《并行算法设计与分析》期末考查题实现与分析报告

计科210X 甘晴void 学号：202108010XXX

1 题目重述

Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or anysuitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrices from the Matrix Market (http://math.nist.gov/MatrixMarket/) and test the performance of your implementation as a function of matrix size and number of threads.

考虑一个以压缩行格式存储的稀疏矩阵（你可以在网上找到这种格式的描述，也可以在稀疏线性代数上找到任何合适的文本）。编写一个OpenMP程序来计算这个矩阵与向量的乘积。从矩阵市场下载矩阵样本(http://math.nist.gov/MatrixMarket/)并将实现的性能作为矩阵大小和线程数的函数进行测试。

2 思路分析

2.1 问题建模

我们可以对问题进行简单重新建模。对于矩阵A（维度`m*n`），和向量B（维度`n*1`），计算其乘积结果C（维度`m*1`）。特别地，该矩阵A有特性稀疏，为压缩行格式存储的稀疏矩阵。

以题目提供的矩阵为例，稀疏性体现如下：该矩阵为3140*3140维度，其非零项有543162项，非零项占比为5.5%左右，即该矩阵绝大部分都是零项。

2.2 CSR格式

CSR格式定义如下

```c
typedef struct
{
    int n_rows;     // 矩阵的行数
    int n_cols;     // 矩阵的列数
    int n_nonzero; // 非零元素的个数
    int *row_ptr;  // 行指针数组
    int *col_ind;  // 列索引数组
    double *val;   // 非零元素值数组
} crs_matrix_t;
```

其中前三个参数比较好理解，后面是3个一维特征数组

- val数组：按行开始数，依次存储每一行的非零元素；
- col_ind数组：每一个非零元素的列索引值；
- row_ptr数组：每一行的第一个非零元素在data数组中的索引值（注意：只统计每一行第一个非零元）；

使用这三个一维特征数组可以唯一表示这个稀疏矩阵。

下面是一个范例：

```
对于如下5*4维度的矩阵
1 0 2 0
0 0 3 0
0 0 0 0
0 5 6 0
0 0 0 3
其三个一维特征数组为：
val     = [1,2,3,5,6,4]
col_ind = [0,2,2,1,2,3]
row_ptr = [0,2,3,3,5,6]
```

2.3 CSR格式矩阵存储

明确了CSR的格式之后，将一个mtx类型的稀疏矩阵转换为一个CSR格式的矩阵就比较好实现了。

其处理逻辑编程实现如下：

```c
// 读取非零元素的行索引、列索引和值
    int row = 0, idx = 0;
```

```
    A.row_ptr[0] = 0;
    for (int i = 0; i < n_nonzero; i++)
    {
        int r, c;
        double v;
        fscanf(f, "%d %d %le", &r, &c, &v);
        A.col_ind[i] = c - 1; // 存储列索引（减1）
        A.val[i] = v;         // 存储值
        while (r > row + 1)
        {
            A.row_ptr[row + 1] = i;
            row++;
        }
        row = r - 1;
    }
    while (row < n_rows)
    {
        A.row_ptr[row + 1] = n_nonzero;
        row++;
    }
```

这种处理方式要求读入的矩阵必须按照行优先而不是列优先的顺序（即a11,a12,a13这样的顺序而不是a11,a21,a31这样的顺序），故题目中所给出的矩阵还需要经过预处理，我将在后面详细叙述。

2.4 CSR格式访问

对于一个矩阵，我们希望能够很清晰地看到它的存储，以确定该矩阵被正确保存了，这也是可解释性的一部分。虽然这不是题目的要求，但我还是做了探究。

确定矩阵A中第i行j列的数m需要i,j,m三个参数。

行索引数组允许我们看到某一行i的第一个非零数在val数组中的位置，而相邻两行的行索引数组就可以确定在该行的所有非零数在val的位置，接下来只要确定它们的列索引j即可。

我们用逐行的方法去遍历，对于每一行i，显然该行非零数在val数组的下标从`A->row_ptr[i]`到`A->row_ptr[i + 1]`（取不到右边界），我们设这个下标为a，那么值m为`val[a]`，其列j为`col_ind[a]`，至此，i,j,m都明确了，该位置的向量值也就出来了。

如果只对矩阵的左上角`m*n`感兴趣，可以只对这一部分进行验证。

用这样的方法可以写出一个小范围验证矩阵是否被正确保存的代码。

```
for (int i = 0; i < rows; i++)
        {
            double *temp_row = new double[A->n_cols]; // 临时数组，逐行输出
            for (int j = 0; j < cols; j++)
                temp_row[j] = 0;
            for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++) // j为该数在data中的位置
            {
                if (A->col_ind[j] < cols) // A->col_ind[j]为该数的列号，i为该数的行号
                {
                    temp_row[A->col_ind[j]] = A->val[j];
                }
            }
            for (int j = 0; j < cols; j++)
                printf("%20.8f", temp_row[j]);
            printf("\n");
        }
```

对于矩阵的部分为：

```
%%MatrixMarket matrix coordinate real general
3140 3140 543162
1 1  6.9033670000000e-01
2 1  1.6855795000000e-03
3 1  2.3404025000000e-03
7 1  6.3543841000000e-03
```

尝试运行如下：

```

```

2.5 CSR格式运算

对于矩阵A（维度`m*n`），和向量B（维度`n*1`），计算其乘积结果C（维度`m*1`）。最终结果是m行。计算方法是，原稀疏矩阵A的每一行（维度`1*n`）都与向量B（维度`n*1`）相乘，这样构成结果矩阵C的每一行。

难点是如何刻画原稀疏矩阵的每一行的每个非零元素，这个其实前面我们在CSR格式访问时提及过了，可以看作先访问到这个元素，再与x向量相应位置做乘积，把结果加起来。

下面是一个简易的实现逻辑。

```
for (int i = 0; i < A->n_rows; i++)
    {
        double sum = 0.0;
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            sum += A->val[j] * x[A->col_ind[j]];
        }
        y[i] = sum;
    }
```

