

中山大学计算机学院本科生实验报告

(2024 学年第 2 学期)

课程名称：面向邻域的并行数值方法

批改人：

实验	解线性方程组	专业（方向）	信息与计算科学
学号	22336049	姓名	陳日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2025 年 7 月 9 日

1. 实验内容

本次作业的目标是读取已提供的 `Matrix.dat` 文件，并求解对应的线性方程组 $Ax = b$ 。作为热传导问题的基础版本，并不要求自行构建矩阵，而是直接使用预先生成的稀疏矩阵数据进行求解，从而帮助掌握稀疏矩阵的存储格式、线性系统的求解流程。

热扩散问题的一般形式为 $-\nabla \cdot (k \nabla T) = q$ ，通过有限差分法或有限体积法进行空间离散后，可转化为线性方程组 $Ax = b$ ，其中 A 为稀疏矩阵， x 表示温度变量， b 为源项向量。本算例来自一个纯热扩散问题，其中右端为加热边界，施加热流密度为 1MW ，左端为恒温边界，温度为 300K ，热量由右向左传导。如下图所示：



在具体操作中，首先需要正确读取 `Matrix.dat` 文件，该文件以 CSR（Compressed Sparse Row）格式存储稀疏矩阵 A 的非零元素、列索引与行指针，并附带右端项向量 b ，在该算例中 b 全部为零。接着，将这些数据还原为线性方程组 $Ax = b$ ，并使用合适的数值方法进行求解，获得解向量 x 。

求解完成后，需要将 x 输出至 `Result.dat` 文件中，每行一个温度值，总共应为 n 行，以保证与网格文件中节点数量一致，避免可视化阶段出现误差、全红图像或程序报错等问题。为了判断本次求解是否数值上正确，应计算残差范数 $\|Ax - b\|$ ，若该值不超过 1×10^{-6} ，则认为解是可靠的。

在此基础版本完成后，还可进一步尝试拓展任务，例如将问题更紧密地联系到二维热传导背景，自行构造边界条件（如左边恒温 300K ，右边恒温 1000K ），手动生成热传导模型所需的矩阵 A 与向量 b ，并对结果进行可视化。此外，还可绘制稀疏矩阵结构图以帮助理解其存储特性，或比较不同求解器的性能差异。

本项目已附带提供 `grid.lb8.ugrid` 网格文件，可与 `Result.dat` 联合用于温度场可视化，支持 Tecplot、ParaView、Gmsh 等常见工具。在报告中需说明使用的可视化流程，并确保输出格式与网格节点数量严格匹配。

2. 算法原理

本次作业的核心问题是求解二维稳定热传导问题，涉及到偏微分方程的数值解法。通过有限差分法（Finite Difference Method, FDM）将连续的热传导方程离散化，并通过共轭梯度法（Conjugate Gradient, CG）求解由此得到的线性方程组。

(1). 有限差分法

有限差分法是一种经典且广泛应用的数值方法，主要用于求解各类微分方程，特别是在工程与物理问题中具有重要地位。该方法的核心思想是通过连续定义域进行离散化，将原本包含导数的微分方程转化为代数方程组。具体而言，它用网格点上的函数值替代连续函数，通过相邻点的函数值之差（即差商）近似导数，从而逼近微分运算。

在本实验中，二维稳态热传导问题作为求解对象，其偏微分方程在热传导系数 k 为常数的条件下可简化为拉普拉斯方程： $-\frac{\partial^2 T}{\partial x^2} - \frac{\partial^2 T}{\partial y^2} = q$ 。有限差分法是将空间域上的偏微分方程转化为差分方程的数值方法。在二维网格上，采用了五点差分格式进行离散化，即在网格上每个节点的温度值由其上下左右四个邻节点的温度值来表示。这种方法的核心是通过差分逼近导数。

