Trabalho 2 - Computação de Alto Desempenho
Semestre 2020/2
Thiago Guimarães - DRE: 118053123

Introdução

O código visa aproximar a equação de laplace utilizando o método Gauss-Seidel, que é um método iterativo para resolução de sistema de equações lineares, tendo a iteração definida por:

$$x^{(k+1)} = (D+L)^{-1}\left(-Ux^{(k)} + b\right),$$

Neste caso, a perfilagem do código tem grande importância no sentido de ajudar o programador a entender possíveis melhorias que podem ser feitas no código objetivando a melhor performance possível.

Utilizando o gprof

Para perfilar o código C++ utilizando o gprof devem ser realizados os seguintes passos:

- Compilar o código com a flag -pg

Ex: > *g++ laplace.cxx -o laplace*

- Executar o executável gerado, o que gerará um arquivo gmon.out

Ex: > *./laplace*

- Utilizar o gprof passando como argumento o arquivo gmon.out gerado no passo anterior - também é interessante armazenar a saída num arquivo de log (opcional)

Ex: > *gprof gmon.out [> log.txt]*

Relatório gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 90.89      0.82     0.82      100     8.18     8.83  LaplaceSolver::timeStep(double)
  7.25      0.88     0.07 24800400     0.00     0.00  SQR(double const&)
  1.12      0.89     0.01        2     5.02     5.02  seconds()
  0.56      0.90     0.01     2000     0.00     0.00  BC(double, double)
  0.00      0.90     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.90     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.90     0.00        1     0.00     0.00  LaplaceSolver::initialize()
```

```
 0.00      0.90     0.00        1     0.00   883.28  LaplaceSolver::solve(int, double)
 0.00      0.90     0.00        1     0.00     0.00  LaplaceSolver::LaplaceSolver(Grid*)
 0.00      0.90     0.00        1     0.00     0.00  LaplaceSolver::~LaplaceSolver()
 0.00      0.90     0.00        1     0.00     5.02  Grid::setBCFunc(double (*)(double, double))
 0.00      0.90     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 1.11% of 0.90 seconds

```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    0.90                 main [1]
                0.00    0.88       1/1           LaplaceSolver::solve(int, double) [3]
                0.01    0.00       2/2           seconds() [5]
                0.00    0.01       1/1           Grid::setBCFunc(double (*)(double, double)) [7]
                0.00    0.00       1/1           Grid::Grid(int, int) [19]
                0.00    0.00       1/1           LaplaceSolver::LaplaceSolver(Grid*) [17]
```

```
                0.00    0.00       1/1           LaplaceSolver::~LaplaceSolver() [18]
-----------------------------------------------
                0.82    0.07     100/100         LaplaceSolver::solve(int, double) [3]
[2]     98.3    0.82    0.07     100         LaplaceSolver::timeStep(double) [2]
                0.07    0.00 24800400/24800400     SQR(double const&) [4]
-----------------------------------------------
                0.00    0.88       1/1           main [1]
[3]     98.3    0.00    0.88       1         LaplaceSolver::solve(int, double) [3]
                0.82    0.07     100/100         LaplaceSolver::timeStep(double) [2]
-----------------------------------------------
                0.07    0.00 24800400/24800400     LaplaceSolver::timeStep(double) [2]
[4]      7.3    0.07    0.00 24800400         SQR(double const&) [4]
-----------------------------------------------
                0.01    0.00       2/2           main [1]
[5]      1.1    0.01    0.00       2         seconds() [5]
-----------------------------------------------
                0.01    0.00    2000/2000        Grid::setBCFunc(double (*)(double, double)) [7]
[6]      0.6    0.01    0.00    2000         BC(double, double) [6]
-----------------------------------------------
                0.00    0.01       1/1           main [1]
[7]      0.6    0.00    0.01       1         Grid::setBCFunc(double (*)(double, double)) [7]
                0.01    0.00    2000/2000        BC(double, double) [6]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [24]
[14]     0.0    0.00    0.00       1         _GLOBAL__sub_I__ZN4GridC2Eii [14]
                0.00    0.00       1/1           __static_initialization_and_destruction_0(int, int) [15]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I__ZN4GridC2Eii [14]
[15]     0.0    0.00    0.00       1         __static_initialization_and_destruction_0(int, int) [15]
-----------------------------------------------
                0.00    0.00       1/1           LaplaceSolver::LaplaceSolver(Grid*) [17]
[16]     0.0    0.00    0.00       1         LaplaceSolver::initialize() [16]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[17]     0.0    0.00    0.00       1         LaplaceSolver::LaplaceSolver(Grid*) [17]
                0.00    0.00       1/1           LaplaceSolver::initialize() [16]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[18]     0.0    0.00    0.00       1         LaplaceSolver::~LaplaceSolver() [18]
-----------------------------------------------
                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

Com o log gerado, observamos que o tempo total de execução foi de 0.90 s. Considerando isso, temos um grande hotspot na função ***LaplaceSolver::timeStep(double)*** e hotspots menores nas funções ***SQR(double const&)*** *- com um número muito alto de chamadas -* e ***seconds()****.* Observando o código, percebemos que este poderia sofrer algumas alterações seguindo as boas práticas de HPC. O código, no geral, segue estas diretrizes de forma parcial.

Alterações Realizadas

Visando a melhoria da performance do programa, foram feitas 3 alterações, cada uma delas atacando um dos hotspots mencionados anteriormente.

- ***LaplaceSolver::timeStep(double)****:*

Nesta função, percebi que havia uma parte da equação executada dentro do loop que não precisava ser computada tantas vezes, dado que seu valor não mudava a cada iteração. Assim, foi feita a seguinte alteração:

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real multiplier = 0.5/(dx2 + dy2);
    Real tmp;
    Real diff;
    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)*multiplier;
            diff = u[i][j] - tmp;
            err += diff*diff;
        }
    }
    return sqrt(err);
}
```

```
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);
}

Real LaplaceSolver :: solve(const int n_iter, const Real eps)
{
    Real err = timeStep();
```