2.6 性能衡量

分别计算串行与并行所用时间，并计算优化加速比。

优化加速比= 串行时间 / 并行时间

3 代码实现与结果分析

3.0 测试环境

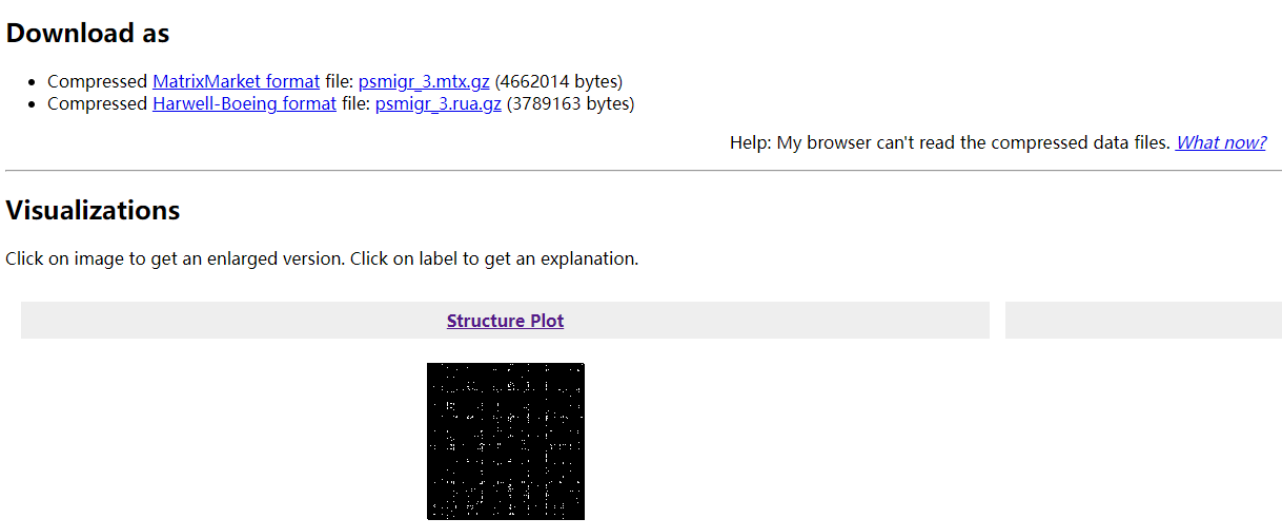
128核CPU，每核逻辑线程数2。

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   48 bits physical, 48 bits
virtual
CPU(s):                          128
On-line CPU(s) list:             0-127
Thread(s) per core:              2
Frequency boost:                 enabled
CPU MHz:                         1500.000
CPU max MHz:                     2800.0000
CPU min MHz:                     1500.0000
BogoMIPS:                        5589.07
```

```
Virtualization:                  AMD-V
L1d cache:                       2 MiB
L1i cache:                       2 MiB
L2 cache:                        32 MiB
L3 cache:                        512 MiB
NUMA node0 CPU(s):               0-31,64-95
NUMA node1 CPU(s):               32-63,96-127
```

未加说明，使用提供的3140*3140矩阵进行测试。

下图是该矩阵的直观概览



我们可以从 矩阵市场 获得更多矩阵。

我另外下载了几个数量级差异较大的，目前共有以下三种供比较。

- fs_760_3.mtx（760 x 760, 5976 entries,非零率1.0%）
- psmigr_3.mtx（3140 x 3140，543162 entries,非零率5.5%）
- fidapm37.mtx（9152 x 9152, 765944 entries,非零率0.91%）

3.1 串行实现

串行显然就是直接使用上面的代码即可。

```c
// 矩阵向量乘法(串行)
void crs_matrix_vector_product_serial(const crs_matrix_t *A,
double *x, double *y)
{
    for (int i = 0; i < A->n_rows; i++)
    {
        double sum = 0.0;
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            sum += A->val[j] * x[A->col_ind[j]];
        }
        y[i] = sum;
    }
}
```

3.2 并行实现

并行有多个层次可以考虑，由于这里实际的算法部分只有两重for循环，我们可以从这两层for循环去考虑优化。共有如下思路：

1. 向量化：利用SIMD（单指令多数据）指令同时对多个数据元素执行操作。这可以通过确保数据对齐和使用编译器内联函数或启用编译器自动向量化（例如，对于Intel CPU使用`-O3 -mavx`）来实现。【这里不太好实现】
2. 私有变量：在OpenMP中，可以将循环变量声明为每个线程的私有变量，以避免假共享。这可以通过在`#pragma omp parallel for`指令中使用`private`子句来完成。【这个后面会探究】
3. 归约：由于每个线程计算一个部分和，可以使用OpenMP的`reduction`子句来安全地合并线程的和，减少同步开销。【这个需要做内层并行时考虑】
4. 线程亲和性：将线程绑定到特定的CPU核心可以减少缓存冲突并提高性能。这可以通过使用`omp_set_affinity_policy`和`omp_get_thread_num`来实现。【这个提升不是很大】
5. 分块策略：如果矩阵非常大，可以考虑使用分块算法，如分块雅可比方法或块循环数据分布，来跨多个线程或进程并行化计算。【稀疏矩阵与此不同】
6. 负载均衡：如果矩阵的稀疏性不均匀，某些行可能有远多于其他行的非零元素。可以使用OpenMP的动态调度（`schedule(dynamic)`）来平衡线程间的工作负载。【这个想法看起来能work，值得试试】
7. 缓存优化：优化数据访问模式以利用缓存局部性。例如，预取数据元素或重新排序循环以减少缓存缺失。【这个似乎比较难实现】

8. 混合并行：在集群或多节点系统上，结合OpenMP和MPI（消息传递接口）进行分布式内存并行。【MPI不在讨论范围内】

经过初步判断，我确定了几条可能可行的思路进行验证，过程如下

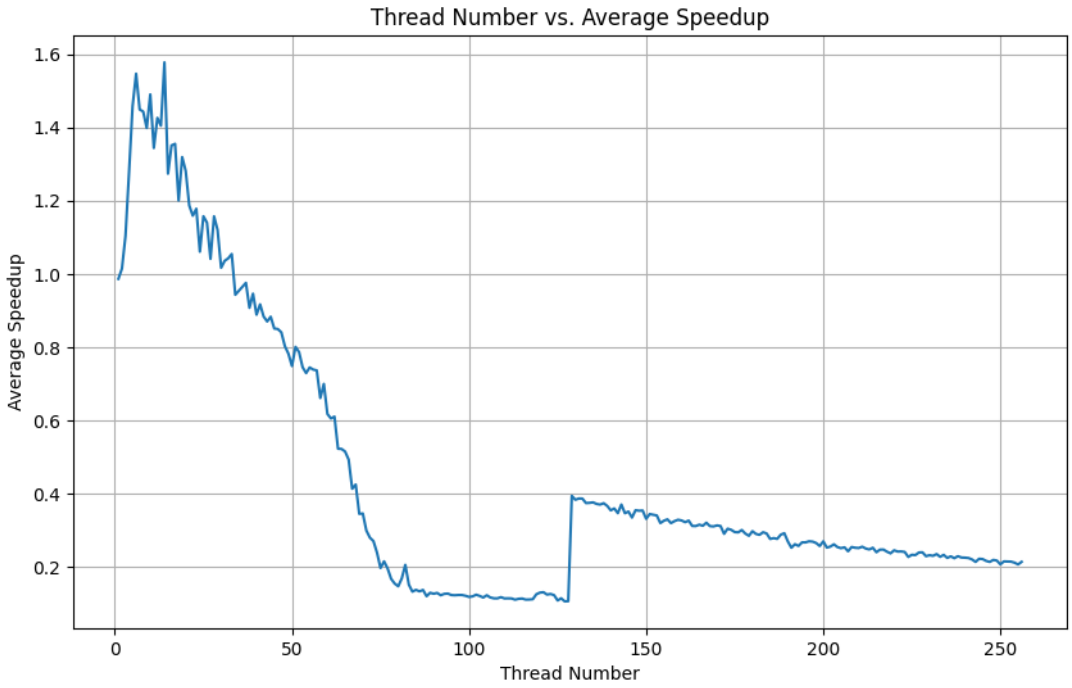
（1）并行化外层循环

这是最容易想到的，由于外层循环相对独立，不同的外层循环之间并没有访问临界区，故不需要做额外的限制处理，直接使用openmp对外层循环并行即可。代码实现如下：