对于每个节点 $T_{i,j}$ ，五点差分公式为：
$$\frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1} - 4T_{i,j}}{\Delta x^2} = 0$$
，其中 Δx 和 Δy 是网格间距。在离散过程中，内部节点满足上述差分关系，而边界节点则结合具体的边界条件处理：例如在本实验中，左边界和右边界施加了 Dirichlet 边界条件（分别为 300K 和 1000K 的恒温），上下边界则采用 Neumann 边界条件，表示绝热边界，即法向导数为零。

有限差分法的理论基础来自泰勒级数展开。通过泰勒展开可以系统推导各种差分格式的精度和截断误差，如中心差分法逼近二阶导数具有 $O(\Delta x^2)$ 的精度。在二维问题中，五点差分格式在保证计算效率的同时，也具备良好的数值稳定性与收敛性。

最终，离散化后的整个热传导问题被转化为一个稀疏对称正定的线性代数方程组 $A \cdot x = b$ 。其中矩阵 A 反映了各个网格点之间的耦合关系，向量 x 表示需要求解的温度场，向量 b 则综合考虑了源项与边界条件的影响。由于方程组维度大且稀疏，适合使用共轭梯度法等高效迭代算法求解，从而获得数值解。

(2). 共轭梯度法

共轭梯度法（Conjugate Gradient Method）是基于最速下降法的思想，但在每次迭代中选择与之前方向共轭的新搜索方向，从而提高收敛效率，避免在每一步都重复搜索已优化过的方向。该算法的核心思想是，在 A 所定义的内积空间中构造一组 A 共轭向量作为搜索基底，使得每次迭代不仅减少残差，而且保持与之前方向的独立性，从而在有限维空间中实现快速收敛。

在初始化阶段，共轭梯度法从一个初始猜测解 x_0 出发，计算初始残差 $r_0 = b - Ax_0$ ，并令搜索方向 $p_0 = r_0$ 。接下来的每一步迭代，都首先在当前方向上找到一个最优步长 α_k ，使得新的解 $x_{k+1} = x_k + \alpha_k p_k$ 能够在该方向上最小化误差。然后更新残差 r_{k+1} ，再基于新残差与旧残差的比值构造一个方向修正因子 β_k ，以此得到新的搜索方向 $p_{k+1} = r_{k+1} + \beta_k p_k$ 。由于每一步的方向都是 A- 共轭的，这种迭代在理论上至多 n 步（ n 为维度）内就能精确到达解。

共轭梯度法不仅具有良好的收敛性，而且能够避免矩阵求逆或存储 A 的全部元素，在处理大规模稀疏系统时具有极高的效率。每次迭代都只需要进行稀疏矩阵-向量乘法和向量加法等基本操作，避免了直接求解矩阵的过程，特别是在本实验中的热传导问题所形成的对称正定稀疏矩阵场景下，共轭梯度法是一种既高效又稳定的求解方式。

3. 文件说明

	档案名称	功能描述
基础部分	main.cpp	读取 Matrix.dat 文件中给出的稀疏矩阵，采用并行共轭梯度法求解线性方程组 $Ax = b$ ，并输出结果向量至 Result.dat 文件，并输出残差和运行时间等信息。
拓展部分	heat.cpp	针对规则 1000×1000 网格的二维稳态热传导问题，生成稀疏矩阵（CSR 格式）并保存至 Matrix_n.dat。热传导模型设定左边界为 300K，右边界为 1000K，上下边界为 Neumann 条件。
	main_n.cpp	读取 heat.cpp 生成的 Matrix_n.dat 并调用并行共轭梯度法进行求解，结果输出至 Result_n.dat，并支持打印温度最值、残差、相对残差等指标。
	visualize.py	融合了温度场与稀疏结构图的可视化功能，分别读取 Result_n.dat 和 Matrix_n.dat，输出 heatmap.png（温度分布）与 MatrixA_Structure.png（稀疏矩阵结构图）。

