**中山大学数据科学与计算机院本科生实验报告**

**（2020学年秋季学期）**

课程名称：**高性能计算程序设计**  任课教师：**黄聃 批改人：**

|  |  |  |  |
| --- | --- | --- | --- |
| 年级+班级 | **2018级五班** | 专业（方向） | **计算机科学与技术** |
| 学号 | **18340126** | 姓名 | **罗仁良** |
| Email | **luorliang@mail2.sysu.edu.cn** | 完成日期 | **2020年12月5日** |

1. **实验目的**
   1. 基于Phreads的parallel\_for函数替换heated\_plate\_openmp中的for循环分解
   2. 基于MPI的进程并行
   3. 使用valgrind性能分析
2. **实验过程和核心代码**
   1. 基于Pthreads的for循环分解

在实验4中已经完成了parallel\_for函数，在接下的实验主要是完成每个循环体的函数设计。实验中一共设计了assignVec、reduction、set\_w、save2u、new\_w、compute\_diff六个函数，对for循环进行拆分。

* + 1. assignVec函数

1. 参数

**//** 表示行向量/列向量

typedef enum vecType

{

col,

row

} vecType;

// 实现矩阵行向量、列向量 赋值

// 给type类型的向量的（index为向量在矩阵中的索引）赋值为value

typedef struct arg1

{

vecType type;

int index;

double value;

} arg1;

1. 函数主体

// 判断向量类型 行/列

if (parg2->type == col)

{

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

{

// 向量赋值

w[i][parg2->index] = parg2->value;

}

}

else

{

// 向量赋值

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

w[parg2->index][i] = parg2->value;

}

1. 循环拆分

对参数进行初始化，直接调用parallel\_for函数即可实现for循环拆分。下面以一个for循环的拆分代码作为示例。

arg1 \*arg = malloc(sizeof(arg1));

arg->type = col;

arg->index = 0;

arg->value = 100.0;

parallel\_for(1, M - 1, 1, assignVec, arg, num\_threads);

上面这段代码等价于openmp的代码如下

#pragma omp for

for (i = 1; i < M - 1; i++)

{

w[i][0] = 100.0;

}

* + 1. reduction函数

这部分函数设计有一点需要注意的是，每个线程mean这个共享变量时需要互斥，每次计算都会改变mean值，需要保证每次线程访问的mean值是最新的。为为防止出现竞争问题，增加一个互斥量，保证线程访问的互斥性。

1. 参数

typedef struct arg2

{

vecType type;

int index1;

int index2;

double \*result;

} arg2;

reduction是完成两个向量相加，需要两个index表示向量在矩阵中的行(列)。

1. 函数主体

if (parg2->type == col)

{

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

{

pthread\_mutex\_lock(&mutex\_sum);

\*(parg2->result) = \*(parg2->result) + w[i][parg2->index1] + w[i][parg2->index2];

pthread\_mutex\_unlock(&mutex\_sum);

}

}

else

{

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

{

pthread\_mutex\_lock(&mutex\_sum);

\*(parg2->result) = \*(parg2->result) + w[parg2->index1][i] + w[parg2->index2][i];

pthread\_mutex\_unlock(&mutex\_sum);

}

}

1. 循环拆分

arg2 \*arg\_reduction = malloc(sizeof(arg2));

arg\_reduction->type = col;

arg\_reduction->index1 = 0;

arg\_reduction->index2 = N - 1;

arg\_reduction->result = &mean;

parallel\_for(1, M - 1, 1, reduction, arg\_reduction, num\_threads);

* + 1. set\_w函数、save2u函数和new\_w函数

这两个函数就是一般的for循环展开，为减少参数的传递，实验中把w、u改为了全局变量。

// save old in u

void \*save2u(void \*arg)

{

targ \*parg1 = (targ \*)arg;

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

for (int j = 0; j < N; j++)

u[i][j] = w[i][j];

return NULL;

}

// new estimate

void \*new\_w(void \*arg)

{

targ \*parg1 = (targ \*)arg;

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

for (int j = 1; j < N - 1; j++)

w[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1]) / 4.0;

return NULL;

}

// new estimate

void \*new\_w(void \*arg)

{

targ \*parg1 = (targ \*)arg;

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

for (int j = 1; j < N - 1; j++)

w[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1]) / 4.0;

return NULL;

}

* + 1. compute\_diff函数

这部分函数的作用是找到全局最大的diff，在线程并行中，先是每个线程找到局部的最大local\_diff，再和全局的diff比较更新。这个过程中要读写全局的diff变量，需要保证互斥访问，使用互斥量解决这个问题。

1. 函数主体

pthread\_mutex\_t mutex\_diff = PTHREAD\_MUTEX\_INITIALIZER;

void \*compute\_diff(void \*arg)

