Trabalho 3 - Computação de Alto Desempenho
Semestre 2020/2
Thiago Guimarães - DRE: 118053123

Introdução

O objeto deste trabalho é utilizar o OpenMP e seu paralelismo para melhora do desempenho do programa fornecido no segundo trabalho da disciplina - laplace.cxx

Identificando loops mais intensivos

Observando a perfilagem realizada no trabalho 2 da disciplina, ficam claros os loops mais intensivos onde serão aplicadas as alterações, sendo estes os 2 loops presentes na função ***LaplaceSolver::timeStep***

```
122     for (int i=1; i<nx-1; ++i) {
123         for (int j=1; j<ny-1; ++j) {
124             tmp = u[i][j];
125             u[i][j] = ((u[i-1][j] + u[i+1][j])*dy2 +
126                        (u[i][j-1] + u[i][j+1])*dx2)*0.5/(dx2 + dy2);
127             err += SQR(u[i][j] - tmp);
128         }
129     }
```

Justificativa das alterações realizadas

As alterações realizadas foram as seguintes:

- Foi adicionada a biblioteca openmp

```
1   /* A pure C/C++ version of
2      speed of a C program ver
3      Python/Numeric/Weave. */
4   #include <omp.h>
```

- Foi adicionado o paralelismo nos loops principais identificados

```
Real LaplaceSolver :: timeStep(const Real dt)
{
    Real dx2 = g->dx*g->dx;
    Real dy2 = g->dy*g->dy;
    Real tmp;
    Real err = 0.0;      You, 3 days ago • adding exercise 3 start
    int nx = g->nx;
    int ny = g->ny;
    Real **u = g->u;
    int i;
    int j;

#pragma omp parallel for private(tmp, i, j) reduction(+ : err)
    for (i=1; i<nx-1; ++i) {
        for (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);
}
```

Aqui, foram colocadas como variáveis privadas de cada thread as variáveis *tmp, i* e *j* e foi aplicado *reduction* para somar os resultados da variável erro nela mesma. As variáveis *tmp, i* e *j* foram colocadas como privadas pois elas poderiam ter seus valores alterados em threads diferentes, comprometendo o resultado final.

- Foi adicionado a chamada *omp_get_wtime()* para computar o tempo de execução no código paralelizado.

```
int main(int argc, char * argv[])
{
    int nx, n_iter;
    const char* number_of_threads = getenv("OMP_NUM_THREADS");
    Real result;
    Real eps;
    double t_start, t_end;
    cin >> nx >> n_iter >> eps;

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

    LaplaceSolver s = LaplaceSolver(g);

#pragma omp parallel
    t_start = omp_get_wtime();
    result = s.solve(n_iter, eps);
    t_end = omp_get_wtime();
    cout << g->nx << "," << t_end - t_start << "," << number_of_threads << "," << result << endl;

    return 0;
}
```

Comparação de desempenho

Rodando o código, foram obtidos os seguintes resultados:

OBS: A coluna run_type indica o tipo de código que foi rodado

vectorized = apenas flag -O3
vectorized_parallelism = flags -O3 e -fopenmp
NoFlags = código cru, sem flags
parallelism = apenas flag -fopenmp

nx	time(s)	n_threads	result	run_type
512	2.19483	1	0.0402737	vectorized
1024	9.24227	1	0.0578281	vectorized
2048	42.4996	1	0.0820855	vectorized
512	2.22711	1	0.0402737	vectorized_parallelism
512	1.13851	2	0.0402737	vectorized_parallelism
512	0.777652	3	0.0402737	vectorized_parallelism
512	0.584296	4	0.0402737	vectorized_parallelism
512	0.526073	5	0.0402737	vectorized_parallelism
512	0.831037	6	0.0402738	vectorized_parallelism
512	1.30276	7	0.0402742	vectorized_parallelism
512	5.59083	8	0.0402764	vectorized_parallelism
1024	9.4939	1	0.0578281	vectorized_parallelism
1024	4.88966	2	0.0578281	vectorized_parallelism
1024	3.34172	3	0.0578281	vectorized_parallelism
1024	2.61676	4	0.0578281	vectorized_parallelism
1024	2.18361	5	0.0578281	vectorized_parallelism
1024	2.57049	6	0.0578282	vectorized_parallelism
1024	3.98154	7	0.0578284	vectorized_parallelism
1024	10.8481	8	0.0578266	vectorized_parallelism
2048	42.9693	1	0.0820855	vectorized_parallelism
2048	23.0002	2	0.0820855	vectorized_parallelism

