técnica do JIT se torna vantajosa já que o tempo gasto na compilação em tempo de execução é compensado pelo ganho de desempenho na execução do código em si. Porém é sempre bom lembrar as questões ligadas ao *warm-up*, ou aquecimento, pois ao menos a primeira execução será mais lenta. Logo, assim como em experimentos podemos cortar os valores extremos (*outliers*), os valores iniciais podem ser ignorados.

Vale lembrar que também pode ocorrer um efeito contrário, de uma aceleração artificial de programas após as primeiras execuções. Isso geralmente ocorre pois os níveis de *cache* já podem conter dados relevantes para às execuções seguintes. Nesses casos, para uma análise de desempenho justa é importante limpar as memórias *cache*. Isso pode ser feito facilmente alternando diferentes programas a serem medidos.

3.5.6. *Profiling*

Profiling é uma forma de análise dinâmica que visa medir diversos aspectos de um programa. Essa análise pode ser realizada com pouco impacto no desempenho do programa ao utilizar funcionalidades disponíveis em processadores. Para análises mais completas ou em sistemas que não possuem suporte de hardware, é possível utilizar técnicas de virtualização que, apesar de reduzirem o desempenho do programa durante o processo, permitem uma análise mais detalhada do comportamento do mesmo.

A linguagem Go provê um ambiente prático por incluir diversas ferramentas para analisar o comportamento do programa, como a *go tool pprof*. A versão `mandelbrot_prof.go`[28] possui as alterações necessárias no código da versão em Go do programa para realizar a coleta dos dados para a ferramenta. Com a alteração, apesar de se comportar como o programa anterior, a versão modificada gera um arquivo `cpu.prof` que contem os detalhes da execução do programa. Como a ferramenta é baseada na coleta de amostras em curtos intervalos do programa, execuções curtas (menos de 5 segundos) podem não trazer todo o detalhe desejado e com isso, uma possibilidade é aumentar a duração da tarefa a ser realizada.

[28] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_prof.go`

flat	flat%	sum%	cum	cum%	node
58.11s	96.67%	96.67%	59.04s	98.22%	main.work
0.93s	1.55%	98.22%	0.93s	1.55%	runtime.asyncPreempt
0.73s	1.21%	99.43%	0.73s	1.21%	runtime.memmove
0	0%	99.43%	0.38s	0.63%	fmt.(*buffer).writeString
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).fmtS
0	0%	99.43%	0.38s	0.63%	fmt.(*fmt).padString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).doPrintln
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).fmtString
0	0%	99.43%	0.38s	0.63%	fmt.(*pp).printArg
0	0%	99.43%	0.39s	0.65%	fmt.Fprintln

Tabela 3.3. `top` gerado pelo pprof para o programa Mandelbrot em Go adaptado com os argumentos: largura = 28000; altura = 28000; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 100; workers = 12

A Tabela 3.3 foi gerada ao utilizar o comando `top` do *go tool pprof*. Ela apresenta os 10 nós que representam o maior tempo de execução do programa. A coluna *flat* representa o tempo executando instruções dentro da função (nó) sem contar o tempo executando invocações internas de outras funções e a coluna *cum* conta o tempo dessas invocações internas. Como podemos observar, boa parte do tempo do programa é gasto determinando se os pontos divergem ou não (main.work) e o tempo gasto para escrever a imagem na saída padrão é ínfimo (fmt.(*buffer).writeString). Assim, o programa possui apenas uma área cuja otimizações e melhorias impactariam de forma significativa o desempenho do programa.

Devido a simplicidade do programa utilizado, o *profiling* acaba sendo uma técnica que não traz muito mais valor do que uma simples análise do código, porém em projetos maiores, as métricas geradas podem auxiliar no processo de encontrar as partes mais importantes a serem otimizadas.

3.5.7. Soma de verificação

Um ponto crucial que não pode ser esquecido é o fato que desempenho deixa de ser relevante quando o resultado obtido não é o esperado. Ou seja, antes de se falar em melhorar o desempenho é essencial sempre garantir (de preferência por meio de testes automatizados) a corretude do programa. Talvez o exemplo mais conhecido foi o erro de divisão de ponto flutuante dos primeiros processadores Pentium [29].

