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

1.1 基于 Phreads 的 parallel_for 函数替换 heated_plate_openmp 中的 for 循环分解

1.2 基于 MPI 的进程并行

1.3 使用 valgrind 性能分析

2. 实验过程和核心代码

2.1 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。

2.1.1 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;
```

2）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;
        }
```

3）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;
    }
```

2.1.2 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)。

2）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);
          }
       }
```

3）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);
```

2.1.3 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;
    }
```

2.1.4 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;
    }
```

2） 循环拆分

```
    parallel_for(1, M - 1, 1, compute_diff, &diff, num_threads);
```

2.2 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。

2.2.1 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);
```

2.2.2 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);
```

2.2.3 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);
        }
```

2.2.4 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);
```

2.3 使用 Valgrind 进行性能分析

详细见实验结果。

3. 实验结果

3.1 执行时间

3.1.1 heated_plate_openmp

时间（s）矩阵大小 / 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。

```
 Number of processors available = 6
 Number of threads =              6

 MEAN = 74.974975

 Iteration  Change

        1  18.743744
        2  9.371872
        4  4.100194
        8  2.290343
       16  1.136985
       32  0.568391
       64  0.282899
      128  0.141824
      256  0.070832
      512  0.035439
     1024  0.017713
     2048  0.008859
     4096  0.004429
     8192  0.002215
    16384  0.001107

    18142  0.001000

 Error tolerance achieved.
 Wallclock time = 46.406280
```

3.1.2 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。

3.1.3 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。

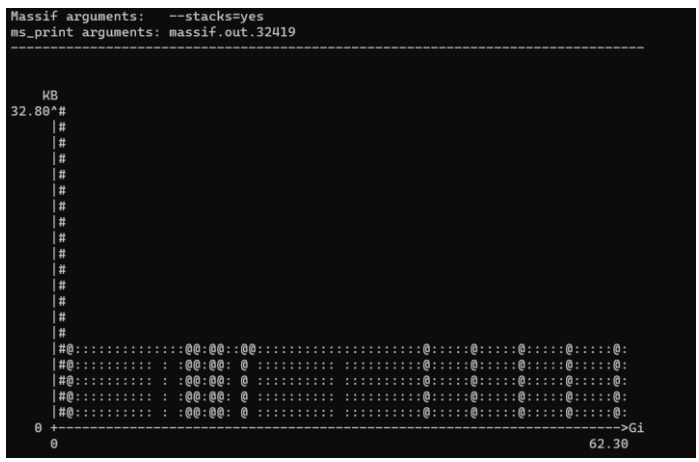
时间（s） 矩阵大小 / 线程数量	250
1	7.060504
2	14.231958
4	29.433239
6	56.517767

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。

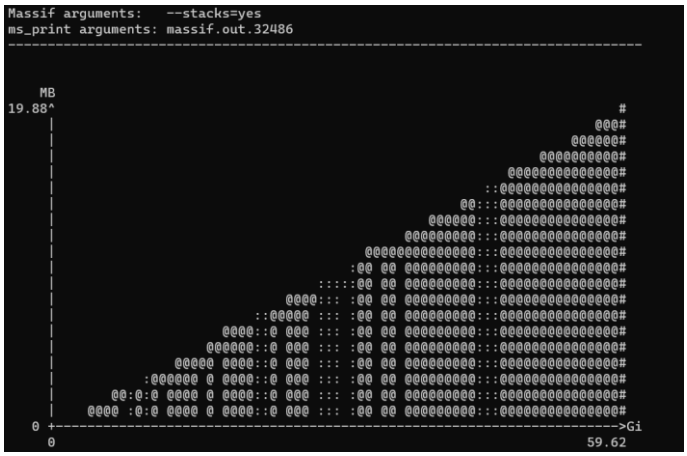
3.2 内存分析

在 500*500 4 线程下运行的结果

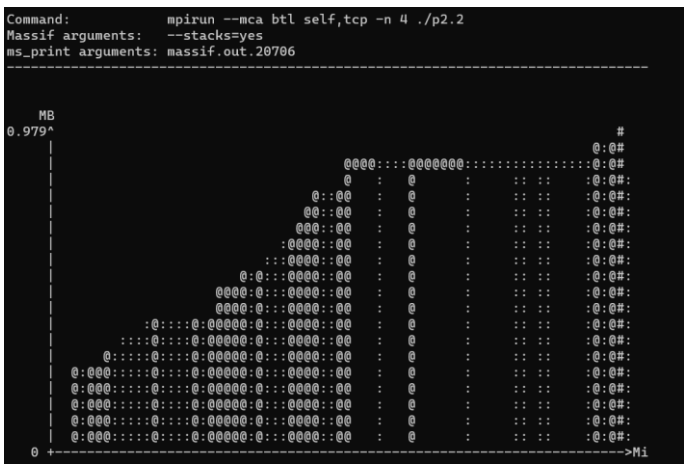
3.2.1 OpenMP



3.2.2 Phtreads



3.2.3 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。

4. 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。