```
// 矩阵向量乘法(并行化外层循环)
void crs_matrix_vector_product_parallel(const crs_matrix_t *A, double *x, double *y)
{
    omp_set_num_threads(num_thread);
#pragma omp parallel for
    for (int i = 0; i < A->n_rows; i++)
    {
        double sum = 0.0;
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            sum += A->val[j] * x[A->col_ind[j]];
        }
        y[i] = sum;
        // cout << "i= " << i << " " << omp_get_thread_num() << "    ";
        // if (i % 10 == 0) cout << endl;
    }
}
```

写python脚本跑多次取平均值，并计算优化加速比。

下面是对于每个并行数跑100次平均的加速比-并行数图像（并行数由1到256），可见最好的情况大约能跑到1.6的加速比，但是并行数需要不能太多。



推测并行数太大的时候，并行线程创建的开销过大，导致效果特别差。其中100与150中间的阶跃在128处，与系统的CPU核数相同，不是巧合。

但是上图的测试时间太长了，之后不测100次了，改为测更少次数。

（2）负载均衡（动态调度）

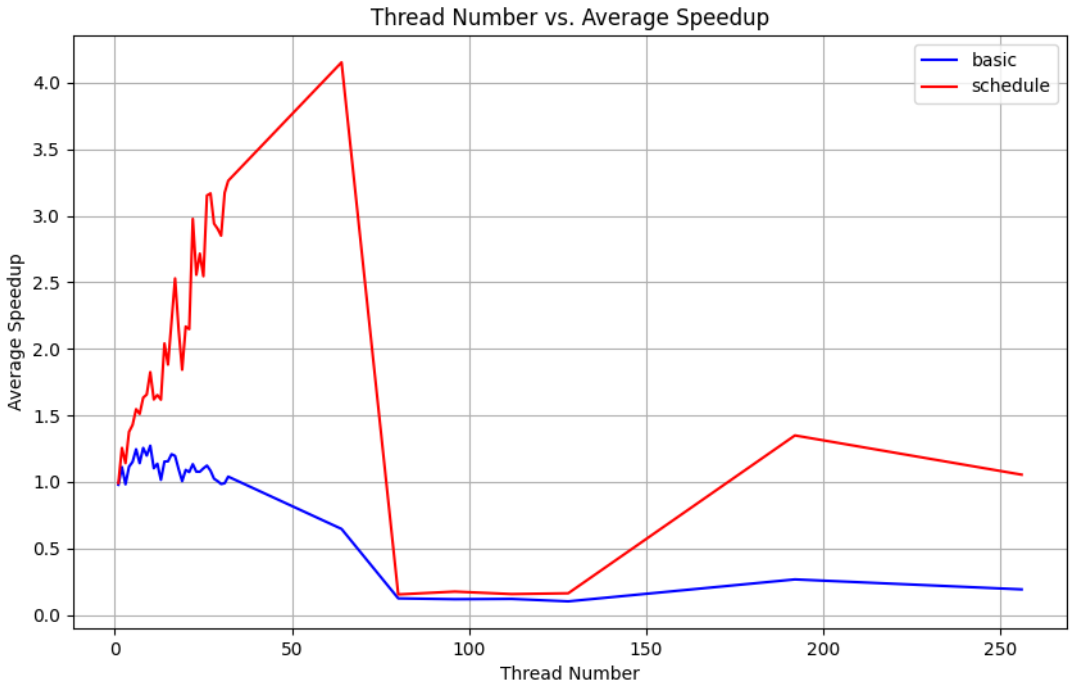
矩阵的稀疏性不均匀，某些行可能有远多于其他行的非零元素。这也是本问题比较显著的一个特征。

可以使用OpenMP的动态调度（`schedule(dynamic)`）来平衡线程间的工作负载。

只需修改openmp调用方式即可

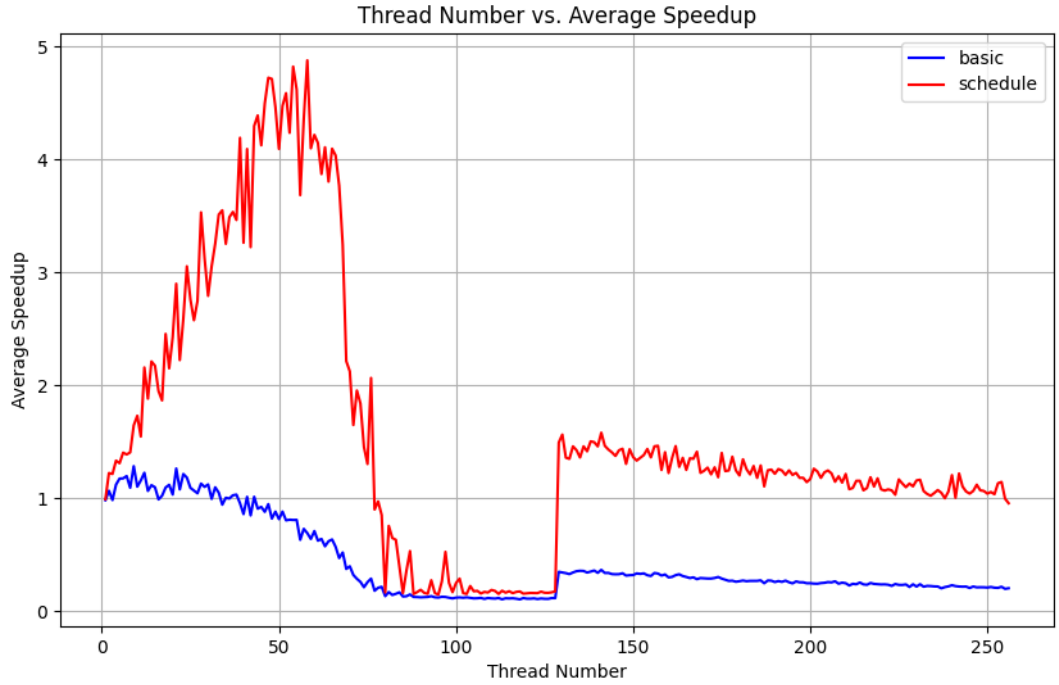
```
1  #pragma omp parallel for schedule(dynamic)
```

使用题目提供矩阵，测试不同并行数（1-32，64，80，96，112，128，192，256），各测20次取平均加速比，进行大致测量，结果如下。



可见，动态调度实现负载均衡对性能优化非常大。其阶跃点为64，这也不是巧合，为CPU核数的一半，推测与调度算法有关。其中32-64和64-80这两段有重要意义，但并没有测点，在得到这张图后，我意识到可能需要把这一段测完整，得到更精确的结果。

使用题目提供矩阵，测试不同并行数（1-256），各测20次取平均加速比，进行大致测量，结果如下。最好的情况下，加速比能达到接近5，这是比较好的优化了。

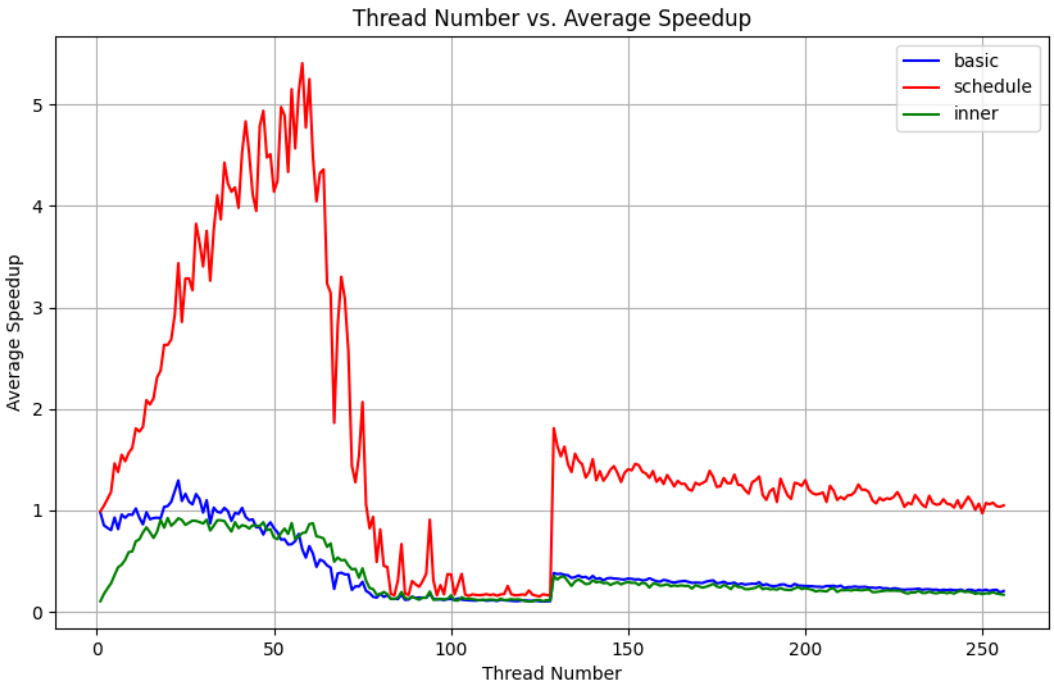


（3）内层并行

在内层计算的时候，也可以采用并行的方法来计算，但这里需要注意，由于sum是临界区变量，这里需要对sum添加一个规约，以避免对于临界区的访问。