O arquivo à esquerda é a versão modificada, enquanto o da direita é a versão original. Aqui, simplesmente isolamos o 0.5(dx2 + dy2) como uma variável *multiplier* antes do loop, de forma a reaproveitar esta computação, melhorando a performance.

- ***SQR(double const&)***

Esta função - como percebido no relatório do gprof - possui 24800400 chamadas, assim, a solução a ser utilizada foi retirar esta função do código e escrevê-la manualmente dentro do loop, único lugar onde ela é utilizada.

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real multiplier = 0.5/(dx2 + dy2);
    Real tmp;
    Real diff;
    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)*multiplier;
            diff = u[i][j] - tmp;
            err += diff*diff;
        }
    }
    return sqrt(err);
}
```

```
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);
}

Real LaplaceSolver :: solve(const int n_iter, const Real eps)
{
    Real err = timeStep();
```

```
inline Real SQR(const Real &x)
{
    return (x*x);
}
```

- ***seconds()***

Aqui, a alteração é análoga a feita com a função ***SQR***, porém, como esta era uma função bem menos impactante para a performance - talvez inclusive nem possa ser considerada um hotspot - sua alteração não traz benefícios tão grandes assim como as outras, porém estes ainda sim são existentes.

Observando a função:

```
inline double seconds(void)
{
    static const double secs_per_tick = 1.0 / CLOCKS_PER_SEC;
    return ( (double) clock() ) * secs_per_tick;
}
```

Percebemos que a cada chamada desta, teremos a criação da const *secs_per_tick*, o que parece ser desnecessário. Assim, vamos retirar o comportamento desta função e escrevê-la manualmente onde ela é chamada:

```
int main(int argc, char * argv[])
{
    int nx, n_iter;
    Real eps;
    Real t_start, t_end;
    std::cout << "Enter nx n_iter eps --> ";
    std::cin >> nx >> n_iter >> eps;

    Grid *g = new Grid(nx, nx);
    g->setBCFunc(BC);

    LaplaceSolver s = LaplaceSolver(g);

    std::cout <<"nx = " << g->nx << ", ny = " << g->ny
            << ", n_iter = " << n_iter << ", eps = "<<eps <<std::endl;

    static const double secs_per_tick = 1.0 / CLOCKS_PER_SEC;

    t_start = ( (double) clock() ) * secs_per_tick;
    std::cout << s.solve(n_iter, eps) << std::endl;
    t_end = ( (double) clock() ) * secs_per_tick;
    std::cout << "Iterations took " << t_end - t_start << " seconds.\n";

    return 0;
}
```

```
int main(int argc, char * argv[])
{
    int nx, n_iter;
    Real eps;
    Real t_start, t_end;
    std::cout << "Enter nx n_iter eps --> ";
    std::cin >> nx >> n_iter >> eps;

    Grid *g = new Grid(nx, nx);
    g->setBCFunc(BC);

