Trabalho 2 - Computação de Alto Desempenho - 2021.1

Nome: Carlos Henrique Ferreira Brito Filho
DRE: 120081409

# 1 Introdução

O trabalho 2 da disciplina Computação de Alto Desempenho consiste na perfilagem de um código escrito em *C++*. Nesse código é implementado a resolução da Equação de Laplace utilizando o Método das Diferenças Finitas. Tal método discretiza EDPs substituindo as derivadas parciais com suas aproximações (também chamadas de diferenças finitas), ou seja, converte EDPs num conjunto de equações lineares e simultâneas. Equações simultâneas escritas em notação de matriz possuem a maioria de seus elementos zerados e, dependendo do problema, a quantidade de tais equações pode se tornar absurda, ultrapassando a casa dos milhares. Para evitar a alocação de matrizes tão grandes são utilizadas soluções iterativas, pois dessa forma se torna desnecessário armazenar tais matrizes. No código a ser perfilado é utilizado o método Gauss-Siedel, um método iterativo utilizado para resolver sistemas de equações lineares. As instruções desse trabalho especificam uma matriz $n \times n$ com $n = 500$, realizando 100 iterações e utilizando $10^{-16}$ de tolerância. O código será perfilado com essas especificações e sem *flags* de otimização, permitindo que seja determinado o desempenho do código ao analisar diversos fatores, como o tempo de execução, quantidade de chamadas e a porcentagem de tempo total para cada função do programa.

# 2 Utilizando o *gprof*

Para perfilar um código em *C++* utilizando o *GNU profiler*, também conhecido como *gprof*, é necessário:

1. Compilar o código com perfilagem ativada (*flag* -pg para códigos em C/C++ compilados com o gcc/g++):

```
g++ -pg -o laplace laplace.cxx
```

2. Executar o código (esse passo irá gerar o arquivo *gmon.out* com os dados do perfil):

```
./laplace
```

3. Executar a ferramenta *gprof* com o executável compilado e o arquivo *gmon.out* como parâmetros:

```
gprof ./laplace gmon.out > laplace.txt
```

# 3 Relatórios gerados pelo *gprof*

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 86.04      0.60     0.60      100     6.02     7.03  LaplaceSolver::
   timeStep(double)
 14.34      0.70     0.10 24800400     0.00     0.00  SQR(double const&)
  0.00      0.70     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.70     0.00        2     0.00     0.00  seconds()
  0.00      0.70     0.00        1     0.00     0.00
   _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.70     0.00        1     0.00     0.00
   __static_initialization_and_destruction_0(int, int)
  0.00      0.70     0.00        1     0.00     0.00  LaplaceSolver::
   initialize()
  0.00      0.70     0.00        1     0.00   702.62  LaplaceSolver::solve(
   int, double)
  0.00      0.70     0.00        1     0.00     0.00  LaplaceSolver::
   LaplaceSolver(Grid*)
  0.00      0.70     0.00        1     0.00     0.00  LaplaceSolver::~
   LaplaceSolver()
  0.00      0.70     0.00        1     0.00     0.00  Grid::setBCFunc(double
    (*)(double, double))
  0.00      0.70     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 1.42% of 0.70 seconds

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.70                 main [1]
                0.00    0.70       1/1           LaplaceSolver::solve(int,
   double) [3]
                0.00    0.00       2/2           seconds() [12]
                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]
-----------------------------------------------
                0.60    0.10     100/100         LaplaceSolver::solve(int,
   double) [3]
[2]    100.0    0.60    0.10     100         LaplaceSolver::timeStep(double)
    [2]
                0.10    0.00 24800400/24800400     SQR(double const&) [4]
-----------------------------------------------
                0.00    0.70       1/1           main [1]
[3]    100.0    0.00    0.70       1         LaplaceSolver::solve(int,
   double) [3]
                0.60    0.10     100/100         LaplaceSolver::timeStep(
   double) [2]
-----------------------------------------------
                0.10    0.00 24800400/24800400     LaplaceSolver::timeStep(
   double) [2]
[4]     14.3    0.10    0.00 24800400         SQR(double const&) [4]
-----------------------------------------------
                0.00    0.00    2000/2000        Grid::setBCFunc(double (*)(
   double, double)) [18]
[11]     0.0    0.00    0.00    2000         BC(double, double) [11]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[12]     0.0    0.00    0.00       2         seconds() [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) [11]
-----------------------------------------------
                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.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