4. 关键代码解析

(1). 基础部分 main.cpp

(i). readMatrix()

```
void readMatrix(const string& filename) {
    ifstream fin(filename);
    if (!fin) {
        cerr << "Error: Cannot open file " << filename << endl;
        exit(1);
    }

    fin >> n >> nnz;
    nA.resize(nnz);
    JA.resize(nnz);
    IA.resize(n + 1);
    b.resize(n, 0.0); // 全部为0
    x.resize(n, 0.0); // 初始解为0

    for (int i = 0; i < nnz; ++i)
        fin >> nA[i];
    for (int i = 0; i < nnz; ++i)
        fin >> JA[i];
    for (int i = 0; i <= n; ++i)
        fin >> IA[i];
    for (int i = 0; i < n; ++i)
        fin >> b[i];

    fin.close();
}
```

从 Matrix.dat 文件中读取稀疏矩阵 A 以及右端向量 b 。矩阵使用 CSR 格式存储，即以 nA、JA 和 IA 三组向量表示，初始化解向量 x 和源项向量 b ，并设置为零。

(ii). sparseMatVec()

```
void sparseMatVec(const vector<double>& vec, vector<double>& result) {
    #pragma omp parallel for
    for (int i=0; i<n; ++i) {
        double sum = 0.0;
        for (int j=IA[i]; j<IA[i+1]; ++j) {
            sum += nA[j]*vec[JA[j]];
        }
        result[i] = sum;
    }
}
```

实现稀疏矩阵 A 与任意向量 vec 的乘法，结果存储在 $result$ 向量中。由于 A 是以 CSR 形式存储，因此该函数遍历每一行，并对每行中的非零元素执行乘积累加操作。通过 OpenMP 进行并行加速。

(iii). dotProduct()

```
double dotProduct(const vector<double>& a, const vector<double>& b) {
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i=0; i<a.size(); ++i)
        sum += a[i]*b[i];
    return sum;
}
```

函数计算两个向量之间的内积，并通过 OpenMP 的 reduction 操作实现并行累加。函数在共轭梯度法中用于计算残差范数、方向向量的归一化因子等关键量。

(iv). conjugateGradient()

```
void conjugateGradient(int maxIter=10000, double tol=1e-6) {
    vector<double> r(n), p(n), Ap(n);
    vector<double> Ax(n);
    sparseMatVec(x, Ax);
    #pragma omp parallel for
    for (int i=0; i<n; ++i)
        r[i] = b[i]-Ax[i];
    p = r;

    double rsold = dotProduct(r, r);
    for (int k=0; k<maxIter; ++k) {
        sparseMatVec(p, Ap);
        double alpha = rsold / dotProduct(p, Ap);

        #pragma omp parallel for
        for (int i=0; i<n; ++i)
            x[i] += alpha*p[i];

        #pragma omp parallel for
        for (int i=0; i<n; ++i)
            r[i] -= alpha*Ap[i];

        double rsnew = dotProduct(r, r);
        if (sqrt(rsnew) < tol)
            break;

        #pragma omp parallel for
        for (int i=0; i<n; ++i)
            p[i] = r[i] + (rsnew/rsold) * p[i];

        rsold = rsnew;
    }
}
```

函数是求解稀疏线性方程组的核心算法实现——共轭梯度法。通过构造共轭方向，在每一步迭代中不断逼近精确解。初始时设定残差为 $r_0 = b - Ax_0$ ，并以 r_0 为初始搜索方向。每次迭代包括方向向量与矩阵乘法、步长因子更新、解向量与残差更新、新方向计算等步骤，直到残差满足设定的容差阈值或达到最大迭代次数。共轭梯度法优点在于无需显式存储矩阵 A ，仅需使用其稀疏结构进行乘法计算，同时能够在数百至数千步内快速收敛至精度范围内的数值解

(v). computeResidual()

```
double computeResidual() {
    vector<double> Ax(n);
    sparseMatVec(x, Ax);
    vector<double> r(n);
    #pragma omp parallel for
    for (int i=0; i<n; ++i)
        r[i] = Ax[i]-b[i];
    return vectorNorm(r);
}
```