    LaplaceSolver s = LaplaceSolver(g);

    std::cout <<"nx = " << g->nx << ", ny = " << g->ny
            << ", n_iter = " << n_iter << ", eps = "<<eps <<std::endl;

    t_start = seconds();
    std::cout << s.solve(n_iter, eps) << std::endl;
    t_end = seconds();
    std::cout << "Iterations took " << t_end - t_start << " seconds.\n";

    return 0;
}
```

Assim, passamos a criar a *secs_per_tick* apenas uma vez, otimizando ainda mais a performance do código.

Conclusão

Após realizar as alterações mencionadas anteriormente, rodamos o gprof novamente e obtivemos um ótimo resultado: o desempenho do programa melhorou consideravelmente.

A realização deste experimento deixou bem clara a importância de perfiladores - tais quais o gprof - como ferramenta de auxílio ao programador, principalmente aquele voltado à área de computação de alto desempenho. Este tipo de ferramenta se demonstrou muito útil, apontando quase que categoricamente pontos do código onde possivelmente caberia alguma melhoria

O novo tempo de execução foi de 52% do tempo de execução inicial, segue relatório abaixo.

Relatório:

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.38      0.47     0.47      100     4.72     4.72  LaplaceSolver::timeStep(double)
  0.00      0.47     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.47     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.47     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.47     0.00        1     0.00     0.00  LaplaceSolver::initialize()
  0.00      0.47     0.00        1     0.00   471.80  LaplaceSolver::solve(int, double)
  0.00      0.47     0.00        1     0.00     0.00  LaplaceSolver::LaplaceSolver(Grid*)
  0.00      0.47     0.00        1     0.00     0.00  LaplaceSolver::~LaplaceSolver()
  0.00      0.47     0.00        1     0.00     0.00  Grid::setBCFunc(double (*)(double, double))
  0.00      0.47     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 2.12% of 0.47 seconds

```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    0.47                 main [1]
                0.00    0.47       1/1           LaplaceSolver::solve(int, double) [3]
                0.00    0.00       1/1           Grid::Grid(int, int) [17]
                0.00    0.00       1/1           Grid::setBCFunc(double (*)(double, double)) [16]
                0.00    0.00       1/1           LaplaceSolver::LaplaceSolver(Grid*) [14]
                0.00    0.00       1/1           LaplaceSolver::~LaplaceSolver() [15]
-----------------------------------------------
                0.47    0.00     100/100         LaplaceSolver::solve(int, double) [3]
[2]    100.0    0.47    0.00     100         LaplaceSolver::timeStep(double) [2]
-----------------------------------------------
                0.00    0.47       1/1           main [1]
[3]    100.0    0.00    0.47       1         LaplaceSolver::solve(int, double) [3]
                0.47    0.00     100/100         LaplaceSolver::timeStep(double) [2]
-----------------------------------------------
                0.00    0.00    2000/2000        Grid::setBCFunc(double (*)(double, double)) [16]
[10]     0.0    0.00    0.00    2000         BC(double, double) [10]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [22]
[11]     0.0    0.00    0.00       1         _GLOBAL__sub_I__ZN4GridC2Eii [11]
                0.00    0.00       1/1           __static_initialization_and_destruction_0(int, int) [12]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I__ZN4GridC2Eii [11]
[12]     0.0    0.00    0.00       1         __static_initialization_and_destruction_0(int, int) [12]
-----------------------------------------------
                0.00    0.00       1/1           LaplaceSolver::LaplaceSolver(Grid*) [14]
[13]     0.0    0.00    0.00       1         LaplaceSolver::initialize() [13]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[14]     0.0    0.00    0.00       1         LaplaceSolver::LaplaceSolver(Grid*) [14]
                0.00    0.00       1/1           LaplaceSolver::initialize() [13]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[15]     0.0    0.00    0.00       1         LaplaceSolver::~LaplaceSolver() [15]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[16]     0.0    0.00    0.00       1         Grid::setBCFunc(double (*)(double, double)) [16]
                0.00    0.00    2000/2000        BC(double, double) [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[17]     0.0    0.00    0.00       1         Grid::Grid(int, int) [17]
-----------------------------------------------
```

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

[11] _GLOBAL__sub_I__ZN4GridC2Eii [3] LaplaceSolver::solve(int, double) [16] Grid::setBCFunc(double (*)(double, double))
[10] BC(double, double) [2] LaplaceSolver::timeStep(double) [17] Grid::Grid(int, int)
[12] __static_initialization_and_destruction_0(int, int) [14] LaplaceSolver::LaplaceSolver(Grid*)
[13] LaplaceSolver::initialize() [15] LaplaceSolver::~LaplaceSolver()

Todo o código utilizado pode ser acessado aqui: [Github]