Index by function name

  [13] _GLOBAL__sub_I__ZN4GridC2Eii [12] seconds()        [16] LaplaceSolver
   ::LaplaceSolver(Grid*)
  [11] BC(double, double)     [15] LaplaceSolver::initialize() [17]
   LaplaceSolver::~LaplaceSolver()
   [4] SQR(double const&)      [3] LaplaceSolver::solve(int, double) [18]
   Grid::setBCFunc(double (*)(double, double))
  [14] __static_initialization_and_destruction_0(int, int) [2] LaplaceSolver
   ::timeStep(double) [19] Grid::Grid(int, int)
```

Ao analisar os relatórios gerados pelo *gprof* é possível perceber que certas funções realizam muitas chamadas (como a função *SQR*, por exemplo), enquanto outras acumulam mais de 2/3 do tempo total de execução do programa (como a função *timeStep*, por exemplo). Essas funções podem ser consideradas *hotspots* e, ao serem investigadas, podem ser encontradas más práticas de HPC, como, por exemplo, funções sendo chamadas dentro de um laço e expressões constantes presentes em laços.

# 4 Alterações realizadas

Analisando a função *SQR*:

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

Que é chamada na função *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);
}
```

É possível observar uma função sendo chamada dentro de um laço. É considerado boa prática possuirmos chamadas de rotinas explicitamente onde elas são chamadas (técnica de *in-lining*), logo a seguinte alteração foi feita na função *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;
    Real newSQR;
    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);
            newSQR = (u[i][j] - tmp);
            err += newSQR * newSQR;
```

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

Comparando o tempo de iteração da versão original (1.19941 segundos) e da versão alterada do código (0.63964 segundos) é possível reparar a diferença causada pela técnica de *in-lining*. Sem a utilização da função *SQR* não temos mais milhões de chamadas tomando cerca de 14% do tempo do programa. A função *timeStep* levava, em média, 7.03 milissegundos por chamada. Após a técnica ser aplicada o valor é de 7.03 milissegundos.

Ainda na função *timeStep* há uma constante ($dx2 + dy2$) sendo chamada dentro de um laço. Ao declarar a constante fora do laço ($dx2PlusDy2$), é possível reparar uma melhora de desempenho no tempo de iteração (0.544623 segundos) e no tempo médio por chamada da função, que agora é de 5.32 milissegundos.

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;
    Real newSQR;
    Real dx2PlusDy2 = dx2 + dy2;
    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/dx2PlusDy2);
            newSQR = (u[i][j] - tmp);
            err += newSQR * newSQR;
        }
    }
    return sqrt(err);
}
```

Porém, é possível perceber que existe uma sub-expressão ($0.5/dx2PlusDy2$) dentro do laço na função. A eliminação dessa sub-expressão criando uma variável fora do laço nos traz um tempo de iteração de 0.519896 segundos e um tempo médio por chamada da função de 5.22 milissegundos.

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;
    Real newSQR;
    Real dx2PlusDy2 = dx2 + dy2;
    Real constantValue = (0.5/dx2PlusDy2);
    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)*constantValue;
            newSQR = (u[i][j] - tmp);
            err += newSQR * newSQR;
        }
    }
    return sqrt(err);
}
```

Finalmente, ao reduzir a sub-expressão presente na variável $constantValue$, temos um tempo de iteração de 0.507527 segundos e um tempo médio por chamada de 5.02 milissegundos.

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;
    Real newSQR;
    Real dx2PlusDy2 = dx2 + dy2;
    Real constantValue = 1/(2*(dx2PlusDy2));
    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)*constantValue;
            newSQR = (u[i][j] - tmp);
            err += newSQR * newSQR;
        }
    }
    return sqrt(err);
}
```

