CENTRO UNIVERSITÁRIO FEI

LEON FERREIRA BELLINI

22.218.002-8

GUILHERME ORMOND SAMPAIO

22.218.007-7

THREADING E *SPEEDUP*, UM ESTUDO SIMPLES

São Bernardo do Campo

2021

SUMÁRIO

1 Motivação

A criação deste pseudo *profiler* dá-se por conta da necessidade de **automatizar** o estudo do comportamento do problema apresentado anteriormente (ocorrências de números primos num arquivo com duzentos e cinquenta mil números). Basicamente, foi pedido que testes sobre uma quantidade de *threads* seja feito **cinquenta vezes** de forma que seja possível retirar a média destes tempos obtidos para que, por fim, sejam utilizados nos cálculos ou gráficos necessários.

1.1 Como este relatório é estruturado

O *core* do programa foi escrito em C++, compilado com o *GNU C compiler* com as devidas bibliotecas importadas, no caso, apenas `pthread`. O comando de compilação pode ser encontrado no Makefile, este podendo ser encontrado na raiz do projeto.

O *wrapper* para contagem dos tempos é escrito em **Common Lisp**, compilado e executado a partir da implementação **SBCL**.

2 O programa principal

O predicado para a definição de um número como primo ou não foi retirado de learnprogramo, tal função tendo a seguinte estrutura:

```
1  bool isPrime(int n)
2  {
3      // extracted from:
4      // https://learnprogramo.com/prime-number-program-in-c-plus-plus/
5
6      // Corner cases
7      if (n <= 1)
8          return false;
9      if (n <= 3)
10         return true;
11     // This is checked so that we can skip
12     // middle five numbers in below loop
13     if (n % 2 == 0 || n % 3 == 0)
14         return false;
15     for (int i{5}; i * i <= n; i += 6)
16         if (n % i == 0 || n % (i + 2) == 0)
17             return false;
18     return true;
19 }
```

Apesar de não ser tão complexa quanto os outros diversos crivos para "detecção"de números primos, o fato desta utilizar **o quadrado** de *i* até *n* agiliza o processo quando comparado com os algoritmos onde o passo é incrementado de maneira unitária.

2.1 Entrada

O executável tratará de certas *flags* as quais o auxiliam distinguir entre a execução por meio do *wrapper* ou por um usuário comum. Tal entrada tem o seguinte formato:

```
./Primos --output=<0 (apenas tempo) ou 1(tempo + resultado)>
\ --qt-threads=<0 ate n>
```

Logo, se o usuário deseja um comportamento *single-threaded*, basta invocar o executável com os seguintes parâmetros:

```
1 ./Primos --output=1 --qt-threads=1
```

6604.2

121300

Onde a primeira linha trata-se do tempo em **milissegundos** e a segunda a quantidade de números primos encontrados dentre a base carregada.

3 O *wrapper*

Como forma de diminuir o ruído na coleta de dados, a equipe decidiu que a contagem do tempo deve ocorrem em relação ao procedimento de soma de números primos, ao invés de **considerar o programa inteiro para tal cálculo**.

A seguir será detalhado o processo adotado para obtenção dos dados relacionados a **Speedup** e tempos médios das execuções. O leitor pode pular para 3.3 se estiver apenas interessado nos resultados finais.

Primeiramente é definido nome do programa e *string* de controle dinamicamente com o intuito de facilitar a modificação destas com a evolução do projeto:

```
1 (defvar *program-name* "Primos")
2 (defvar *program-control-string* "./~a --qt-threads=~a --output=~a")
```

O procedimento para a execução em si do programa é simples e sinaliza um erro caso este não exista no diretório atual.

```
1 (defun execute-program (&optional n)
2   "Executes the program defined at top level, while checking if it exists.
3 It will always execute said program while passing ``n'' as an argument and 0 in
4 order to silence its output"
5   (if (uiop:file-exists-p *program-name*)
6       (read-from-string
7        (uiop:run-program (format nil *program-control-string* *program-name* n 0)
8                          :output :string))
9     (error "You should compile Main.cpp")))
```

3.1 Cálculo da média

Como sugerido nas instruções do projeto, o programa será executado cinquenta vezes a fim de obter uma média simples em relação a todas as execuções. Para que seja possível um controle maior, o número de execuções será customizável.

```
1 (defparameter *total-runs* 50)
```

O cálculo da média ocorre em `get-average`.

```
1 (defun get-average (runs &optional (n 1))
2   "Runs the program x times (runs) and returns the average runtime (sum/runs).
3 ``n'' refers to the number of threads in which our program must execute on,
4 defaults to 1."
5   (loop for i
6         from 0 below runs
7         sum (execute-program n) into result
8         finally (return (/ result runs))))
```

3.2 Cálculo do *Speedup*

Partindo do cálculo simples do *Speedup* onde $Speedup = \frac{T_1}{T_n}$, podemos escrever a seguinte função:

```
1 (defun get-speedup (serial-time threaded-time)
2   "Returns the speedup (``serial-time''/``threaded-time'') of a process/job."
3   (/ serial-time threaded-time))
```

3.3 Coletando os dados das execuções

Alguns fatores devem ser, primeiramente, expostos a fim de tornar este experimento o mais transparente possível.

a) O experimento será executado numa CPU **quad-core**, ou seja, num melhor caso quatro *threads* do programa ocuparão as CPUs antes de uma troca de contexto. Não há garantia que o aumento de *threads* com $n > 4$ aumente as chances do programa estar em uma das CPUs.

b) Como maneira de se autopreservar, sistemas diminuem as frequências de suas CPUs como forma de diminuir a temperatura para um valor que não seja danoso. Logo, alguns valores podem se apresentar como maiores do que deveriam nas iterações mais avançadas.