2048	16.8643	3	0.0820855	vectorized_parallelism
2048	13.5646	4	0.0820855	vectorized_parallelism
2048	11.6042	5	0.0820855	vectorized_parallelism
2048	10.9529	6	0.0820855	vectorized_parallelism
2048	15.5535	7	0.0820858	vectorized_parallelism
2048	22.4566	8	0.0820861	vectorized_parallelism
512	3.50258	1	0.0402737	parallelism
512	1.98624	2	0.0402737	parallelism
512	1.71649	3	0.0402737	parallelism
512	1.37711	4	0.0402737	parallelism
512	1.09112	5	0.0402737	parallelism
512	0.984572	6	0.0402737	parallelism
512	1.01823	7	0.040274	parallelism
512	2.96999	8	0.0402738	parallelism
1024	14.6699	1	0.0578281	parallelism
1024	7.93187	2	0.0578281	parallelism
1024	5.76047	3	0.0578281	parallelism
1024	5.15026	4	0.0578281	parallelism
1024	4.80428	5	0.0578281	parallelism
1024	4.03812	6	0.0578281	parallelism
1024	5.57751	7	0.0578282	parallelism
1024	9.12892	8	0.0578291	parallelism
2048	65.2737	1	0.0820855	parallelism
2048	37.0863	2	0.0820855	parallelism
2048	29.7721	3	0.0820855	parallelism
2048	25662	4	0.0820855	parallelism
2048	23.2789	5	0.0820855	parallelism
2048	20.1422	6	0.0820855	parallelism
2048	26.0097	7	0.0820857	parallelism

2048	30.7216	8	0.082086	parallelism
512	3.62287	1	0.0402737	noFlags
1024	14597	1	0.0578281	noFlags
2048	63.3798	1	0.0820855	noFlags

Observando os resultados obtidos, podemos observar algumas coisas: O ganho de performance é proporcional ao uso das flags adicionais, possuindo inclusive valores diferentes de performance, no caso do uso da flag *fopenmp*, com diferentes número de threads. A melhor performance é observada com 5 ou 6 threads, a depender do caso, e utilizando a flag de vetorização. A vetorização em si também melhora a performance sozinha e o mesmo vale para o uso de threads por si só. Com mais do que 6 threads, a performance piora consideravelmente.

Todo o código auxiliar utilizado pode ser encontrado aqui: Github
O código principal (laplace.cxx) também pode ser encontrado logo abaixo:

```
/* 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 <omp.h>
#include <iostream>
#include <cmath>
#include <time.h>
using namespace std;

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

```
}

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;
    int nx = g->nx;
    int ny = g->ny;
    Real **u = g->u;
    int i;
    int j;

#pragma omp parallel for private(tmp, i, j) reduction(+ : err)
    for (i=1; i<nx-1; ++i) {
        for (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();
    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;
    const char* number_of_threads = getenv("OMP_NUM_THREADS");
    Real result;
    Real eps;
    double t_start, t_end;
    cin >> nx >> n_iter >> eps;

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

    LaplaceSolver s = LaplaceSolver(g);

#pragma omp parallel
    t_start = omp_get_wtime();
    result = s.solve(n_iter, eps);
    t_end = omp_get_wtime();
    cout << g->nx << "," << t_end - t_start << "," <<
number_of_threads << "," << result << endl;

    return 0;
}
```