{

targ \*parg1 = (targ \*)arg;

double \*diff = (double \*)(parg1->arg);

double local\_diff = 0.0;

// 局部diff

for (int i = parg1->first; i < parg1->last; i += parg1->stride)

for (int j = 1; j < N - 1; j++)

if (local\_diff < fabs(w[i][j] - u[i][j]))

{

local\_diff = fabs(w[i][j] - u[i][j]);

}

// 互斥访问

pthread\_mutex\_lock(&mutex\_diff);

if (\*diff < local\_diff)

\*diff = local\_diff;

pthread\_mutex\_unlock(&mutex\_diff);

return NULL;

}

1. 循环拆分

parallel\_for(1, M - 1, 1, compute\_diff, &diff, num\_threads);

* 1. 基于MPI的进程并行

将原程序改成基于MPI的进程并行程序主要需要做的是每次迭代过程，每个进程只负责部分的w更新，计算完成后把更新后的w发送到0号进程上，最后0号进程再把所有更新的w广播到所有进程上。

* + 1. 初始化

为避免多次传递的额外开销，所有参数的初始化都是在0号进程中完成的，最后0号进程再广播初始化后的参数，主要包括w，mean等其他变量的初始化。

广播初始化后的w

MPI\_Bcast(w, M \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

* + 1. 保存w的旧值到u中

每次迭代过程中，更新后的w值都是汇总到0号进程，所以只需要0号把w赋值到u中，再广播同步u值即可。

if (my\_rank == 0) // w -> u

{

{

for (int i = 0; i < M; i++)

{

for (int j = 0; j < N; j++)

u[i \* N + j] = w[i \* N + j];

}

}

MPI\_Bcast(u, M \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

else MPI\_Bcast(u, M \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

* + 1. 更新w值

每个进程负责计算部分行的w值更新，最后汇总到0号进程上。

// 多进程并行更新w的值

index my\_index;

my\_index = get\_task(1, M - 1, my\_rank, num\_threads);

for (i = my\_index.first; i < my\_index.last; i++)

{

for (j = 1; j < N - 1; j++)

{

w[i \* N + j] = (u[(i - 1) \* N + j] + u[(i + 1) \*N +j] + u[i \* N + j - 1] + u[i \* N + j + 1]) / 4.0;

}

}

// 汇总更新后的w到0号进程

if (my\_rank == 0)

{

for (int i = 1; i < num\_threads; i++)

{

index i\_index;

i\_index = get\_task(1, M - 1, i, num\_threads);

MPI\_Recv(w + i\_index.first \* N, (i\_index.last - i\_index.first) \* N, MPI\_DOUBLE, i, i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

}

else

{

MPI\_Send(w + my\_index.first \* N, (my\_index.last - my\_index.first) \* N, MPI\_DOUBLE, 0, my\_rank, MPI\_COMM\_WORLD);

}

* + 1. 计算diff

同计算w的方式，每个进程负责计算出局部的最大my\_diff，最后调用MPI\_Reduce函数进行归一，找到最大的diff。

MPI\_Reduce(&my\_diff, &diff, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

* 1. 使用Valgrind进行性能分析

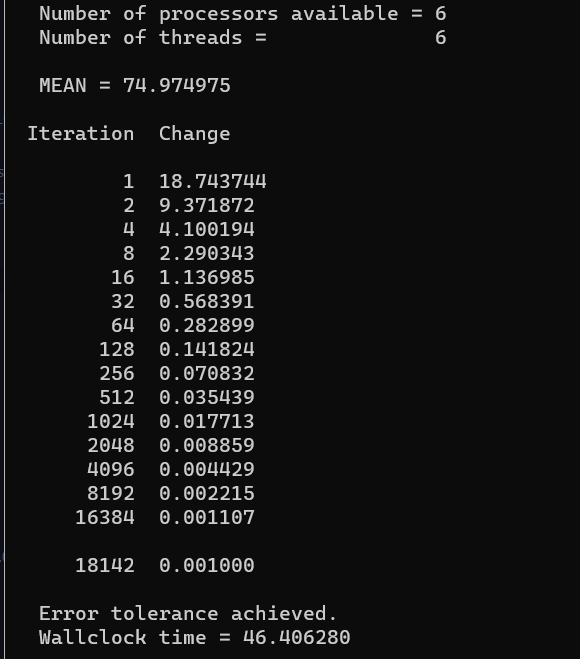
详细见实验结果。

1. **实验结果**
   1. 执行时间
      1. heated\_plate\_openmp

矩阵大小

|  |  |  |  |
| --- | --- | --- | --- |
| 时间（s）  线程数量 | 250 | 500 | 1000 |
| 1 | 5.375164 | 36.581361 | 156.969325 |
| 2 | 2.749534 | 18.495933 | 81.400048 |
| 4 | 1.474791 | 9.918650 | 47.641823 |
| 6 | 1.190890 | 8.503088 | 46.406280 |

