Perfilação

Computação de Alto Desempenho

Lucas Ribeiro Ikuhara - DRE 119019172

Considerações iniciais

Ao analizarmos o código, pode-se perceber que se trata de uma aproximação de uma equação de laplace, para que possa ser resolvida usando o método de relaxação. Essa técnica é comumente usada para resolver numericamente equações elipticas. A aplicação analizada é de caráter científico, e pode-se imaginar que pode demorar muito dependendo das entradas. Para melhorar a perfomance da implementação, pode-se usar um perfilador, para descobrirmos quais partes do código gastam mais tempo para serem executadas no total (hotspots), para dirigirmos nossos esforços de otimização.

Passos para perfilação

Para conseguirmos usar o gprof, é necessário que o código seja compilado com os flags `-pg`, e que o executável gerado seja executado pelo menos uma vez. Quando a execução é finalizada, é gerado um arquivo *gmon.out*. Nesse ponto, o perfilador pode ser chamado.

Utilizando o g++ e gprof, o processo de compilação e perfilação pode ser exemplificado assim:

```
$ g++ laplace.cxx -pg -o saida.out
$ ./saida.out
Enter nx n_iter eps --> 500 100 1e-6
$ grof saida.out
```

Resultados da perfilação

O perfilador gera um relatório detalhado com exlicações adicionais, que serão coladas integralmente no final do relatório. Os resultados mais relevantes, no entanto, foram compilados em uma tabela, com o nome das funções, quantidade de chamadas e tempo gasto em cada uma.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.44      0.15     0.15      100     1.51     2.16
LaplaceSolver::timeStep(double)
 29.66      0.22     0.07 24800400     0.00     0.00  SQR(double const&)
  2.28      0.22     0.01        2     2.51     2.51  seconds()
  0.00      0.22     0.00     2000     0.00     0.00  BC(double, double)
```

```
  0.00      0.22     0.00        1     0.00     0.00
_GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.22     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::initialize()
  0.00      0.22     0.00        1     0.00   215.81
LaplaceSolver::solve(int, double)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::LaplaceSolver(Grid*)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::~LaplaceSolver()
  0.00      0.22     0.00        1     0.00     0.00  Grid::setBCFunc(double
(*)(double, double))
  0.00      0.22     0.00        1     0.00     0.00  Grid::Grid(int, int)
```

Ao analizar os resultados explicitados, podemos ver que os dois hotspots principais são as funções `LaplaceSolver::timeStep` e `SQR`, que deverão ser nosso foco ao tentar otimizar o código.

Ao analizarmos o código fonte dessas duas funções, é fácil perceber que `Real LaplaceSolver :: timeStep` não segue boas práticas de CAD. Entretanto, a função `SQR` já se encontra relativamente otimizada e seria dificil conseguir ganhos de perfomance alterando ela.

- Real LaplaceSolver :: timeStep

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;
    int nx = g->nx;
    int ny = g->ny;
    Real **u = g->u;

    for (int i=1; i<nx-1; ++i) {
        for (int j=1; j<ny-1; ++j) {
            tmp = u[i][j];
            u[i][j] = ((u[i-1][j] + u[i+1][j])*dy2 +
                       (u[i][j-1] + u[i][j+1])*dx2)*0.5/(dx2 + dy2);
            err += SQR(u[i][j] - tmp);
        }
    }
```

```
    return sqrt(err);
}
```

Nessa função, podemos imediatamente observar que os loops realizam operações para computar sua condição de parada, que poderia ter sido feitas fora de sua declaração. Além disso, são realizadas divisões custosas no loop que poderiam ter sido evitadas.

Otimizando o código

Resolvendo os problemas analizados acima, tentaremos melhorar a performance do programa. O tempo de execução inicial reportado pelo programa é 0.446201 segundos.

Como anteriormente citado, deve-se evitar operações de divisão custosas dentro de loops; Dessa forma, em `LaplaceSolver :: timeStep`, o trecho `u[i][j] = ((u[i-1][j] + u[i+1][j])*dy2 + (u[i][j-1] + u[i][j+1])*dx2);` foi alterado, e partes dele foram calculadas antes de começar a iteraração. Visando reduzindo a quantidade de multiplicações e divisões por iteração, temos:

```
    Real k = 0.5/(dx2 + dy2);
    dx2 = dx2 * k;
    dy2 = dy2 * k;

    for (int i=1; i<nx-1; ++i) {
        for (int j=1; j<ny-1; ++j) {
            tmp = u[i][j];
            u[i][j] = (u[i-1][j] + u[i+1][j])*dy2 +
                      (u[i][j-1] + u[i][j+1])*dx2;
            err += SQR(u[i][j] - tmp);
        }
    }
```

Após essa alteração, o tempo de execução foi reduzido de 0.446201 segundos para 0.417577 segundos.

Foram testadas mais duas alterações, mas nenhuma delas melhorou a performance. A primeira foi tentar alterar a função `SQR`, alterando o trecho `(x*x)` por `pow(x, 2)`. A ideia por trás dessa mudança é que possivelmente a implementação de exponenciação da biblioteca cmath seria mais rápida do que a operação inicial. essa hipótese foi rapidamente refutada, uma vez que a mudança dobrou o tempo de execução do programa.

