## OpenMP编程

**代码**

#define MATRIXLENGTH 1000

#define TIME 100

double matrix[MATRIXLENGTH][MATRIXLENGTH] = { 0.0 };

double newMatrix[MATRIXLENGTH][MATRIXLENGTH] = { 0.0 };

void calculate(double matrix[MATRIXLENGTH][MATRIXLENGTH], double newMatrix[MATRIXLENGTH][MATRIXLENGTH], int height, int length, int ompNum) {

// 被除数，如果在四个角上，只能计算2邻域，则被除数应该为2，在边上，只能计算3个邻域，被除数应该是3

int num = 0;

omp\_set\_num\_threads(ompNum);

#pragma omp parallel

{

#pragma omp for private(num)

for (int i = 0; i < height; i++) {

for (int j = 0; j < length; j++) {

if (i > 0) {newMatrix[i][j] += matrix[i - 1][j]; num += 1;}

if (j > 0) {newMatrix[i][j] += matrix[i][j - 1]; num += 1;}

if (i < height - 1) {newMatrix[i][j] += matrix[i + 1][j]; num += 1;}

if (j < length - 1) {newMatrix[i][j] += matrix[i][j + 1]; num += 1;}

newMatrix[i][j] /= num;

num = 0;

}

}

}

};

int main() {

double duration[TIME] = { 0.0 };

for (int d = 0; d < TIME; d++) {

double start = omp\_get\_wtime();

calculate(matrix, newMatrix, MATRIXLENGTH, MATRIXLENGTH, 4);

double end = omp\_get\_wtime();

duration[d] = end - start;

printf("%f s ", duration[d]);

}

cout << "average time is : " << getAverage(duration) << endl;

return 0;

}

注：由于初始化成0.0和初始化成任意值，在矩阵中计算不会相差太多，故为了节省其初始化时间加快实验，选择在此直接初始化成0.0；其中的getAverage只是计算TIME次时间的平均。

**运行结果（10000\*10000矩阵计算四邻域值10次求平均，因为10000\*10000计算效果较明显，且只计算10次求平均，所以以此展示）**

1.328000 s 0.843000 s 0.569000 s 0.552000 s 0.529000 s 0.582000 s 0.630000 s 0.671000 s 0.661000 s 0.584000 s

average time is : 0.6949

**代码分析**

本代码不考虑初始化两个矩阵的时间，封装了一个calculate函数，传入一个参数ompNum用于控制开启多少线程，考虑四个角上和每条边上的元素应该求的是2邻域或者3邻域，所以在OpenMP开启时使用了一个私有变量num。为了时间计算的精准性，选用的是OpenMP自带的时间函数omp\_get\_wtime()。

**优化方案**

第一版指最初的代码，接下来有两种优化方式：

1. 第二个版本，如果不考虑边角，则不需要考虑2邻域或3邻域，可以去除私有变量num，这样能减少一部分计算量，由此考虑去除私有变量num来进行计算并且对比效率；

void calculate(double matrix[MATRIXLENGTH][MATRIXLENGTH], double newMatrix[MATRIXLENGTH][MATRIXLENGTH], int height, int length, int ompNum) {

omp\_set\_num\_threads(ompNum);

#pragma omp parallel

{

#pragma omp for

for (int i = 0; i < height; i++) {

for (int j = 0; j < length; j++) {

if (i > 0) {newMatrix[i][j] += matrix[i - 1][j];}

if (j > 0) {newMatrix[i][j] += matrix[i][j - 1];}

if (i < height - 1) {newMatrix[i][j] += matrix[i + 1][j];}

if (j < length - 1) {newMatrix[i][j] += matrix[i][j + 1];}

newMatrix[i][j] /= 4;

}

}

}

};

1. 第三个版本，由于会判断是否出界，所以使用了4个if，如果加上一圈padding，全部填充0，则可以去掉所有if语句，同时也去除了计算2邻域或3邻域的问题，但是会占用较高内存；

#define MATRIXLENGTH 1002

void calculate(double matrix[MATRIXLENGTH][MATRIXLENGTH], double newMatrix[MATRIXLENGTH][MATRIXLENGTH], int height, int length, int ompNum) {

omp\_set\_num\_threads(ompNum);

#pragma omp parallel

{

#pragma omp for

for (int i = 1; i < height - 1; i++) {

for (int j = 1; j < length - 1; j++) {

newMatrix[i][j] = (matrix[i - 1][j] + matrix[i + 1][j] + matrix[i][j - 1] + matrix[i][j + 1]) / 4;

}

}

}

};

1. 第四个版本，考虑将第三个版本的矩阵拉直成一个向量，即1002\*1002的向量，这样在第二个版本的基础上减少一次for循环（不考虑矩阵拉直和还原的时间），代码如下：

#define MATRIXLENGTH 1002

#define TIME 100

double matrix[MATRIXLENGTH\*MATRIXLENGTH] = { 0.0 };

double newMatrix[MATRIXLENGTH\*MATRIXLENGTH] = { 0.0 };

void calculate(double matrix[MATRIXLENGTH\*MATRIXLENGTH], double newMatrix[MATRIXLENGTH\*MATRIXLENGTH], int ompNum) {

omp\_set\_num\_threads(ompNum);