Entretanto, para saídas muito grandes, não é viável sempre comparar uma saída completa. Para evitar isso, uma forma é usar algo como uma função de *hash*, permitindo que seja necessário guardar e comparar apenas pequenas strings no lugar de de toda a saída do programa. Além de simplificar o processo de validar a saída de um programa, ela pode reduzir o tempo de escrita, pois os algoritmos de *hash* podem funcionar como dispositivos de escrita que não exigem chamadas de sistema. A versão `mandelbrot_-`

[29] `https://en.wikipedia.org/wiki/Pentium_FDIV_bug`

Versão	Tempo médio	Linhas de Código
Go	0.064s	80
C	0.068s	145
Python (Pypy)	0.562s	44
Python (CPython)	7.421s	44

Tabela 3.4. Tempo médio de 10 execuções dos programas com os argumentos: largura = 1600; altura = 1600; xr = -1.5; xi = -1.5; yr = 1.5; yi = 1.5; máximo de iterações = 50; workers = 4

`sha.go`[30] possui as alterações necessárias para utilizar a função SHA-256[31] como função dispositivo de saída.

Outro ponto importante é que guardar a saída do programa pode ajudar a detectar erros de natureza não-determinística, como condições de corrida.

3.5.8. Diferença de desempenho entre linguagens

Como um assunto extra, a Tabela 3.4 apresenta o tempo médio de execução dos diferentes programas para uma mesma entrada (utilizando 4 *workers* nas versões em C e Go). É possível observar uma separação considerável entre as versões de linguagens compiladas e a versão em Python. Mesmo utilizando um interpretador JIT, a versão em Python continua uma ordem de grandeza mais lenta que as versões em C e em Go.

Ao comparar C e Python é possível argumentar que, apesar do pior desempenho, o programa possui um menor número de linhas de código. Porém, a versão em Go conseguiu apresentar o mesmo desempenho que a versão em C e quase metade das linhas de código, ao mesmo tempo que provê funcionalidades como Coletor de Lixo e CSP[32] na linguagem. Uma possível razão para Go ter apresentado um desempenho um pouco melhor que C se dá pela linguagem fazer uso de *goroutines*[33], que apresentam um overhead menor que as *threads* do *pthreads*.

Os programas executados na Parte 2 rodaram em MacBook Pro (15", 2019) com um processador Intel I7-9750H 6-core, 16 GB 2400 MHz DDR4 e rodando o MacOS 12.2.1 com o compilador C Apple clang version 13.0.0, compilador Go go1.17.8 e interpretadores Python 3.9.10 e PyPy 7.3.8.

3.6. Conclusão

Nesse texto apresentamos diversas técnicas de medir desempenho e de mostrar os resultados com relevância estatística. O nosso objetivo foi mostrar que é possível sistematizar a análise de desempenho. Além disso, o desempenho depende de diversos fatores, entre eles a linguagem de programação e/ou o compilador usado. Esperamos que esse texto

[30] `https://github.com/elisauhura/ERAD-benchmarks/blob/main/Parte2/mandelbrot_sha.go`
[31] `https://pkg.go.dev/crypto/sha256`
[32] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[33] https://go.dev/doc/effective_go#goroutines

sirva como um começo de uma jornada!

Referências

[Bruel et al. 2017] Bruel, P., Amaris Gonzalez, M., and Goldman, A. (2017). Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation Practice and Experience*, 29.

[Goldman et al. 2022] Goldman, A., Uhura, E., and Bruschi, S. (2022). Coisas para saber antes de fazer o seu próprio *Benchmarks Game*. In Lorenzon, A., Castro, M., and Pillon, M., editors, *Minicursos da XXII Escola Regional de Alto Desempenho da Região Sul*.

[Hoos 2012] Hoos, H. H. (2012). Programming by optimization. *Commun. ACM*, 55(2):70–80.

[Jain 1991] Jain, R. (1991). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York.

[Kalibera and Jones 2013] Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, page 63–74, New York, NY, USA. Association for Computing Machinery.