A outra alteração testada foi substituir a condição de parada do loop `for (int i=1; i<nx-1; ++i) {` para calcular `nx-1` antes de sua declaração. Esa alteração não aumentou nem diminuiu apreciavelmene o tempo de execução. Como em média testando algumas vezes seguidas o tempo aumentou, mesmo que por muito pouco, a alteração foi descartada.

Além disso, cosiderou-se a possibilidade de usar as diretivas `#pragma` para tentar forçar a vetorização do loop. Entretanto, olhando mais atenciosamente o código, pode-se perceber que as iterações são dependentes uma da outra.

Conclusões finais

É evidente que perfiladores são ferramentas muito poderosas quando se busca extrair mais perfomance. Não só porquê nos dão ferramentas robustas para conduzir testes relacionados a performance, mas porquê tornam o processo como um todo mais objetivo e cientifico, possibilitando que esforços sejam concentrados nas partes mais importantes do código: os hotspots. O ganho total de performance foi de aproximadamente 6.41%, que adimitidamente modesto, não é desconsiderável, e

Saída do perfilador completa

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 68.44      0.15     0.15      100     1.51     2.16
LaplaceSolver::timeStep(double)
 29.66      0.22     0.07 24800400     0.00     0.00  SQR(double const&)
  2.28      0.22     0.01        2     2.51     2.51  seconds()
  0.00      0.22     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.22     0.00        1     0.00     0.00
_GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.22     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::initialize()
  0.00      0.22     0.00        1     0.00   215.81
LaplaceSolver::solve(int, double)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::LaplaceSolver(Grid*)
  0.00      0.22     0.00        1     0.00     0.00
LaplaceSolver::~LaplaceSolver()
  0.00      0.22     0.00        1     0.00     0.00  Grid::setBCFunc(double
(*)(double, double))
  0.00      0.22     0.00        1     0.00     0.00  Grid::Grid(int, 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 2 byte(s) for 4.53% of 0.22 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.22                 main [1]
                0.00    0.22       1/1           LaplaceSolver::solve(int,
double) [3]
                0.01    0.00       2/2           seconds() [5]
```

```
                0.00    0.00       1/1           Grid::Grid(int, int) [19]
                0.00    0.00       1/1           Grid::setBCFunc(double (*)
(double, double)) [18]
                0.00    0.00       1/1
LaplaceSolver::LaplaceSolver(Grid*) [16]
                0.00    0.00       1/1
LaplaceSolver::~LaplaceSolver() [17]
-----------------------------------------------
                0.15    0.07     100/100         LaplaceSolver::solve(int,
double) [3]
[2]     97.7    0.15    0.07     100         LaplaceSolver::timeStep(double)
[2]
                0.07    0.00 24800400/24800400     SQR(double const&) [4]
-----------------------------------------------
                0.00    0.22       1/1           main [1]
[3]     97.7    0.00    0.22       1         LaplaceSolver::solve(int,
double) [3]
                0.15    0.07     100/100
LaplaceSolver::timeStep(double) [2]
-----------------------------------------------
                0.07    0.00 24800400/24800400
LaplaceSolver::timeStep(double) [2]
[4]     29.5    0.07    0.00 24800400         SQR(double const&) [4]
-----------------------------------------------
                0.01    0.00       2/2           main [1]
[5]      2.3    0.01    0.00       2         seconds() [5]
-----------------------------------------------
                0.00    0.00    2000/2000        Grid::setBCFunc(double (*)
(double, double)) [18]
[12]     0.0    0.00    0.00    2000         BC(double, double) [12]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [24]
[13]     0.0    0.00    0.00       1         _GLOBAL__sub_I__ZN4GridC2Eii
[13]
                0.00    0.00       1/1
__static_initialization_and_destruction_0(int, int) [14]
-----------------------------------------------
                0.00    0.00       1/1
_GLOBAL__sub_I__ZN4GridC2Eii [13]
[14]     0.0    0.00    0.00       1
__static_initialization_and_destruction_0(int, int) [14]
-----------------------------------------------
                0.00    0.00       1/1
```

```
LaplaceSolver::LaplaceSolver(Grid*) [16]
[15]     0.0    0.00    0.00       1         LaplaceSolver::initialize()
[15]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[16]     0.0    0.00    0.00       1
LaplaceSolver::LaplaceSolver(Grid*) [16]
                0.00    0.00       1/1           LaplaceSolver::initialize()
[15]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[17]     0.0    0.00    0.00       1         LaplaceSolver::~LaplaceSolver()
[17]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[18]     0.0    0.00    0.00       1         Grid::setBCFunc(double (*)
(double, double)) [18]
                0.00    0.00    2000/2000        BC(double, double) [12]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[19]     0.0    0.00    0.00       1         Grid::Grid(int, int) [19]
-----------------------------------------------
```

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