Computação de Alto Desempenho – COC472

Trabalho 02 – Perfilagem de Código

Lucas Tavares Da Silva Ferreira

Engenharia de Computação e Informação - Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil

GitHub: https://github.com/lucastavarex/hpc

Introdução

O objetivo deste trabalho é realizar a otimização de um programa que soluciona sistemas de equações lineares pelo método iterativo de Jacobi. O método de Jacobi é um algoritmo iterativo utilizado para encontrar a solução de sistemas lineares. O programa fornecido implementa esse método e permite a resolução de sistemas de equações lineares com matrizes de diferentes tamanhos.

Neste trabalho, seguiremos alguns passos para otimizar o desempenho do programa Os passos incluem a compilação e execução do programa para diferentes tamanhos de matriz, a recompilação e execução com otimização automática, a análise de desempenho usando ferramentas como o gprof ou o Intel VTune, a identificação de gargalos de desempenho por meio da análise de perfil e a aplicação de ações de otimização com base nessas informações.

O objetivo final é comparar os tempos de execução das diferentes versões do programa (inicial, com otimização automática e após a otimização identificada na análise de perfil) e avaliar se as otimizações resultaram em melhorias significativas no desempenho.

Task 01 - Compile e execute o programa para pelo menos 3 tamanhos de matrizes e escolha 1 tamanho para prosseguir.

Seguiremos os seguintes passos:

Compilaremos o código fornecido usando o GCC.
Executaremos o programa com diferentes tamanhos de matriz: 500, 1000 e 2000
Tomaremos os tempos de execução obtidos para cada tamanho de matriz.

n=500

```
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ ./jacobi -n 500
------------------------------------
Matrix size:            500x500
Maximum iterations:     20000
Convergence threshold:  0.000100
------------------------------------
Solution error = 0.024952
Iterations     = 1465
Total runtime  = 4.224211 seconds
Solver runtime = 4.205202 seconds
------------------------------------
```

n=1000

```
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ ./jacobi
------------------------------------
Matrix size:            1000x1000
Maximum iterations:     20000
Convergence threshold:  0.000100
------------------------------------
Solution error = 0.050047
Iterations     = 2957
Total runtime  = 43.852504 seconds
Solver runtime = 43.789850 seconds
------------------------------------
```

n=2000

```
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ ./jacobi -n 2000
------------------------------------
Matrix size:            2000x2000
Maximum iterations:     20000
Convergence threshold:  0.000100
------------------------------------
Solution error = 0.099977
Iterations     = 5479
Total runtime  = 353.672394 seconds
Solver runtime = 353.398447 seconds
------------------------------------
```

Seguiremos com o estudo com n = 2000.

Task 02 - Recompile e execute o programa com otimização automática.

Seguiremos os seguintes passos:

Tomaremos o código-fonte.
Adicionamos a opção de compilação para otimização automática ao comando de compilação. Por exemplo, usando a opção "-O3" no GCC.
Salvamos o arquivo modificado.
Recompilamos o código usando o comando de compilação atualizado.
Executamos o programa com o mesmo tamanho de matriz usado no passo anterior.
Comparamos os tempos de execução com a versão anterior.

```
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ ./jacobi_03_opt -n 2000
---------------------------------
Matrix size:            2000x2000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------
Solution error = 0.099977
Iterations     = 5479
Total runtime  = 285.805131 seconds
Solver runtime = 285.541394 seconds
---------------------------------
```

Comparando com o valor obtido anteriormente, tivemos uma melhora de 19.15%
((353.398447 - 285.541394) / 353.398447) * 100 = 19.15%

Task 03 - Execute a perfilagem do programa usando o gprof ou o Intel VTune.