#pragma omp parallel

{

#pragma omp for

for (int i = MATRIXLENGTH; i < MATRIXLENGTH\*(MATRIXLENGTH - 1); i++) {

newMatrix[i] = (matrix[i - MATRIXLENGTH] + matrix[i + MATRIXLENGTH] + matrix[i - 1] + matrix[i + 1]) / 4;

}

}

};

**表格对比**

以下是实验对比，原题是对1000\*1000的矩阵进行实验，由于变量比较好改动，于是也进行了对10000\*10000的矩阵进行实验并且分析，其中次数表示运行多次并且取平均时间，其中时间单位为秒（s）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 矩阵大小/次数 | 第一版串行 | 第一版并行 | 第二版并行 | 第三版并行 | 第四版并行 |
| 1000\*1000/100 | 0.0173 | 0.00961001 | 0.00854999 | 0.00349001 | 0.00279 |
| 10000\*10000/10 | 2.1687 | 0.5959 | 0.5676 | 0.2687 | 0.1748 |

**实验分析**

1. 并行效率明显高于串行；
2. 从1000\*1000和10000\*10000的矩阵可以看出，虽然并行都是4线程跑，但是实际上优化效率分别是180%,364%，达不到4倍，可以得知并行开启关闭线程也需要时间，实际上运行量越大，优化效果越好；
3. 第一个版本考虑2,3邻域，设置了num作为私有变量，多进行了一些相加操作，这些操作也会增加一定复杂度，从第二版和第一版的并行时间可以看出，去掉一些可能不必要的操作是有一定优化的；
4. 第三个版本和前面几个版本效率相差极大，关键在于去除了边界判定的if操作，所以if操作非常消耗性能；
5. 并行的第一次任务启动会很慢（串行也是，测试效果差不多，不再贴出结果），猜测是因为第一次运行任务，需要启动一些环境和启动一些寄存器、堆栈等，第一次运行完后，因为程序未结束，所以寄存器和堆栈并未关闭，后续的程序也可以在这部分环境中运行，所以后面进行同样的操作速度会更快；
6. 将矩阵拉直，仅使用一次for循环，还可以进一步提升效率。

## Cuda编程

**代码**

#define MATRIXLENGTH 1002

#define TIME 100

int matrixSize = MATRIXLENGTH \* MATRIXLENGTH \* sizeof(double);

double \*\*matrix = new double\*[MATRIXLENGTH];

double \*\*newMatrix = new double\*[MATRIXLENGTH];

double \*matrixData = new double[MATRIXLENGTH\*MATRIXLENGTH];

double \*newMatrixData = new double[MATRIXLENGTH\*MATRIXLENGTH];

double \*\*d\_x, \*\*d\_y;

\_\_global\_\_ void kernel(double \*\*matrix, double \*\*newMatrix, int n) {

int i = threadIdx.x + blockIdx.x\*blockDim.x;

int j = threadIdx.y + blockIdx.y\*blockDim.y;

if (i > 1 && j > 1 && i < n-1 && j < n-1) {

newMatrix[i][j] = (matrix[i - 1][j] + matrix[i + 1][j] + matrix[i][j - 1] + matrix[i][j + 1]) / 4;

}

}

void calculate() {

cudaMalloc(&d\_x, MATRIXLENGTH \* sizeof(double\*));

cudaMalloc(&d\_y, MATRIXLENGTH \* sizeof(double\*));

double \*d\_x\_data, \*d\_y\_data;

cudaMalloc(&d\_x\_data, matrixSize);

cudaMalloc(&d\_y\_data, matrixSize);

cudaMemcpy(d\_x\_data, matrixData, matrixSize, cudaMemcpyHostToDevice);

cudaMemcpy(d\_x, matrix, MATRIXLENGTH \* sizeof(double\*), cudaMemcpyHostToDevice);

cudaMemcpy(d\_y, newMatrix, MATRIXLENGTH \* sizeof(double\*), cudaMemcpyHostToDevice);

dim3 blockSize(3, 3);

dim3 threadSize(5,5);

kernel << <threadSize, blockSize >> >(d\_x, d\_y, MATRIXLENGTH);

// device拷贝到host

cudaMemcpy(newMatrixData, d\_y\_data, matrixSize, cudaMemcpyDeviceToHost);

}

int main() {

double duration[TIME] = { 0.0 };

for (int d = 0; d < TIME; d++) {

double start = omp\_get\_wtime();

calculate();

double end = omp\_get\_wtime();

duration[d] = end - start;

printf("%f s ", duration[d]);

}

cout << "average time is : " << getAverage(duration) << endl;

system("pause");

return 0;

}

**运行结果**

0.575142 s 0.000013 s 0.000015 s … 06 s 0.000006 s 0.000006 s 0.000006 s 0.000006 s 0.000007 s 0.000006 s 0.000006 s 0.000006 s 0.000006 s

average time is : 0.00576268

**优化方案**

暂无。里的并行计算与OpenMP的并行不相同，不是采用for循环，所以优化for循环意义不大。

**实验分析**

cuda编程的并行与OpenMP不同，是将内存中的量传入显存，由显存的gpu单元对每个量单独进行运算，没有显式的for循环，实际上全部都在并行，所以在gpu配置较好的情况下计算速度应该更快（本机配置cpu为i5-6500，gpu为gtx-650Ti，可能是由于显卡较差，所以速度没有使用OpenMP的计算快）。