```
#pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < A->n_rows; i++)
    {
        double sum = 0.0;
#pragma omp parallel for reduction(+ : sum)
        for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
        {
            sum += A->val[j] * x[A->col_ind[j]];
        }
        y[i] = sum;
    }
```

但是，意料之外地，内层并行的效果不是很好，如图是在动态调度的基础上添加内层并行的结果，很大程度上甚至没有最基础的外层并行好，可见，在这里内层并行起了反效果。



其实这里也可以理解，如果每个内层的计算都再分出一个线程来，这个线程其实只执行了一个语句，但它的开销确实很大的，这样反而不如直接串行完成了。

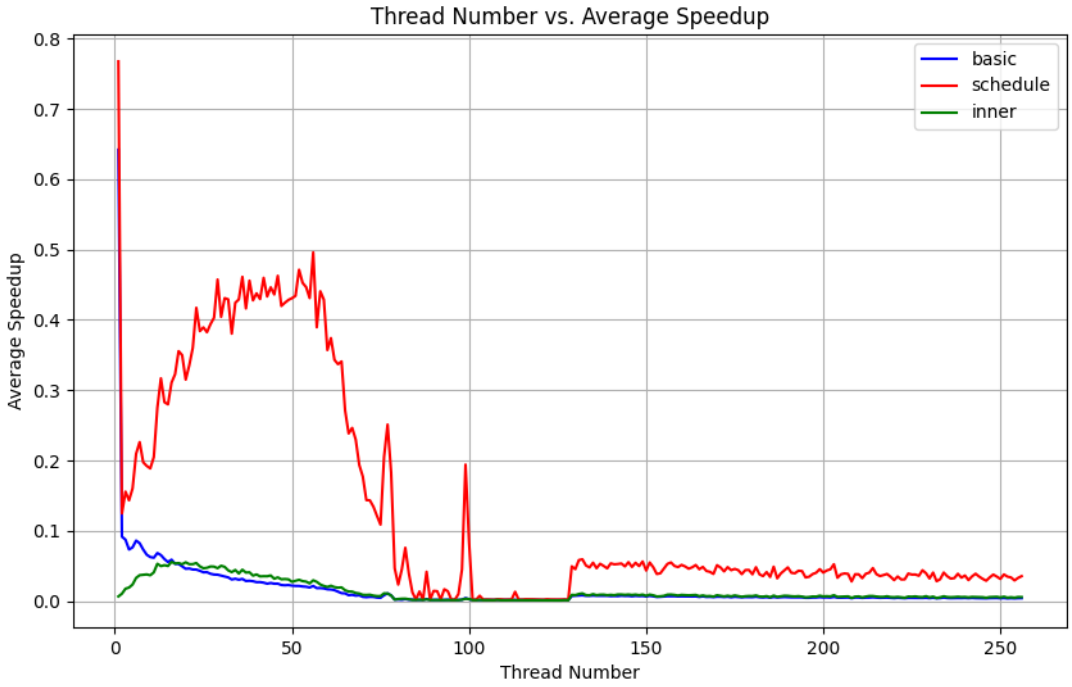
（4）矩阵规模影响

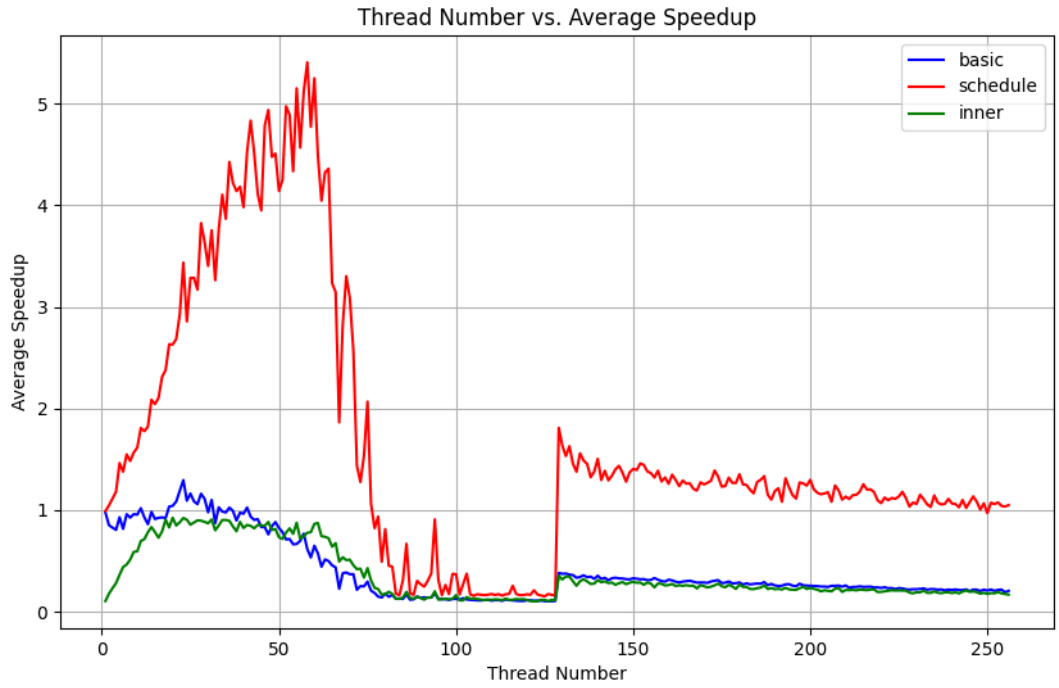
综上可以发现，使用并行化的外层循环+动态调度可以得到较好的结果，我们大致可以确定最好的并行数大致在64上下，接下来可以进一步探究在最好并行数的情况下，不同规模矩阵的加速比。

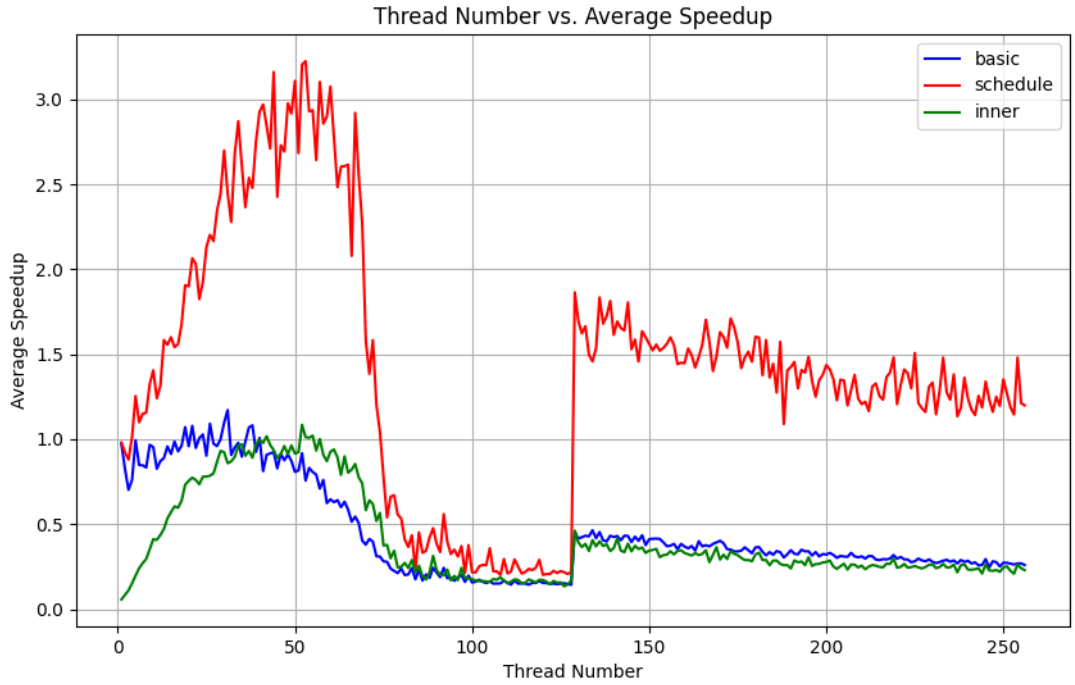
我下载了3个矩阵，这是矩阵的大小信息，可以由非零率看到，都是稀疏矩阵。

- fs_760_3.mtx（760 x 760, 5976 entries,非零率1.0%）
- psmigr_3.mtx（3140 x 3140，543162 entries,非零率5.5%）
- fidapm37.mtx（9152 x 9152, 765944 entries,非零率0.91%）

加速比与并发数关系图如下：







可见在矩阵规模较小时，并行优化的加速比并不高，甚至在效率上不如串行，但随着矩阵规模逐渐变大，优化调度的并行的优势逐渐变大，但其加速比会有一个先变大后减小的过程。但是如果不经优化调度，并行算法甚至会比串行算法更差。

其中，考虑动态调度，且较好的情况下，加速比达到过8