Seguiremos os seguintes passos:

Com o gprof configurado, o utilizaremos para perfilar o programa e obter informações sobre o tempo de execução.
Compilaremos novamente o código com a opção de compilação "-pg" para permitir a geração de informações.
Após a conclusão da execução, o gprof gerará um arquivo de saída que contém os resultados da perfilagem, chamado ‘gmon.out’.
Analise o arquivo de saída para identificar quais partes do código estão consumindo mais tempo de execução.

```
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ gcc -pg -o jacobi_gprof jacobi.c -lm
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ ./jacobi_gprof -n 2000
---------------------------------
Matrix size:            2000x2000
Maximum iterations:     20000
Convergence threshold:  0.000100
---------------------------------
Solution error = 0.099977
Iterations     = 5479
Total runtime  = 364.476966 seconds
Solver runtime = 364.172755 seconds
---------------------------------
lucastavares@pop-os:~/Desktop/HPC/hpc-trabalhos/hpc/TASK2$ gprof jacobi_gprof gmon.out > analysis.txt
```

Resultado do arquivo analysis.txt:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.97	364.11	364.11	1	364.11	364.11	run
0.03	364.22	0.11				main
0.00	364.22	0.00	4	0.00	0.00	get_timestamp
0.00	364.22	0.00	1	0.00	0.00	parse_arguments
0.00	364.22	0.00	1	0.00	0.00	parse_int

 %   the percentage of the total running time of the
time   program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self   the number of seconds accounted for by this
seconds   function alone. This is the major sort for this
           listing.

calls   the number of times this function was invoked, if
           this function is profiled, else blank.

 self   the average number of milliseconds spent in this
ms/call   function per call, if this function is profiled,
           else blank.