**注：由于实验电脑性能有限，最高支持6线性，为防止线程过多，最高只测试了6线程的情况；**

可以看到随着线程数量的增加，运行时间也基本成比例关系缩短。

* + 1. 基于phtreads

|  |  |  |  |
| --- | --- | --- | --- |
| 时间（s）  矩阵大小  线程数量 | 250 | 500 | 1000 |
| 1 | 7.060504 | 41.862795 | - |
| 2 | 4.643751 | 22.972735 | - |
| 4 | 4.415231 | 14.811484 | 52.077878 |
| 6 | 6.990557 | 17.154135 | 47.594453 |

**注：1000\*1000部分耗时太久，没有测试所有情况。**

由表中数据可以1-2-4的加速效果比较明显符合预期。但在6线程的运行时间反而更长，与预期不符，但随着数据量的增加，6线程运行又更快了，这个现象比较奇怪。我猜测的主要原因是线程负载均衡或者线程调度的额外开销导致的。

* + 1. 基于MPI

由于MPI的程序运行较慢，只做了小规模矩阵的性能测试。

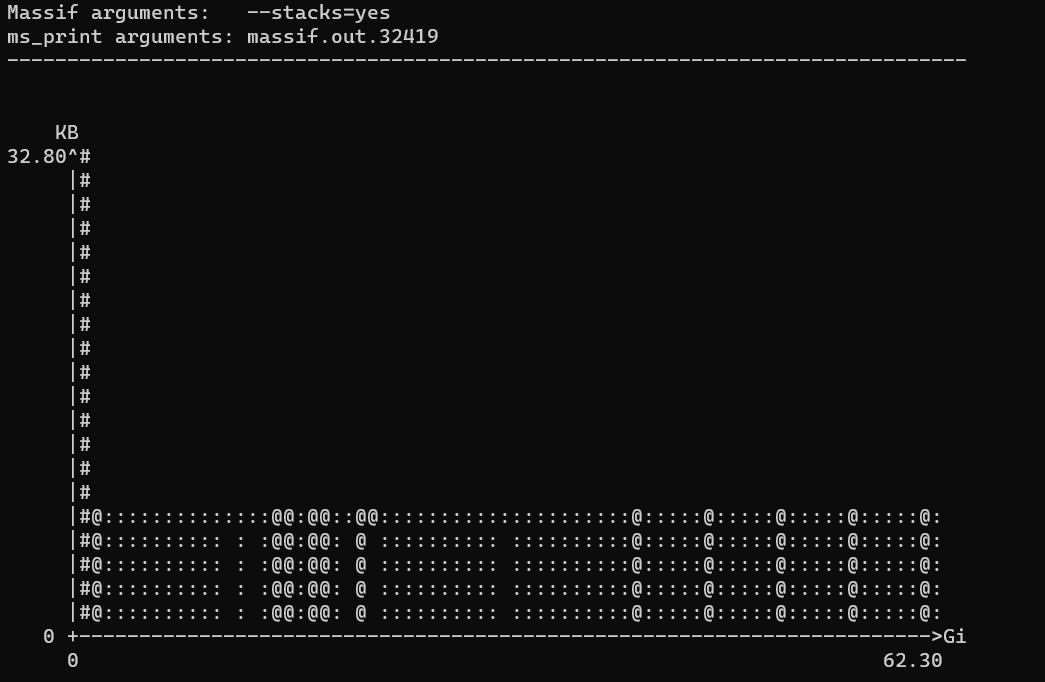
矩阵大小

|  |  |
| --- | --- |
| 时间（s）  线程数量 | 250 |
| 1 | 7.060504 |
| 2 | 14.231958 |
| 4 | 29.433239 |
| 6 | 56.517767 |

由测试结果可以看到，在数据量比较小时，通信的额外开销是不可接受的。

* 1. 内存分析

在500\*500 4线程下运行的结果

* + 1. OpenMP
    2. Phtreads
    3. MPI

OpenMP对内存的使用最少，这是基于内存共享的通信的优势。Phreads的内存使用最多，推测是在参数传递，引起的额外内存开销。

1. **实验感想**

完成本次实验进一步加强了phtreads和MPI在并行编程的使用，同时学习了基础的valgrind的应用。这次在MPI编程的过程中遇到了较大的问题，主要出在数据划分通信上，实验中由尝试pack的方式进行数据传输，但是失败了，最后采用较慢的点对点通信完成数据传输。下来还需要继续研究，争取完善程序。