Profiling

Bruno da Silva Machado[1*]

[1] Curso de Bacharelado em Física com ênfase em Física Computacional,
Instituto de Ciências Exatas, Universidade Federal Fluminense,
27213-145 Volta Redonda - RJ, Brasil

November 3, 2018

Resumo

Neste artigo vamos efetuar o profiling de dois programas escritos na linguagem C. O primeiro é usa o método da relaxação para descrever um potencial elétrico ao redor de um para-raio, enquanto o segundo programa usa o método de Jacobi para a solução de sistemas de equações lineares. Fazer o profiling dos programas é muito importante pois permite reunir informações sobre o comportamento de um programa, isto é o uso dos recursos do mesmo. e não só isso auxilia na otimização do código pois nos mostra quais funções são as mais utilizadas e se é possível otimiza-las.

1 Descrição dos programas

O primeiro programa consiste em solucionar o problema do para-raio que consiste em descobrir o comportamento do potencial elétrico ao redor de uma para-raio. Este problema é descrito pela equação de Laplace, esta por sua vez pode ser solucionado pelo método da relaxação.

O arquivo que possuí o código deste problema é o "relaxacaoPeriodo.c"e a versão otimizada do código é a "relaxacaoPeridoOtimizado.c".

Dentro dos arquivos possui as seguintes funções: *void contorno()* responsável de gerar as condições de contorno na matriz de relaxação. O *void relaxacao()* função responsável de fazer a relaxação da matriz é assim solucionar a equação de laplace, *void periodo()* adiciona e faz a manutenção das condições de periodicidade da matriz de relaxação, o *void imprime()* imprime os dados da matriz dentro de uma arquivo de texto. *void trs()* é a função responsável pela condição de parada que consiste em verificar se o traço da matriz é menor que um certo valor ϵ. Por fim a *int main()* é a entrada do programa e é o local onde as demais funções são chamadas.

*brunosilvamachado@id.uff.br

O segundo programa consiste em ler um arquivo de texto com um sistema de equações lineares de quantidade de incógnitas arbitraria e soluciona -lo através do método de Gauss.

O arquivo que possuí o código deste problema é o "metodoGauss.c"e a versão otimizada do código é a "metodoGaussOtimizado.c".

Dentro dos arquivos possui as seguintes funções: O *double * substituicaoRegressiva()* usada neste programa para calcular as raízes do sistema de equações, o *int triangularSuperior_p()* reduz a matriz para a forma de triangular superior com pivotamento entre as linhas. *double **lerMatrizCompleta()* recebe o nome do arquivo contendo o sistema de equações lineares e carrega para a memoria para ser usada pelo programa, *void imprimeMatrizCompleta()* e *void imprimeRaiz()* imprime a matriz informada e as soluções dos sistemas lineares respectivamente. Por fim a *int main()* é a entrada do programa e é o local onde as demais funções são chamadas.

2 O profiling

2.1 Programa 1

2.1.1 Profiling sem otimização

A tabela 1 é o resultado do profiling do programa sem otimizações do compilador ou de código.

Tabela 1: Profiling do programa 1 sem otimização

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
99.92	93.83	93.83	72340	1.30	1.30	relaxacao
0.34	94.15	0.32	144680	0.00	0.00	trs
0.27	94.40	0.25	72340	0.00	0.00	periodo
0.00	94.40	0.00	1	0.00	0.00	contorno
0.00	94.40	0.00	1	0.00	0.00	imprime

Na tabela 1 temos as seguintes colunas.A primeira coluna informa a porcentagem do tempo total de execução do programa usado por esta função, a segunda coluna são os segundo acumulados que é uma soma corrente do numero de segundos contabilizados pelas funções listadas acima,a terceira coluna é o numero de segundos contabilizados por esta função sozinha. Este é o tipo principal da tabela. A quarta coluna é o numero de vezes que a função é invocada, se esta for perfilada, senão fica em branco, a quinta coluna é o valor médio de milissegundos gasto nesta função por chamada, se esta função tiver perfil, senão em branco. A sexta coluna é o numero média de milissegundos gastos neste e em seus descendentes por chamada, se esta função é perfilada, senão em branco, a ultima coluna são os nomes das funções estas estão ordenadas pelo numero de segundos contabilizadas individualmente.

Através deste profile vemos que a função que mais consome tempo de memoria é o *void relaxacao()* com 99.92% do tempo de execução, vale ressaltar que ela é a função que vai solucionar a equação de laplace através do método da relaxação, isso justifica o fato de ser a função que mais tempo fica na memoria, mas o seu tempo de execução sozinho que é extremamente alto nos revela que ela é a fonte de gargalo no programa e é esta que deve ser otimizada para aumentar o desempenho do código.

