Computação de Alto Desempenho COC472 - Trabalho 3

Bruno Dantas de Paiva
DRE: 118048097

May 23, 2021

1 Questão 1

Para o código utilizado neste problema, é possível observar que os loops mais intensivos são os dois loops contidos na função timeStep onde estes foram identificados no trabalho anterior como um dos hotspots do código.

2 Questão 2

Para as modificações realizadas, foi necessário utilizar algumas variáveis privadas, compartilhadas e também o operando reduction.

A escolha de variáveis privadas foi feita com base na necessidade de cada thread ter seu valor único. Ou seja, para o loop mais intensivo, serão as variáveis: i, j e tmp.

A escolha de variáveis compartilhadas foi feita com base na necessidade das threads compartilharem os valores obtidos, onde, para o loop mais intensivo, serão as variáveis: u, dx2 e dxy.

Finalmente, porém não menos importante, a escolha do reduction foi necessária para que as threads façam a redução da variável em questão para obter o resultado final, onde, neste caso, será a variável err.

Além disso, outras modificações foram realizadas no código para que fosse possível melhorar a visualização dos dados como será possível observar na questão 3.

3 Questão 3

Para esta questão, foi desenvolvido um arquivo bash para que fosse possível comparar os tempos obtidos pelo arquivo modificado e arquivo não modificado com todas as flags requisitadas. Tal arquivo é possível observar abaixo:

Resultado	Valor de Nx	Tempo(s)	Flags	Thread(s)
0.0402737	512	2.5493	None	1
0.0578281	1024	10.3824	None	1
0.0820855	2048	44.7824	None	1
0.0402737	512	1.65883	O3	1
0.0578281	1024	6.77318	O3	1
0.0820855	2048	28.9183	O3	1
0.0402737	512	2.53473	fopenmp	1
0.0578281	1024	10.4357	fopenmp	1
0.0820855	2048	45.0093	fopenmp	1
0.0402737	512	1.29655	fopenmp	2
0.0578281	1024	5.34382	fopenmp	2
0.0820855	2048	23.2102	fopenmp	2
0.0402737	512	0.875816	fopenmp	3
0.0578281	1024	3.57281	fopenmp	3
0.0820855	2048	16.5417	fopenmp	3
0.0402737	512	0.670192	fopenmp	4
0.0578281	1024	2.7771	fopenmp	4
0.0820855	2048	12.9988	fopenmp	4
0.0402737	512	0.561462	fopenmp	5
0.0578281	1024	2.26664	fopenmp	5
0.0820855	2048	10.5943	fopenmp	5
0.0402737	512	0.484002	fopenmp	6
0.0578281	1024	2.00666	fopenmp	6
0.0820855	2048	9.46175	fopenmp	6
0.0402737	512	1.65881	O3,fopenmp	1
0.0578281	1024	6.75583	O3,fopenmp	1
0.0820855	2048	28.9666	O3,fopenmp	1
0.0402737	512	0.844062	O3,fopenmp	2
0.0578281	1024	3.47093	O3,fopenmp	2
0.0820855	2048	14.8859	O3,fopenmp	2
0.0402737	512	0.575362	O3,fopenmp	3
0.0578281	1024	2.31364	O3,fopenmp	3
0.0820855	2048	10.5487	O3,fopenmp	3
0.0402737	512	0.438704	O3,fopenmp	4
0.0578281	1024	1.77192	O3,fopenmp	4
0.0820855	2048	8.15395	O3,fopenmp	4
0.0402737	512	0.3921	O3,fopenmp	5
0.0578281	1024	1.435	O3,fopenmp	5
0.0820855	2048	6.63067	O3,fopenmp	5
0.0402737	512	0.327476	O3,fopenmp	6
0.0578282	1024	1.33472	O3,fopenmp	6
0.0820855	2048	6.12336	O3,fopenmp	6

Table 1: Tabela com os resultados de tempo e número de threads

Como é possível observar com a tabela, o código inicialmente sem as modificações possui um tempo de execução muito maior que o código otimizado com o maior número de threads da máquina utilizada.

Além disso, com relação à escalabilidade, é possível ver que com o aumento do número de threads o código passa a ser paralelizado no número de threads do computador dividindo o tempo conforme este número aumenta. Também é possível observar que o tempo após 5 threads não diminui tanto quanto a diferença de tempo entre um número menor de threads.