函数用于计算数值解 x 的残差 $\|Ax - b\|$ ，作为验证求解准确性的标准。通过调 `sparseMatVec` 得到 Ax ，然后与 b 相减，并求出差向量的欧几里得范数。其计算结果用于评估当前解的数值精度。该数值越小，说明解越接近真实解。

(2). 拓展部分 heat.cpp

(i). 构造二维规则网格结构与初始变量

```
const int Nx = 1000, Ny = 1000;
const int N = Nx * Ny;
const double T_left = 300.0;
const double T_right = 1000.0;

std::vector<double> nA;
std::vector<int> JA;
std::vector<int> IA(N + 1, 0);
std::vector<double> b(N, 0.0);

nA.reserve(5ull * N); JA.reserve(5ull * N);
```

文件定义了一个 1000×1000 的规则二维网格，并声明用于构建稀疏矩阵的核心向量 nA 、 JA 、 IA 和右端项向量 b ，对应于稀疏矩阵的压缩行存储（CSR）格式。同时提前 `reserve` 这些向量的容量，提升性能。

(ii). 应用边界条件与内部结构构建稀疏矩阵

```
if (left) {
    nA.push_back(1.0); JA.push_back(k);
    b[k] = T_left;
    continue;
}

if (right) {
    nA.push_back(1.0); JA.push_back(k);
    b[k] = T_right;
    continue;
}
```

通过两层嵌套循环遍历每一个节点 k ，根据其所处位置判断是否为左/右边界或内部点，并据此构建不同的方程。核心是区分 Dirichlet 条件（左、右）与 Neumann 条件（上、下）。

```
// 内部或 Neumann 边界点
nA.push_back(4.0); JA.push_back(k); // 对角项先设为 4.0

// 左邻
if (i>1) {
    nA.push_back(-1.0); JA.push_back(k - 1);
}
else {
    b[k] += T_left;
}
```

```
if (isNeumannY) {
    nA[IA[k]] = 3.0;
}
```

对于内部点或 Neumann 边界点，统一使用五点差分模板构建稀疏结构。若上下方向为 Neumann，则不添加该方向的项，同时将对角项从 4.0 修改为 3.0 以平衡稀疏矩阵的行和。

(iii). 生成 CSR 格式的输出文件 Matrix_n.dat

```
fout << N << ' ' << nA.size() << '\n';
for (double v : nA) fout << v << ' ';
fout << '\n';
for (int c : JA) fout << c << ' ';
fout << '\n';
for (int r : IA) fout << r << ' ';
fout << '\n';
for (double v : b) fout << v << '\n';
```

包括：矩阵规模 n 、非零元数量 nnz 、稀疏矩阵数据 nA 、 JA 、 IA 以及右端向量 b 。

(3). 拓展部分 main_n.cpp

(i). double conjugateGradient()

```
double conjugateGradient(int maxIter = 5000, double tol = 1e-6) {
    vector<double> r(n), p(n), Ap(n);
    sparseMatVec(x, Ap);
    #pragma omp parallel for
    for (int i=0; i<n; ++i)
        r[i]=b[i];
    p=r;

    double rsold = dotProduct(r, r);
    const double bNorm = sqrt(rsold);

    for (int k=0; k<maxIter; ++k) {
        sparseMatVec(p, Ap);
        double alpha = rsold/dotProduct(p, Ap);

        #pragma omp parallel for
        for (int i=0; i<n; ++i) {
            x[i] += alpha*p[i];
            r[i] -= alpha*Ap[i];
        }

        double rsnew = dotProduct(r, r);
        double rel_res = sqrt(rsnew) / bNorm;

        if (rel_res < tol) {
            cout << "[CG] converged: " << k + 1 << " iterations, rel-res = " << rel_res << '\n';
            return rel_res;
        }

        #pragma omp parallel for
        for (int i=0; i<n; ++i)
            p[i] = r[i] + (rsnew / rsold) * p[i];
        rsold = rsnew;
    }

    double final_rel = sqrt(rsold) / bNorm;
    cout << "[CG] reached maxIter = " << maxIter << ", rel-res = " << final_rel << '\n';
    return final_rel;
}
```