 total   the average number of milliseconds spent in this
ms/call   function and its descendents per call, if this
           function is profiled, else blank.
name   the name of the function. This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.00% of 364.22 seconds

```
index % time    self  children    called     name
                                              <spontaneous>
[1]    100.0    0.11  364.11                 main [1]
             364.11    0.00       1/1          run [2]
               0.00    0.00       4/4          get_timestamp [3]
               0.00    0.00       1/1          parse_arguments [4]
-----------------------------------------------
             364.11    0.00       1/1          main [1]
[2]    100.0  364.11    0.00       1          run [2]
-----------------------------------------------
               0.00    0.00       4/4          main [1]
[3]      0.0    0.00    0.00       4          get_timestamp [3]
-----------------------------------------------
               0.00    0.00       1/1          main [1]
[4]      0.0    0.00    0.00       1          parse_arguments [4]
```

```
                0.00    0.00       1/1          parse_int [5]
-----------------------------------------------
                0.00    0.00       1/1          parse_arguments [4]
[5]      0.0    0.00    0.00         1        parse_int [5]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index — A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

% time — This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self — This is the total amount of time spent in this function.

children — This is the total amount of time propagated into this function by its children.

called — This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name — The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self — This is the amount of time that was propagated directly from the function into this parent.

children — This is the amount of time that was propagated from the function's children into this parent.

called — This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[3] get_timestamp [4] parse_arguments [2] run
[1] main [5] parse_int

Task 04 - Analise o resultado da perfilagem.

Seguiremos os seguintes passos:

Com base nos resultados da perfilagem, identificamos os trechos de código que consomem a maior parte do tempo de execução.
Verificamos se há laços ou operações computacionalmente intensivas que possam ser otimizadas.
Identificamos as oportunidades de melhoria com base nas informações fornecidas pela análise de desempenho.

Task 05 - De posse dos resultados de sua análise, tente 1 ação de otimização.

Seguiremos os seguintes passos:

Com base nas informações obtidas na análise de desempenho, otimiza-se o trecho de código identificado como gargalo de desempenho.
Isto pode envolver otimizações como redução de operações redundantes, exploração de paralelismo, utilização de instruções vetoriais, entre outras técnicas.
Faremos as alterações necessárias no código e o compilaremos novamente.
Executaremos o programa com o mesmo tamanho de matriz para verificar se a otimização teve um impacto positivo nos tempos de execução.

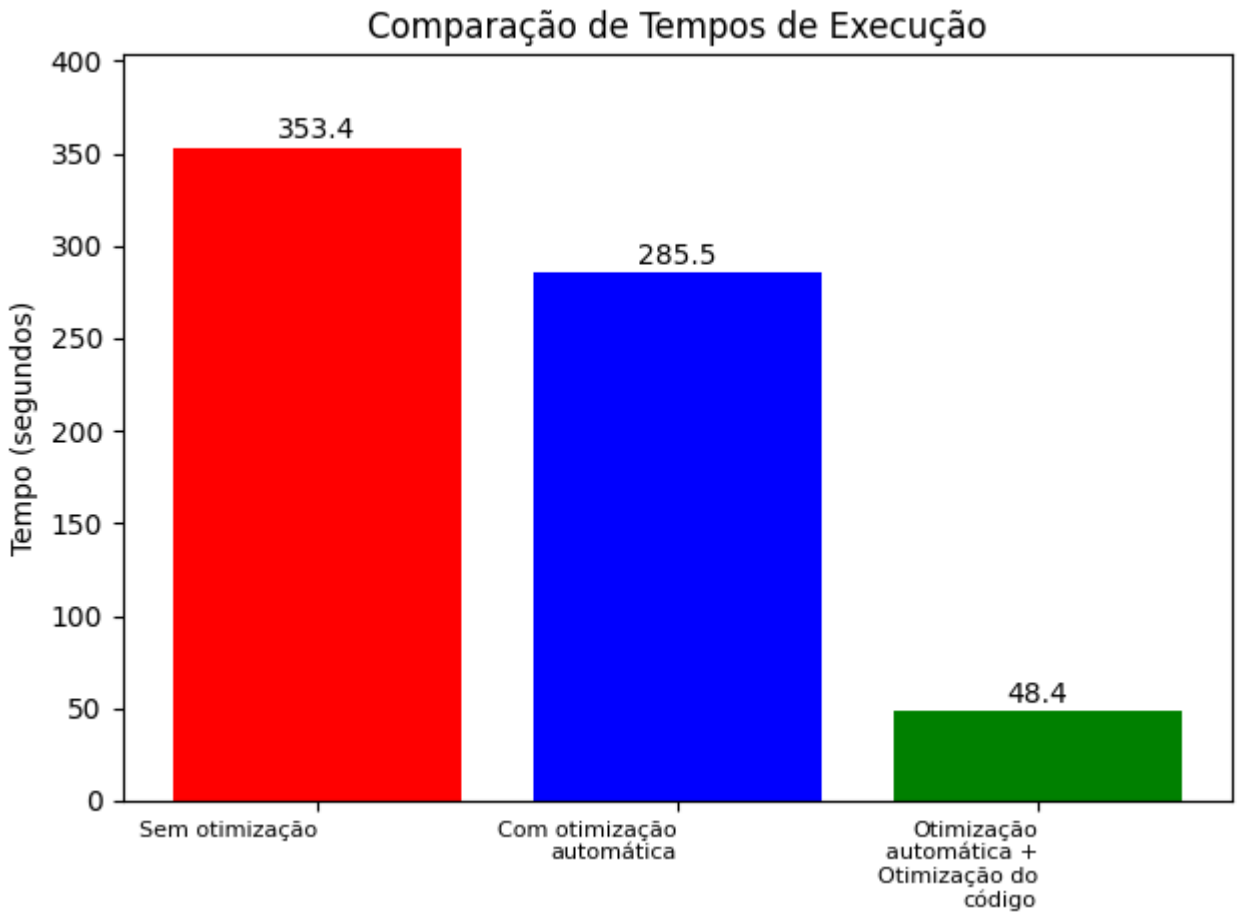
O código optimizado se encontra no meu diretório do GitHub:
https://github.com/lucastavarex/hpc/blob/main/TASK2/jacobi_opt.c

Tivemos uma melhora enorme com a optimização do código.

Task 06 - Compare os tempos das versões.

Seguiremos os seguintes passos:

Comparamos os tempos de execução obtidos nas diferentes versões do programa: a versão inicial, a versão com otimização automática e a versão após a aplicação da otimização identificada na análise de desempenho.
Verificamos se houve melhorias significativas nos tempos de execução após as otimizações.



Com isso, percebemos que de fato houve uma grande melhora na performance do código.