```
(base) root@dea92cb95233:/home/datatry/openMP# ./mv 64 psmigr_3
set thread number : 64
set matrix name   : psmigr_3
system max concurrency = 128
parallel-basic  accuracy: 1      (outcome accurate)
parallel-2  accuracy: 1      (outcome accurate)
parallel-3  accuracy: 1      (outcome accurate)
serial time   =1.8277ms

parallel time =3.53468ms
speedup1      =0.517077

parallel time2=0.219468ms
speedup2      =8.32788

parallel time3=2.01641ms
speedup3      =0.906414
```

连续多次运行的平均情况也接近5

```
(base) root@dea92cb95233:/home/datatry/openMP# ./mv 64 psmigr_3
set thread number : 64
set matrix name   : psmigr_3
system max concurrency = 128
parallel-basic  accuracy: 1      (outcome accurate)
parallel-2  accuracy: 1      (outcome accurate)
parallel-3  accuracy: 1      (outcome accurate)
serial time   =1.8565ms

parallel time =2.88609ms
speedup1      =0.643257

parallel time2=0.374135ms
speedup2      =4.9621

parallel time3=2.82333ms
speedup3      =0.657556
(base) root@dea92cb95233:/home/datatry/openMP# ./mv 64 psmigr_3
set thread number : 64
set matrix name   : psmigr_3
system max concurrency = 128
parallel-basic  accuracy: 1      (outcome accurate)
parallel-2  accuracy: 1      (outcome accurate)
parallel-3  accuracy: 1      (outcome accurate)
serial time   =1.85865ms

parallel time =2.8581ms
speedup1      =0.650311

parallel time2=0.440678ms
speedup2      =4.21771

parallel time3=2.38619ms
speedup3      =0.77892
```

4 实验总结

我实现了串行和并行实现，并对并行实现进行了多种优化方法的探究。

在探究中，我发现，仅考虑外层循环并行，且考虑动态调度的负载均衡时，性能最好。在并行数约为总CPU核数一半时能性能达到最优。矩阵规模上，当矩阵规模较大时，性能较好。性能最好时加速比可以达到8，平均加速比可以达到接近5。

总结来说，并行并不能无限制地增加效率，还需要考虑运算规模，运算结构等影响因素。一般来说，若运算规模较大，则并行开销能被冲淡，此时的并行效果会较好。

5 代码

5.1 预处理

将压缩矩阵从列优先转换为行优先。不改变原文件大小，最终输出的结果仅为原文件行的重新排列。

下面是一个矩阵例子：

```
%%列优先矩阵
5 4 6
1 1 1
4 2 5
1 3 2
2 3 3
4 3 6
5 4 4

%%行优先矩阵
5 4 6
1 1 1
1 3 2
2 3 3
4 2 5
4 3 6
5 4 4
```

思想即为维护一个优先队列，优先级为先行再列，如下段代码cmp所示。

代码如下：