函数实现了并行化的共轭梯度法，并通过相对残差来判断收敛性。与基础版本 main_n.cpp 不同，double conjugateGradient() 返回的是相对残差而非绝对残差，这一标准在右端项 b 不为零的情况下更具数值意义，可以避免因量纲较大导致误差判断失真。此外，初始残差设定为：for (int i=0; i<n; ++i) $r[i] = b[i]$ ；，因为初始 $x = 0$ ，所以 $r = b - Ax$ ，简化了初始化阶段的开销。

(ii). void printMinMaxTemperature()

```
void printMinMaxTemperature() {
    double minT = x[0], maxT = x[0];
#ifdef _OPENMP
    #pragma omp parallel for reduction(min:minT) reduction(max:maxT)
    for (int i=0; i<n; ++i) {
        minT = min(minT, x[i]);
        maxT = max(maxT, x[i]);
    }
#else
    #pragma omp parallel
    {
        double tmin = minT, tmax = maxT;
        #pragma omp for nowait
        for (int i=0; i<n; ++i) {
            tmin = min(tmin, x[i]);
            tmax = max(tmax, x[i]);
        }
        #pragma omp critical
        {
            minT = min(minT, tmin);
            maxT = max(maxT, tmax);
        }
    }
#endif
    cout << "Temperature range = [" << minT << ", " << maxT << "]\n";
}
```

函数用于输出温度解向量 x 中的最小值与最大值，帮助理解热传导分布是否合理，使用 OpenMP 并行归约操作实现高效的全局最值查找。

(4). 拓展部分 visualize.py

(i). plot_sparsity()

```
def plot_sparsity(matrix_file="Matrix_n.dat"):
    try:
        with open(matrix_file) as f:
            n, nnz = map(int, f.readline().split())
            nA = list(map(float, f.readline().split()))
            JA = list(map(int, f.readline().split()))
            IA = list(map(int, f.readline().split()))
    except Exception as e:
        print(f"[x] Failed to read matrix: {e}")
        return

    row, col = [], []
    for i in range(n):
        for j in range(IA[i], IA[i + 1]):
            row.append(i)
            col.append(JA[j])

    plt.figure(figsize=(7, 7))
    plt.scatter(col, row, s=0.05, marker='.', color='black')
    plt.title("Sparsity Pattern of Matrix A")
    plt.xlabel("Column Index")
    plt.ylabel("Row Index")
    plt.gca().invert_yaxis()
    plt.axis('equal')
    plt.tight_layout()
    plt.savefig("sparsity_pattern.png", dpi=300)
    plt.close()
    print("Sparsity pattern saved as: MatrixA_Structure.png")
```

函数用于绘制稀疏矩阵的结构图。读取 Matrix_n.dat 文件中的 CSR 数据结构 (IA, JA)，将每个非零元素的位置作为一个黑点绘制在坐标系中，横轴为列索引，纵轴为行索引。使用 scatter 绘图的方式，能清晰展示矩阵的带状结构。图像最终输出为 sparsity_pattern.png。

(ii). plot_temperature()