A seguir se encontram a versão final do código onde todas as mudanças acima foram aplicadas e os relatórios gerados pelo *gprof* para esta versão:

## 4.1 Código

```
/* A pure C/C++ version of a Gauss-Siedel Laplacian solver to test the
speed of a C program versus that of doing it with
Python/Numeric/Weave. */
#include <iostream>
#include <cmath>
#include <time.h>

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

inline Real BC(Real x, Real y)
{
    return (x*x - y*y);
}

struct Grid {
    Real dx, dy;
    int nx, ny;
    Real **u;
    Grid(const int n_x=10, const int n_y=10);
    ~Grid();
    void setBCFunc(Real (*f)(const Real, const Real));
```

```
    /*Real computeError();*/
};
Grid :: Grid(const int n_x, const int n_y) : nx(n_x), ny(n_y)
{
    dx = 1.0/Real(nx - 1);
    dy = 1.0/Real(ny - 1);

    u = new Real* [nx];
    for (int i=0; i<nx; ++i) {
        u[i] = new double [ny];
    }

    for (int i=0; i<nx; ++i) {
        for (int j=0; j<ny; ++j) {
            u[i][j] = 0.0;
        }
    }
}

Grid :: ~Grid()
{
    for (int i=0; i<nx; ++i) {
        delete [] u[i];
    }
    delete [] u;
}

void Grid :: setBCFunc(Real (*f)(const Real, const Real))
{
    Real xmin, ymin, xmax, ymax, x, y;
    xmin = 0.0;
    ymin = 0.0;
    xmax = 1.0;
    ymax = 1.0;
    /* Left and right sides. */
    for (int j=0; j<ny; ++j) {
        y = j*dy;
        u[0][j] = f(xmin, y);
        u[nx-1][j] = f(xmax, y);
    }
    /* Top and bottom sides. */
    for (int i=0; i<nx; ++i) {
        x = i*dx;
        u[i][0] = f(x, ymin);
        u[i][ny-1] = f(x, ymax);
```

```
    }
}

struct LaplaceSolver{
    Grid *g;

    LaplaceSolver(Grid *g);
    ~LaplaceSolver();
    void initialize();
    Real timeStep(const Real dt=0.0);
    Real solve(const int n_iter=0, const Real eps=1e-16);
};

LaplaceSolver :: LaplaceSolver(Grid *grid)
{
    g = grid;
    initialize();
}

LaplaceSolver:: ~LaplaceSolver()
{
}

void LaplaceSolver :: initialize()
{
}

Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;
    Real newSQR;
    Real dx2PlusDy2 = dx2 + dy2;
    Real constantValue = 1/(2*(dx2PlusDy2));
    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)*constantValue;
            newSQR = (u[i][j] - tmp);
```

```
            err += newSQR * newSQR;
        }
    }
    return sqrt(err);
}

Real LaplaceSolver :: solve(const int n_iter, const Real eps)
{
    Real err = timeStep();
    int count = 1;
    while (err > eps) {
        if (n_iter && (count >= n_iter)) {
            return err;
        }
        err = timeStep();
        ++count;
    }
    return Real(count);
}

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;
}
```