2.1.2 Profiling com otimização de código

A partir da seção anterior já sabemos que a função que causa o gargalo é o *void relaxacao()*. Nessa seção vamos buscar otimizar o código desta função para isso usaremos uma técnica bem simples que é simplificar se possível as operações aritméticas em termos de soma e produto.

```
1  // metodo antes da otimizacao
2  void relaxacao()
3  {
4      int i,j;
5
6      for(i = 1;i < N - 1; i++)
7      {
8          for(j = 1; j < N - 1;j++)
9          {
10             if(mascara[i][j] == 0)
11             matriz[i][j] = (matriz[i - 1][j] + matriz[i + 1][j
       ] + matriz[i][j - 1] + matriz[i][j + 1])/4.0;
12         }
13     }
14 }
```

```
1  //metodo depois da otimizacao do codigo
2  void relaxacao()
3  {
4    int i,j;
5
6    for(i = 1;i < N - 1; i++)
7    {
8      for(j = 1; j < N - 1;j++)
9      {
10       if(mascara[i][j] == 0)
11       matriz[i][j] = (matriz[i - 1][j] + matriz[i + 1][j] +
       matriz[i][j - 1] + matriz[i][j + 1])*0.25;
12     }
13   }
```

```
14 }
```

Na tabela 2 podemos ver temos os novos resultados após a otimização, lembrando que as colunas desta tabela tem o mesmo significado da tabela 1. Podemos ver na tabela que a função *void relaxacao()* apresenta uma redução de 11% no tempo de execução sozinha, mas ela continua sendo a função que passa maior tempo em execução.

Tabela 2: Profiling do programa 1 com otimização de código

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
99.62	83.44	83.44	72340	1.15	1.15	relaxacao
0.46	83.83	0.38	144680	0.00	0.00	trs
0.43	84.19	0.36	72340	0.01	0.01	periodo
0.02	84.21	0.02				main
0.00	84.21	0.00	1	0.00	0.00	contorno
0.00	84.21	0.00	1	0.00	0.00	imprime

2.1.3 Profiling com otimização de codigo e flags

Por fim com o código devidamente otimizado vamos buscar agora uma melhor performance usando as flags de otimização do compilador. Nesse trabalho compilamos o código usado GNU GCC Compiler versão 5.31 e as flags foram *-O2 -march=corei7-avx*. Podemos ver na tabela 3 um ganho de performance absurdo no programa comparando os valores da tabela 1 o ganho foi de 65% um excelente resultado nos revelando que a otimização do código foi um sucesso

Tabela 3: Profiling do programa 1 com otimização de código + flags -O2 -march=corei7-avx

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
99.42	32.73	32.73	72340	452.45	452.45	relaxacao
0.58	32.92	0.19				main
0.00	32.92	0.00	1	0.00	0.00	imprime

2.2 Programa 2

2.2.1 Profiling sem otimização

Seguindo o o mesmo procedimento da seção anterior a tabela 4 é o resultado do profiling do programa sem otimizações do compilador ou de código.

Na tabela 4 temos as seguintes colunas como na seção anterior. A primeira coluna informa a porcentagem do tempo total de execução do programa usado

Tabela 4: Profiling do programa 2 sem otimização

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
96.83	3.66	3.66	1	3.66	3.66	triangularSuperior_p
2.91	3.77	0.11	2	0.06	0.06	imprimeMatrizCompleta
0.26	3.78	0.01	1	0.01	0.01	lerMatrizCompleta
0.00	3.78	0.00	1	0.00	0.00	imprimeRaiz
0.00	3.78	0.00	1	0.00	0.00	substituicaoRegressiva

por esta função, a segunda coluna são os segundo acumulados que é uma soma corrente do numero de segundos contabilizados pelas funções listadas acima,a terceira coluna é o numero de segundos contabilizados por esta função sozinha. Este é o tipo principal da tabela. A quarta coluna é o numero de vezes que a função é invocada, se esta for perfilada, senão fica em branco, a quinta coluna é o valor médio de milissegundos gasto nesta função por chamada, se esta função tiver perfil, senão em branco. A sexta coluna é o numero média de milissegundos gastos neste e em seus descendentes por chamada, se esta função é perfilada, senão em branco, a ultima coluna são os nomes das funções estas estão ordenadas pelo numero de segundos contabilizadas individualmente.

