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 2, 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 Jacobi.

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

Dentro dos arquivos possui as seguintes funções: *double norm()* responsável calcular a norma de um vetor de dimensão N. O *double * metodoJacobi()* é um algoritmo clássico para resolver sistemas de equações lineares. Raramente utilizado em sistemas lineares de pequenas dimensões, já que o tempo requerido para obter um minimo de precisão ultrapassa o requerido pelas técnicas diretas como a eliminação gaussiana. Contudo para sistemas grandes, com grande porcentagem de entradas de zeros, essa técnica aparece como uma alternativa mais eficiente. *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

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