COC472 - Computação de Alto Desempenho

Segundo Trabalho

UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

- Pedro Tubenchlak Boechat - DRE 119065050
  - pedroboechat@poli.ufrj.br

O código-fonte desse trabalho está disponível no meu GitHub (https://github.com/pedroboechat/UFRJ/tree/main/COC472/Trabalho%202).

Rio de Janeiro - RJ
3 de Setembro de 2021 (2021.1)

Introdução

O profilador é uma ferramenta muito importante na computação. Profiladores ajudam a analisar, por exemplo, a complexidade de espaço em memória ou tempo, uso de certas instruções e até a frequência e duração da chamada de funções em um programa, métricas importantes para entender os *hotspots* desse e otimizá-lo. Para este trabalho utilizaremos o profilador *Gproof* para analisar o código *laplace.cxx* dado. Este código consiste em um programa, implementado com o método iterativo de Gauss-Seidel e na linguagem C/C++, que resolve a Equação de Laplace utilizando o Método das Diferenças Finitas.

Uso do Gproof

Para utilizar o *Gproof*:

1. Compile o código utilizando as flags *-pg*;
2. Execute o programa compilado uma vez;
3. Execute o *Gproof* passando como argumentos o programa compilado e o arquivo *.out* gerado na compilação.

Relatório do profilador

O arquivo contendo o relatório completo do profilador pode ser acessado pelo repositório do GitHub, cujo link está presente na capa, ou por este link. A execução durou 0.645611 segundos. Analisando o flat profile gerado pelo *Gprof* (abaixo), identificamos *hotspots* nas funções *LaplaceSolver::timeStep* e *SQR*. O código poderia, visando seguir as boas práticas de HPC, remover algumas funções e editar a implementação de outras para que execute mais rápido.

```
Enter nx n_iter eps --> 500 100 1e-16
nx = 500, ny = 500, n_iter = 100, eps = 1e-16
0.230163
Iterations took 0.645611 seconds.
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 87.83      0.28     0.28      100     2.81     3.21  LaplaceSolver::timeStep(double)
 12.55      0.32     0.04 24800400     0.00     0.00  SQR(double const&)
  0.00      0.32     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.32     0.00        2     0.00     0.00  seconds()
  0.00      0.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.32     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.32     0.00        1     0.00     0.00  LaplaceSolver::initialize()
  0.00      0.32     0.00        1     0.00   321.20  LaplaceSolver::solve(int, double)
  0.00      0.32     0.00        1     0.00     0.00  LaplaceSolver::LaplaceSolver(Grid*)
  0.00      0.32     0.00        1     0.00     0.00  LaplaceSolver::~LaplaceSolver()
  0.00      0.32     0.00        1     0.00     0.00  Grid::setBCFunc(double (*)(double, double))
  0.00      0.32     0.00        1     0.00     0.00  Grid::Grid(int, int)
```

Otimização dos *hotspots*

1. Função *LaplaceSolver::timeStep*
   Essa função é chamada poucas vezes, mas representa 86.04% do tempo de execução do programa. Ao analisá-la, encontramos que há uma multiplicação por constante na linha 89:

```
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);
```

Para otimizar isso, podemos armazenar essa constante em uma variável, da seguinte forma:

```
Real factor = dx2*0.5/(dx2 + dy2);
u[i][j] = ((u[i-1][j] + u[i+1][j])*dy2 + (u[i][j-1] + u[i][j+1]))*factor;
```

2. Função *SQR*
   Esta função é chamada 24.8 milhões de vezes e representa 12.55% do tempo de execução do programa. Ao analisá-la, percebemos que ela compreende apenas uma operação de multiplicação e só é chamada em um lugar no programa, na linha 90:

```
err += SQR(u[i][j] - tmp);
```

Assim, é possível removê-la e reescrever essa multiplicação de forma otimizada, da seguinte forma:

```
Real tmpErr; // declarado fora do loop
for (...
— for (...
—— tmpErr = u[i][j] - tmp;
—— err += tmpErr*tmpErr;
```

Conclusão

Após serem feitas as otimizações, o programa foi executado novamente. Dessa vez, sua execução foi concluida em 0.182107 segundos, representando uma redução de 71.8% em relação ao código original e confirmando a importância da profilagem em HPC. Ainda, ao analisarmos o novo flat profile (abaixo), confirmamos que a alteração feita na função *LaplaceSolver::timeStep* refletiu em uma redução de 35.7% no tempo gasto na mesma.

```
Enter nx n_iter eps --> 500 100 1e-16
nx = 500, ny = 500, n_iter = 100, eps = 1e-16
0.230163
Iterations took 0.182107 seconds.
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.38      0.18     0.18      100     1.81     1.81  LaplaceSolver::timeStep(double)
  0.00      0.18     0.00     2000     0.00     0.00  BC(double, double)
  0.00      0.18     0.00        2     0.00     0.00  seconds()
  0.00      0.18     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN4GridC2Eii
  0.00      0.18     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.18     0.00        1     0.00     0.00  LaplaceSolver::initialize()
  0.00      0.18     0.00        1     0.00   180.69  LaplaceSolver::solve(int, double)
  0.00      0.18     0.00        1     0.00     0.00  LaplaceSolver::LaplaceSolver(Grid*)
  0.00      0.18     0.00        1     0.00     0.00  LaplaceSolver::~LaplaceSolver()
  0.00      0.18     0.00        1     0.00     0.00  Grid::setBCFunc(double (*)(double, double))
  0.00      0.18     0.00        1     0.00     0.00  Grid::Grid(int, int)
```