```
#include "fstream"
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
using namespace std;

struct node
{
    int row;
    int col;
    double value;
    node(int r, int c, double v)
    {
        this->row = r;
        this->col = c;
        this->value = v;
    }
};

struct cmp
{
    bool operator()(node a, node b)
    {
        if (a.row > b.row)
            return 1;
        else if (a.row == b.row && a.col > b.col)
            return 1;
        return 0;
    }
};

void process(string filename)
{
    string file_e = filename + ".mtx";
    const char *file = file_e.c_str();
    FILE *f = fopen(file, "r");
    if (f == NULL)
    {
        fprintf(stderr, "Error opening file: %s\n", file);
```

```
        exit(1);
    }

    // 跳过第一行的说明信息
    char line[256];
    if (fgets(line, sizeof(line), f) == NULL)
    {
        fprintf(stderr, "Error reading file header\n");
        fclose(f);
        exit(1);
    }

    // 读取矩阵的基本信息
    int n_rows, n_cols, n_nonzero;
    if (fscanf(f, "%d %d %d", &n_rows, &n_cols, &n_nonzero) !=
3)
    {
        fprintf(stderr, "Error reading matrix dimensions\n");
        fclose(f);
        exit(1);
    }
    // cout << n_rows << " " << n_cols << " " << n_nonzero;

    std::priority_queue<node, std::vector<node>, cmp> q;

    for (int i = 0; i < n_nonzero; i++)
    {
        int r, c;
        double v;
        if (fscanf(f, "%d %d %le", &r, &c, &v) != 3)
        {
            fprintf(stderr, "Error reading matrix element\n");
            fclose(f);
            exit(1);
        }
        node new_node(r, c, v);
        q.push(new_node);
    }

    // 将队列元素复制到vector
    std::vector<node> vec;
    while (!q.empty())
```

```
82      {
83          vec.push_back(q.top());
84          q.pop();
85      }
86
87      cout << "start to output" << endl;
88
89      // 输出
90      string outfile = filename + "_line.mtx";
91      std::ofstream outFile(outfile);
92      if (outFile.is_open())
93      {
94          // 将输出写入文件
95          outFile << line;
96          outFile << n_rows << " " << n_cols << " " << n_nonzero
    << endl;
97          // 反向遍历vector，因为复制到vector中的元素顺序是逆序的
98          for (auto it = vec.begin(); it != vec.end(); ++it)
99          {
100             outFile << std::fixed << it->row << " " << it->col
    << " ";
101             outFile << std::scientific << std::setprecision(13)
    << it->value << std::endl;
102         }
103         outFile << endl;
104         // 关闭文件
105         outFile.close();
106         std::cout << "Output written to output." << std::endl;
107     }
108     else
109     {
110         std::cerr << "Unable to open output file." <<
    std::endl;
111         return;
112     }
113     fclose(f);
114     return;
115 }
116
117 int main(int argc, char *argv[])
118 {
119     // 读取参数，确定处理目标
```

```
    string target;
    if (argc < 2)
    {
        std::cout << "Usage: " << argv[0] << " <matrix name>\n";
        return 1;
    }
    try
    {
        target = argv[1];
        std::cout << "process target : " << target << std::endl;
    }
    catch (const std::invalid_argument &e)
    {
        std::cerr << "Error: " << e.what() << " - Invalid argument: " << argv[1] << std::endl;
        return 1;
    }
    catch (const std::out_of_range &e)
    {
        std::cerr << "Error: " << e.what() << " - Argument " << argv[1] << " is out of range" << std::endl;
        return 1;
    }

    string filename = target;
    process(filename);
    return 0;
}

```

5.2 算法

```
#include "algorithm"
#include "fstream"
#include "iostream"
#include "omp.h"
#include <chrono>
#include <stdio.h>
#include <stdlib.h>

```

```cpp
#include <string>
#include <time.h>
#include <vector>

using namespace std;

int num_thread = 2;

typedef struct
{
    int n_rows;    // 矩阵的行数
    int n_cols;    // 矩阵的列数
    int n_nonzero; // 非零元素的个数
    int *row_ptr;  // 行指针数组
    int *col_ind;  // 列索引数组
    double *val;   // 非零元素值数组
} crs_matrix_t;

// 读入稀疏矩阵
crs_matrix_t read_mtx_file(const char *filename)
{
    FILE *f = fopen(filename, "r");
    if (f == NULL)
    {
        fprintf(stderr, "Error opening file: %s\n", filename);
        exit(1);
    }

    // 跳过第一行的说明信息
    char line[256];
    if (fgets(line, sizeof(line), f) == NULL)
    {
        fprintf(stderr, "Error reading file header\n");
        fclose(f);
        exit(1);
    }

    // 读取矩阵的基本信息
    int n_rows, n_cols, n_nonzero;
    if (fscanf(f, "%d %d %d", &n_rows, &n_cols, &n_nonzero) !=
3)
    {
```

```
        fprintf(stderr, "Error reading matrix dimensions\n");
        fclose(f);
        exit(1);
    }
    // cout << n_rows << " " << n_cols << " " << n_nonzero;

    // 分配内存
    crs_matrix_t A;
    A.n_rows = n_rows;
    A.n_cols = n_cols;
    A.n_nonzero = n_nonzero;
    A.row_ptr = (int *)malloc((n_rows + 1) * sizeof(int));
    A.col_ind = (int *)malloc(n_nonzero * sizeof(int));
    A.val = (double *)malloc(n_nonzero * sizeof(double));

    // 读取非零元素的行索引、列索引和值
    int row = 0, idx = 0;
    A.row_ptr[0] = 0;
    for (int i = 0; i < n_nonzero; i++)
    {
        int r, c;
        double v;
        if (fscanf(f, "%d %d %le", &r, &c, &v) != 3)
        {
            fprintf(stderr, "Error reading matrix element\n");
            fclose(f);
            free(A.row_ptr);
            free(A.col_ind);
            free(A.val);
            exit(1);
        }
        A.col_ind[i] = c - 1; // 存储列索引（减1）
        A.val[i] = v;         // 存储值
        while (r > row + 1)
        {
            A.row_ptr[row + 1] = i;
            row++;
        }
        row = r - 1;
    }
    while (row < n_rows)
    {
```

```
            A.row_ptr[row + 1] = n_nonzero;
            row++;
        }

        fclose(f);
        return A;
}

// 输出稀疏矩阵参数
void print_matrix_par(const crs_matrix_t *A)
{
    // data
    cout << " data:      ";
    for (int i = 0; i < A->n_nonzero; i++)
    {
        cout << A->val[i] << " ";
    }
    cout << endl;
    // col
    cout << " col index: ";
    for (int i = 0; i < A->n_nonzero; i++)
    {
        cout << A->col_ind[i] << " ";
    }
    cout << endl;
    // row
    cout << " row ptr:   ";
    for (int i = 0; i < A->n_rows + 1; i++)
    {
        cout << A->row_ptr[i] << " ";
    }
    cout << endl;
}

// 打印部分矩阵
void print_matrix_head(const crs_matrix_t *A, int rows, int
cols)
{
    printf("Printing the first %d x %d elements of the
matrix:\n", rows, cols);
    // 打开一个新的输出文件
    std::ofstream outFile("output.txt");
```