Neste profile vemos que a função que mais consome tempo de memoria é o *void triangularSuperior_p()* com 96.83% do tempo de execução, Esta é justamente a função responsável por transformar a matriz quadrada em uma triangular e demorasse um tempo considerável na realização da tarefa por estar manipulando matrizes grandes, mas o seu tempo de execução sozinho que é alto em relação aos demais revelando que ela é a fonte de gargalo no programa e é esta que deve ser otimizada para aumentar o desempenho do código.

2.2.2 Profiling com otimização de código

A partir da seção anterior já sabemos que a função que causa o gargalo é o *void triangularSuperior_p()*. Nessa seção vamos buscar otimizar o código desta função para isso usaremos uma técnica bem simples que é a substituição se possível de funções do externas por operações de soma e produto.

```
1  // metodo antes da otimizacao
2  void triangularSuperior_p(double **m, int dim)
3  {
4    int i,j,k,l,troca;
5    double n;
6
7    double *aux;
8
9    aux = (double*)malloc((dim + 1) * sizeof(double));
10
11   for(i = 0; i < dim; i++)
```

```
12    {
13
14      troca = -1;
15      for(l = i; l < dim; l++)
16      {
17        if(m[i][i] < fabs(m[l][i]))
18        {
19          troca = l;
20        }
21      }
22
23      if(troca != -1)
24      {
25        memcpy(aux, m[troca], (dim + 1)*sizeof(double));
26        memcpy(m[troca], m[i], (dim + 1)*sizeof(double));
27        memcpy(m[i], aux, (dim + 1)*sizeof(double));
28      }
29
30      for(j = i + 1; j < dim; j++)
31      {
32        n = m[j][i]/(double)m[i][i];
33
34        for(k = 0; k < dim + 1; k++)
35        {
36          m[j][k] = m[j][k] - n * m[i][k];
37        }
38      }
39    }
40  }
```

```
1  //metodo depois da otimizacao do codigo
2  void triangularSuperior_p(double **m, int dim)
3  {
4    int i,troca;
5    double n;
6
7    double *aux;
8
9    aux = (double*)malloc((dim + 1) * sizeof(double));
10
11   for(i = 0; i < dim; i++)
12   {
13
14     troca = -1;
15     for(int l = i; l < dim; l++)
16     {
17       if(m[i][i]*m[i][i] < m[l][i]*m[l][i])
```

```
18          {
19            troca = l;
20          }
21        }
22
23        if(troca != -1)
24        {
25          memcpy(aux, m[troca], (dim + 1)*sizeof(double));
26          memcpy(m[troca], m[i], (dim + 1)*sizeof(double));
27          memcpy(m[i], aux, (dim + 1)*sizeof(double));
28        }
29
30        for(int j = i + 1; j < dim; j++)
31        {
32          n = m[j][i]/(double)m[i][i];
33
34          for(int k = 0; k < dim + 1; k++)
35          {
36            m[j][k] = m[j][k] - n * m[i][k];
37          }
38        }
39      }
40  }
```

Na tabela 5 podemos ver temos os novos resultados após a otimização, lembrando que as colunas desta tabela tem o mesmo significado da tabela 4. Podemos ver que a função *void triangularSuperior_p()* apresenta uma redução de 40% no tempo de execução sozinha, mas ela continua sendo a função que passa maior tempo em execução.

Tabela 5: Profiling do programa 2 com otimização de código

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
84.56	2.19	2.19	1	2.19	2.19	triangularSuperior_p
15.06	2.58	0.39	2	0.20	0.20	imprimeMatrizCompleta
0.39	2.59	0.01	1	0.01	0.01	lerMatrizCompleta
0.00	2.59	0.00	1	0.00	0.00	imprimeRaiz
0.00	2.59	0.00	1	0.00	0.00	substituicaoRegressiva

2.2.3 Profiling com otimização de codigo e flags

Novamente com o código devidamente otimizado vamos buscar agora uma melhor performance usando as flags de otimização do compilador. Usaremos as mesmas flags de da subseção anterior *-O2 -march=corei7-avx*. Podemos ver na tabela 6 um ganho de performance surpreendente no programa comparando os

valores da tabela 4 o ganho foi de 81% um excelente resultado nos revelando a importância de como escolher as flags corretamente nos leva a códigos devidamente otimizados.

Tabela 6: Profiling do programa 2 com otimização de código + flags

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
67.03	0.61	0.61	1	610.00	610.00	triangularSuperior_p
32.97	0.91	0.30	2	150.00	150.00	imprimeMatrizCompleta
0.00	0.91	0.00	1	0.01	0.01	lerMatrizCompleta
0.00	0.91	0.00	1	0.00	0.00	imprimeRaiz
0.00	0.91	0.00	1	0.00	0.00	substituicaoRegressiva