4 Códigos

4.1 Bash

```
INPUT_DIR=input_file
INPUT_FILE=input.txt

OUTPUT_DIR=output_file
OUTPUT_FILE=output.csv

CPP_DIR=cpp_files
BASE_PROGRAM=laplace.cxx
OPTIMIZED_PROGRAM=laplace_openmp.cxx

EXECUTOR=runner

BASE_PROGRAM_PATH=$CPP_DIR/$BASE_PROGRAM
OPTIMIZED_PROGRAM_PATH=$CPP_DIR/$OPTIMIZED_PROGRAM
INPUT_PATH=$INPUT_DIR/$INPUT_FILE
OUTPUT_PATH=$OUTPUT_DIR/$OUTPUT_FILE

mkdir -p $INPUT_DIR
mkdir -p $OUTPUT_DIR

MAX_THREADS=$(python -c "import psutil; print(psutil.cpu_count(logical=True))")

echo 'Result;Nx;Time(s);Flags;Thread(s)' > $OUTPUT_PATH

echo "Starting the base program without flags"
g++ $BASE_PROGRAM_PATH -o $EXECUTOR
for nx in 512 1024 2048;
do
    echo $nx 1000 0.000000000000001 "None" > $INPUT_PATH
    echo $(./$EXECUTOR < $INPUT_PATH >> $OUTPUT_PATH) "Nx used was $nx"
done

echo "Starting the base program with O3 flag"
g++ $BASE_PROGRAM_PATH -O3 -o $EXECUTOR
for nx in 512 1024 2048;
do
    echo $nx 1000 0.000000000000001 "O3" > $INPUT_PATH
    echo $(./$EXECUTOR < $INPUT_PATH >> $OUTPUT_PATH) "Nx used was $nx"
done
```

```
echo "Starting the openmp program without flags"
g++ $OPTIMIZED_PROGRAM_PATH -fopenmp -o $EXECUTOR
for counter in $(seq 1 $MAX_THREADS);
do
    echo "Actual number of used threads is $counter"
    export OMP_NUM_THREADS=$counter
    for nx in 512 1024 2048;
    do
        echo $nx 1000 0.000000000000001 "fopenmp" > $INPUT_PATH
        echo $(./$EXECUTOR < $INPUT_PATH >> $OUTPUT_PATH) "Nx used was $nx"
    done
done

echo "Starting the openmp program with O3 flag"
g++ $OPTIMIZED_PROGRAM_PATH -O3 -fopenmp -o $EXECUTOR
for counter in $(seq 1 $MAX_THREADS);
do
    echo "Actual number of used threads is $counter"
    export OMP_NUM_THREADS=$counter
    for nx in 512 1024 2048;
    do
        echo $nx 1000 0.000000000000001 "O3,fopenmp" > $INPUT_PATH
        echo $(./$EXECUTOR < $INPUT_PATH >> $OUTPUT_PATH) "Nx used was $nx"
    done
done

rm $EXECUTOR
rm -rf $INPUT_DIR
```

4.2 C++

```
#include <omp.h>


Grid::Grid(const int n_x, const int n_y) : nx(n_x), ny(n_y)
{
    int i, j;

    dx = 1.0 / Real(nx - 1);
    dy = 1.0 / Real(ny - 1);

    u = new Real *[nx];

#pragma omp parallel for
    for (i = 0; i < nx; ++i)
    {
        u[i] = new double[ny];
    }

#pragma omp parallel for
    for (i = 0; i < nx; ++i)
    {
        for (j = 0; j < ny; ++j)
        {
            u[i][j] = 0.0;
        }
    }
}

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

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

#pragma omp parallel for private(tmp, i, j) shared(u, dx2, dy2) 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);
}

int main(int argc, char *argv[])
{
    int nx, n_iter, th_id, nthreads;
    double t_start, t_end;
    std::string method;
    Real eps;
    Real result;
    std::cin >> nx >> n_iter >> eps >> method;

#pragma omp parallel
    nthreads = omp_get_num_threads();

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

    LaplaceSolver s = LaplaceSolver(g);

    t_start = omp_get_wtime();
    result = s.solve(n_iter, eps);
    t_end = omp_get_wtime();
    std::cout << result << ";" << g->nx << ";" << t_end - t_start << ";'" << method << "

    return 0;
}
```

4.3 Observação

No código em C++, foi colocada somente as modificações realizadas no código. Também, é possível verificar que algumas modificações não contribuem necessariamente para a otimização do código, como as mudanças no construtor do grid, estas foram feitas para fins de estudos e melhor entendimento da biblioteca Openmp.

4.4 Execução

Para executar o programa, conforme comentado anteriormente, foi criado um arquivo em bash, deste modo seria possível automatizar a execução dos códigos e alternar entre o número de threads existente no computador.

Para verificar o número de threads automaticamente, foi criada uma rotina em python para retornar esse valor. Deste modo, é possível que qualquer pessoa execute o código com base no número de threads no seu computador sem qualquer modificação.

Todo o trabalho desenvolvido com os códigos também se encontra neste repositório: Third_Exercise.