```
131
132     // 检查文件是否打开成功
133     if (outFile.is_open())
134     {
135         // 将输出写入文件
136         for (int i = 0; i < rows; i++)
137         {
138             double *temp_row = new double[A->n_cols]; // 临时数
    组，逐行输出
139             for (int j = 0; j < cols; j++)
140                 temp_row[j] = 0;
141             for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1];
    j++) // j为该数在data中的位置
142             {
143                 if (A->col_ind[j] < cols) // A->col_ind[j]为该数
    的列号，i为该数的行号
144                 {
145                     temp_row[A->col_ind[j]] = A->val[j];
146                 }
147             }
148             for (int j = 0; j < cols; j++)
149                 printf("%20.8f", temp_row[j]);
150             printf("\n");
151         }
152
153         // 关闭文件
154         outFile.close();
155
156         std::cout << "Output written to 'output.txt'." <<
    std::endl;
157     }
158     else
159     {
160         std::cerr << "Unable to open output file." <<
    std::endl;
161         return;
162     }
163
164     // for (int i = 0; i < rows; i++)
165     // {
166     //     double *temp_row = new double[A->n_cols]; // 临时数
    组，逐行输出
```

```
167     //     for (int j = 0; j < cols; j++)
168     //         temp_row[j] = 0;
169     //     for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1];
    j++) // j为该数在data中的位置
170     //     {
171     //         if (A->col_ind[j] < cols) // A->col_ind[j]为该数
    的列号，i为该数的行号
172     //         {
173     //             temp_row[A->col_ind[j]] = A->val[j];
174     //         }
175     //     }
176     //     for (int j = 0; j < cols; j++)
177     //         printf("%20.8f", temp_row[j]);
178     //     printf("\n");
179     // }
180 }
181
182 // 生成随机向量
183 double *generate_random_vector(int n, double min, double max)
184 {
185     double *vec = (double *)malloc(n * sizeof(double));
186     // 初始化随机数种子
187     srand(time(NULL));
188     // cout << "random vec: ";
189     for (int i = 0; i < n; i++)
190     {
191         vec[i] = min + (double)rand() / RAND_MAX * (max - min);
192         // cout << vec[i] << " ";
193     }
194     // cout << endl;
195
196     return vec;
197 }
198
199 // 矩阵向量乘法(串行)
200 void crs_matrix_vector_product_serial(const crs_matrix_t *A,
    double *x, double *y)
201 {
202     for (int i = 0; i < A->n_rows; i++)
203     {
204         double sum = 0.0;
205         for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
```

```
206         {
207             sum += A->val[j] * x[A->col_ind[j]];
208         }
209         y[i] = sum;
210     }
211 }
212 
213 // 矩阵向量乘法(并行化外层循环)
214 void crs_matrix_vector_product_parallel(const crs_matrix_t *A,
    double *x, double *y)
215 {
216     omp_set_num_threads(num_thread);
217 #pragma omp parallel for schedule(dynamic)
218     for (int i = 0; i < A->n_rows; i++)
219     {
220         double sum = 0.0;
221         for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
222         {
223             sum += A->val[j] * x[A->col_ind[j]];
224         }
225         y[i] = sum;
226         // cout << "i= " << i << " " << omp_get_thread_num() <<
    "    ";
227         // if (i % 10 == 0) cout << endl;
228     }
229 }
230 
231 // 矩阵向量乘法(并行化外层循环+动态调度)
232 void crs_matrix_vector_product_parallel2(const crs_matrix_t *A,
    double *x, double *y)
233 {
234     omp_set_num_threads(num_thread);
235 #pragma omp parallel for schedule(dynamic)
236     for (int i = 0; i < A->n_rows; i++)
237     {
238         double sum = 0.0;
239         for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
240         {
241             sum += A->val[j] * x[A->col_ind[j]];
242         }
243         y[i] = sum;
```

```
244         // cout << "i= " << i << " " << omp_get_thread_num() <<
    "    ";
245         // if (i % 10 == 0) cout << endl;
246     }
247 }
248
249 // 矩阵向量乘法(并行化外层循环+动态调度+内层并行)
250 void crs_matrix_vector_product_parallel3(const crs_matrix_t *A,
    double *x, double *y)
251 {
252     omp_set_num_threads(num_thread);
253 #pragma omp parallel for schedule(dynamic)
254     for (int i = 0; i < A->n_rows; i++)
255     {
256         double sum = 0.0;
257 #pragma omp parallel for schedule(dynamic) reduction(+ : sum)
258         for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++)
259         {
260             sum += A->val[j] * x[A->col_ind[j]];
261         }
262         y[i] = sum;
263     }
264 }
265
266 // 矩阵向量乘法（私有临时x向量）
267 void crs_matrix_vector_product_parallel4(const crs_matrix_t *A,
    double *x, double *y)
268 {
269     omp_set_num_threads(num_thread);
270 #pragma omp parallel
271     {
272         // 声明线程私有的临时 x 向量
273         double private_x[A->n_cols];
274         for (int i = 0; i < A->n_cols; i++)
275         {
276             // 将 x 向量拷贝到线程私有的缓存中
277             private_x[i] = x[i];
278         }
279
280 #pragma omp for
281         for (int i = 0; i < A->n_rows; i++)
282         {
```

```
            double sum = 0.0;
            for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1];
j++)
            {
                // 使用线程私有的 private_x 向量进行计算
                sum += A->val[j] * private_x[A->col_ind[j]];
            }
            y[i] = sum;
        }
    }
}

// 比较函数（比较串并行结果是否一致）
void compare(double *x, double *y, int size, string name)
{
    bool flag = true;
    int wrong_num = 0;
    for (int i = 0; i < size; i++)
    {
        if (x[i] != y[i])
        {
            flag = false;
            wrong_num++;
        }
    }
    cout << name << "  accuracy: " << (double)(size -
wrong_num) / (double)size << "      ";
    if (flag == true)
        cout << "(outcome accurate)" << endl;
    else
        cout << "(outcome wrong)" << endl;
}

int main(int argc, char *argv[])
{
    // 读取参数，确定线程数
    string matrix_name;
    if (argc < 3)
    {
        std::cout << "Usage: " << argv[0] << " <thread_num>
<matrix_name>\n";
        return 1;
```

```
322     }
323     try
324     {
325         num_thread = std::stoi(argv[1]);
326         std::cout << "set thread number : " << num_thread <<
    std::endl;
327         matrix_name = argv[2];
328         std::cout << "set matrix name   : " << matrix_name <<
    std::endl;
329     }
330     catch (const std::invalid_argument &e)
331     {
332         std::cerr << "Error: " << e.what() << " - Invalid
    argument: " << argv[1] << std::endl;
333         return 1;
334     }
335     catch (const std::out_of_range &e)
336     {
337         std::cerr << "Error: " << e.what() << " - Argument " <<
    argv[1] << " is out of range" << std::endl;
338         return 1;
339     }
340
341     // 检测环境
342     int concurrency = omp_get_num_procs();
343     cout << "system max concurrency = " << concurrency << endl;
344     //
345     const char *filename = "psmigr_3_line.mtx"; // default
346     string file_e = matrix_name + "_line.mtx";
347     filename = file_e.c_str();
348     crs_matrix_t A = read_mtx_file(filename);
349     // print_matrix_par(&A);
350     // print_matrix_head(&A, 5, 4);
351
352     double *v = generate_random_vector(A.n_cols, 1, 1);
353     double *ans_serial = new double[A.n_cols];
354     double *ans_parallel = new double[A.n_cols];
355     double *ans_parallel2 = new double[A.n_cols];
356     double *ans_parallel3 = new double[A.n_cols];
357
358     // 串行计算
```