## 4.2 Relatórios

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.38      0.50     0.50      100     5.02     5.02  LaplaceSolver::
   timeStep(double)
  0.00      0.50     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.50     0.00        2     0.00     0.00  seconds()
  0.00      0.50     0.00        1     0.00     0.00
   _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.50     0.00        1     0.00     0.00
   __static_initialization_and_destruction_0(int, int)
  0.00      0.50     0.00        1     0.00     0.00  LaplaceSolver::
   initialize()
  0.00      0.50     0.00        1     0.00   501.91  LaplaceSolver::solve(
   int, double)
  0.00      0.50     0.00        1     0.00     0.00  LaplaceSolver::
   LaplaceSolver(Grid*)
  0.00      0.50     0.00        1     0.00     0.00  LaplaceSolver::~
   LaplaceSolver()
  0.00      0.50     0.00        1     0.00     0.00  Grid::setBCFunc(double
    (*)(double, double))
  0.00      0.50     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.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.99% of 0.50 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.50                 main [1]
                0.00    0.50       1/1           LaplaceSolver::solve(int,
   double) [3]
                0.00    0.00       2/2           seconds() [11]
                0.00    0.00       1/1           Grid::Grid(int, int) [18]
                0.00    0.00       1/1           Grid::setBCFunc(double (*)(
   double, double)) [17]
                0.00    0.00       1/1           LaplaceSolver::
   LaplaceSolver(Grid*) [15]
                0.00    0.00       1/1           LaplaceSolver::~
   LaplaceSolver() [16]
-----------------------------------------------
                0.50    0.00     100/100         LaplaceSolver::solve(int,
   double) [3]
[2]    100.0    0.50    0.00     100         LaplaceSolver::timeStep(double)
    [2]
-----------------------------------------------
```

```
                0.00    0.50       1/1           main [1]
[3]    100.0    0.00    0.50       1         LaplaceSolver::solve(int,
   double) [3]
                0.50    0.00     100/100         LaplaceSolver::timeStep(
   double) [2]
-----------------------------------------------
                0.00    0.00    2000/2000        Grid::setBCFunc(double (*)(
   double, double)) [17]
[10]     0.0    0.00    0.00    2000         BC(double, double) [10]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[11]     0.0    0.00    0.00       2         seconds() [11]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [23]
[12]     0.0    0.00    0.00       1         _GLOBAL__sub_I__ZN4GridC2Eii
   [12]
                0.00    0.00       1/1
   __static_initialization_and_destruction_0(int, int) [13]
-----------------------------------------------
                0.00    0.00       1/1
   _GLOBAL__sub_I__ZN4GridC2Eii [12]
[13]     0.0    0.00    0.00       1
   __static_initialization_and_destruction_0(int, int) [13]
-----------------------------------------------
                0.00    0.00       1/1           LaplaceSolver::
   LaplaceSolver(Grid*) [15]
[14]     0.0    0.00    0.00       1         LaplaceSolver::initialize()
   [14]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[15]     0.0    0.00    0.00       1         LaplaceSolver::LaplaceSolver(
   Grid*) [15]
                0.00    0.00       1/1           LaplaceSolver::initialize()
    [14]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[16]     0.0    0.00    0.00       1         LaplaceSolver::~LaplaceSolver()
    [16]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[17]     0.0    0.00    0.00       1         Grid::setBCFunc(double (*)(
   double, double)) [17]
                0.00    0.00    2000/2000        BC(double, double) [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
```

```
[18]     0.0    0.00    0.00       1         Grid::Grid(int, int) [18]
-----------------------------------------------
```

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.

```
notice and this notice are preserved.


Index by function name

  [12] _GLOBAL__sub_I__ZN4GridC2Eii [14] LaplaceSolver::initialize() [16]
   LaplaceSolver::~LaplaceSolver()
  [10] BC(double, double)      [3] LaplaceSolver::solve(int, double) [17]
   Grid::setBCFunc(double (*)(double, double))
  [13] __static_initialization_and_destruction_0(int, int) [2] LaplaceSolver
   ::timeStep(double) [18] Grid::Grid(int, int)
  [11] seconds()              [15] LaplaceSolver::LaplaceSolver(Grid*)
```

# 5 Conclusão

É possível observar que houve uma melhora de cerca de 58% no tempo de iteração e de cerca de 28% no tempo médio por chamada da função *timeStep* após serem aplicadas as técnicas de otimização manual no código. Foi possível aplicar tais técnicas graças ao uso do *gprof*, responsável por nos mostrar os *hotspots* do programa ao ser executado após cada mudança feita no código, até que o mesmo fosse considerado aceitável de acordo com as boas práticas de programação.