O grupo definiu **dez threads** como limite, gerando 50×10 execuções.

```
1 (defparameter *thread-count* 10)
```

Após terminação, será produzida uma tabela cujas colunas representam:

a) Número de *threads n*

b) Tempo em milissegundos da execução

c) *Speedup*

As linhas sendo produzidas pela seguinte função:

```
1  (defun get-row (n &key serial)
2    "Returns a single row of our table.
3  Having the number of threads to execute the main program being ``n'',
4  this procedure will return a list with the following structure:
5
6  (number-of-threads (average-time . speedup))
7
8  The optional keyword argument, ``serial'' tells us that the serial time was
9  supplied, otherwise the average time will be divided by itself in order to
10 obtain speedup info"
11   (let* ((time (get-average *total-runs* n))
12          (speedup (get-speedup (or serial time) time)))
13     (list n time speedup)))
```

E por fim, a função "principal".

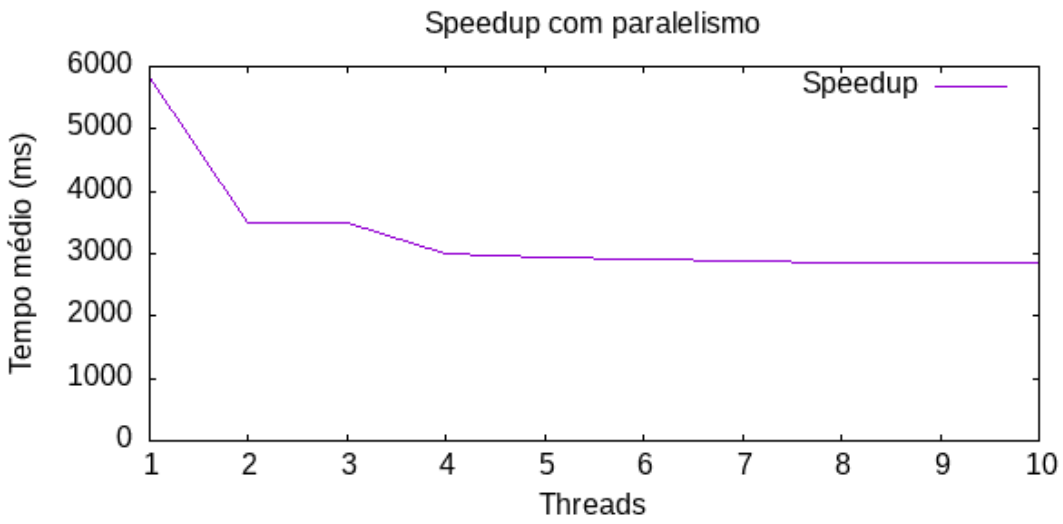
```
(defun produce-table ()
  "Produces the main table to be read by org."
  (let* ((serial-row (get-row 1))
         (serial-time (second serial-row))
         (parallel-table (loop for n from 2 to *thread-count*
                               collect (get-row n :serial serial-time ))))
    (cons serial-row parallel-table)))
```

Após duas horas de execução, foram obtidos os seguintes resultados:

```
(produce-table)
```

Threads	Tempo médio (ms)	Speedup
1	5831.78	1.0
2	3496.3193	1.6679769
3	3483.8306	1.6739562
4	3004.3293	1.9411253
5	2950.7893	1.9763457
6	2926.4277	1.9927982
7	2885.4094	2.0211272
8	2855.3257	2.0424218
9	2849.0715	2.0469053
10	2839.9644	2.0534694

3.4 O gráfico gerado



4 Discussões e conclusão

Como detalhado pela lei de **Amdahl**, a estabilização do fator de *Speedup* é esperada, como pode se notar ao se observar o gráfico, ocorrendo em torno do valor de 2.04, o que é animador, uma vez que tal valor nos indica uma diminuição para a metade do tempo de processamento do problema. Entretanto, devido a esta estabilização, o recomendado seria, então, manter o programa em sete ou oito *threads*, mesmo que ainda ocorra uma melhora diminuta com nove e dez *threads*. Pode-se imaginar que a tendência do problema é manter-se no intervalo $]1.9, 2.1[$ com mais *threads* com a possibilidade de piora, uma vez que a divisão do problema para dezenas ou centenas de *threads* pode incitar uma "concorrência"desnecessária entre os processos durante escalonamento.