```
359     auto start_serial =
    std::chrono::high_resolution_clock::now();
360     crs_matrix_vector_product_serial(&A, v, ans_serial);
361     auto end_serial =
    std::chrono::high_resolution_clock::now();
362     // cout << "ans = " << endl;
363     // for (int i = 0; i < A.n_rows; i++)
364     //     cout << ans_serial[i] << " ";
365     // cout << endl;
366
367     // 并行计算(优化外层循环)
368     auto start_parallel =
    std::chrono::high_resolution_clock::now();
369     crs_matrix_vector_product_parallel(&A, v, ans_parallel);
370     auto end_parallel =
    std::chrono::high_resolution_clock::now();
371
372     // 并行计算（优化外层循环+动态调度）
373     auto start_parallel2 =
    std::chrono::high_resolution_clock::now();
374     crs_matrix_vector_product_parallel2(&A, v, ans_parallel2);
375     auto end_parallel2 =
    std::chrono::high_resolution_clock::now();
376
377     // 并行计算（优化外层循环+动态调度+内层并行）
378     auto start_parallel3 =
    std::chrono::high_resolution_clock::now();
379     crs_matrix_vector_product_parallel3(&A, v, ans_parallel3);
380     auto end_parallel3 =
    std::chrono::high_resolution_clock::now();
381
382     // 比较准确性
383     compare(ans_serial, ans_parallel, A.n_cols, "parallel-
    basic");
384     compare(ans_serial, ans_parallel2, A.n_cols, "parallel-2");
385     compare(ans_serial, ans_parallel3, A.n_cols, "parallel-3");
386
387     // 比较性能
388     std::chrono::duration<double, std::milli> duration_serial =
    end_serial - start_serial;
389     std::chrono::duration<double, std::milli> duration_parallel
    = end_parallel - start_parallel;
```

```
390     std::chrono::duration<double, std::milli>
    duration_parallel2 = end_parallel2 - start_parallel2;
391     std::chrono::duration<double, std::milli>
    duration_parallel3 = end_parallel3 - start_parallel3;
392     cout << "serial time   =" << duration_serial.count() <<
    "ms" << endl
393          << endl;
394     cout << "parallel time =" << duration_parallel.count() <<
    "ms" << endl;
395     cout << "speedup1      =" << duration_serial.count() /
    duration_parallel.count() << endl
396          << endl;
397     cout << "parallel time2=" << duration_parallel2.count() <<
    "ms" << endl;
398     cout << "speedup2      =" << duration_serial.count() /
    duration_parallel2.count() << endl
399          << endl;
400     cout << "parallel time3=" << duration_parallel3.count() <<
    "ms" << endl;
401     cout << "speedup3      =" << duration_serial.count() /
    duration_parallel3.count() << endl;
402
403     //  释放内存
404     free(A.row_ptr);
405     free(A.col_ind);
406     free(A.val);
407     return 0;
408 }
409
```

5.3 测试与绘图

```
1  import subprocess
2  import time
3  import matplotlib.pyplot as plt
4  import itertools
5
6  # 每个矩阵测试轮数
7  test_num = 20
8  # 测试矩阵名
9  matrix_name = "psmigr_3"
```

```
10
11  # 定义需要测试的线程数
12  thread_num1 = range(1,32)                  # 范围
13  thread_num2 = [64,96,112,128,192,256]    # 离散点
14  thread_numbers = sorted(list(itertools.chain(thread_num1,
    thread_num2)))
15  print(thread_numbers)
16
17  # 记录每个线程数的平均speedup
18  speedups = []
19  speedups2 = []
20  speedups3 = []
21
22  for thread_num in thread_numbers:
23      total_speedup = 0
24      total_speedup2 = 0
25      total_speedup3 = 0
26      for _ in range(test_num):
27          # 执行可执行文件并记录时间
28          result = subprocess.run(['./mv', str(thread_num),
    matrix_name], stdout=subprocess.PIPE, stderr=subprocess.PIPE,
    universal_newlines=True)
29
30          # 解析输出结果
31          lines = result.stdout.strip().split('\n')
32          speedup = 0
33          speedup2 = 0
34          speedup3 = 0
35          for line in lines:
36              if line.startswith('speedup1'):
37                  speedup = float(line.split('=')[1][:-1])
38              if line.startswith('speedup2'):
39                  speedup2 = float(line.split('=')[1][:-1])
40              if line.startswith('speedup3'):
41                  speedup3 = float(line.split('=')[1][:-1])
42          total_speedup += speedup
43          total_speedup2 += speedup2
44          total_speedup3 += speedup3
45      print("thread_num {} complete".format(thread_num))
46
47      # 计算平均speedup
48      avg_speedup = total_speedup / test_num
```

```
    speedups.append(avg_speedup)
    avg_speedup2 = total_speedup2 / test_num
    speedups2.append(avg_speedup2)
    avg_speedup3 = total_speedup3 / test_num
    speedups3.append(avg_speedup3)


print(speedups)
print(speedups2)
print(speedups3)
# 绘制参数-speedup图
plt.figure(figsize=(10, 6))
plt.plot(thread_numbers, speedups, label="basic", color="blue")
plt.plot(thread_numbers, speedups2, label="schedule",
color="red")
plt.plot(thread_numbers, speedups3, label="inner",
color="green")
plt.xlabel('Thread Number')
plt.ylabel('Average Speedup')
plt.legend()
plt.title('Thread Number vs. Average Speedup')
plt.grid()
plt.savefig('thread_speedup.png')

```

5.4 Linux操作命令

按照如下流程可以复现该实验，并得到相似的结果。

```
# （1）从矩阵市场获取矩阵
wget https://math.nist.gov/pub/MatrixMarket2/misc/qcd/conf5.0-
00l4x4-1400.mtx.gz
# （2）解压矩阵
gzip -d psmigr_1.mtx.gz
# （3）编译执行order，并输入matrix_name，不要包括.mtx。将生成
<matrix_name>_line.mtx
# e.g. ./order psmigr_3
g++ order.cpp -o order
./order <matrix_name>
# （4）编译执行mv，并输入并行数与matrix_name，不要包括.mtx，将输出结果。
# e.g. ./mv 64 psmigr_3

```

```
g++ -fopenmp mv.cpp -o mv
./mv <thread_num> <matrix_name>
# （5）【可选】修改python代码，选择想要绘图的并行数与矩阵名
# （6）运行python，绘制图像
python3 test.py

```