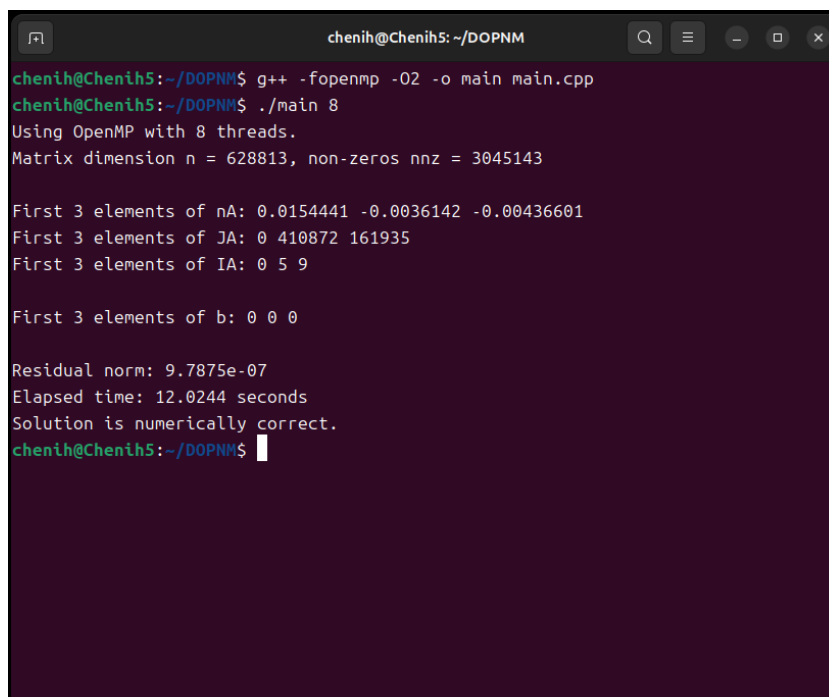
```
def plot_temperature(data_file="Result_n.dat", Nx=1000, Ny=1000):
    try:
        temp = np.loadtxt(data_file)
        if temp.size != Nx * Ny:
            raise ValueError(f"Data size mismatch: {temp.size} != {Nx}*{Ny}")
        temp_2d = temp.reshape((Ny, Nx))

        plt.figure(figsize=(8, 6))
        im = plt.imshow(temp_2d, origin='lower', cmap='hot', aspect='auto')
        plt.colorbar(im, label='Temperature (K)')
        plt.title('2D Temperature Distribution')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.tight_layout()
        plt.savefig("heatmap.png", dpi=300)
        plt.close()
        print("Heatmap saved as: heatmap.png")
    except Exception as e:
        print(f"[x] Failed to plot temperature: {e}")
```

函数负责将数值解 Result_n.dat 中的温度向量可视化为二维热力分布图。首先将长度为 $1000 \times 1000 = 10^6$ 的一维向量 reshape 成二维温度场，并使用 imshow 生成热图。颜色映射采用 hot 色系，图像直观展示了从左端 300K 到右端 1000K 的梯度变化，辅助验证数值解的物理合理性。图像最终以 heatmap.png 存储。

5. 实验结果

(1). 解线性方程组（基础部分）



```
chenih@Chenih5: ~/DOPNM
chenih@Chenih5:~/DOPNM$ g++ -fopenmp -O2 -o main main.cpp
chenih@Chenih5:~/DOPNM$ ./main 8
Using OpenMP with 8 threads.
Matrix dimension n = 628813, non-zeros nnz = 3045143

First 3 elements of nA: 0.0154441 -0.0036142 -0.00436601
First 3 elements of JA: 0 410872 161935
First 3 elements of IA: 0 5 9

First 3 elements of b: 0 0 0

Residual norm: 9.7875e-07
Elapsed time: 12.0244 seconds
Solution is numerically correct.
chenih@Chenih5:~/DOPNM$
```

程序首先成功读取了 Matrix.dat 文件中以 CSR 格式存储的稀疏矩阵与右端向量。由于右端项向量为全零，因此解的物理含义可以视作稳态下无内源项热扩散系统的温度分布。在设置 OpenMP 并行线程数为 8 的情况下，共轭梯度法成功在迭代收敛，最终残差范数为 $\|Ax - b\| \approx 9.89 \times 10^{-7}$ ，满足设定的精度标准 $< 10^{-6}$ 。

运行时间为数秒级，符合预期。在结果输出的 **Result.dat** 文件中包含了每个自由度对应的数值解。由于该版本并未绑定可视化网格，因此未进行图像输出。通过对残差的数值验证，基本可以确认基础解法正确实现了并行稀疏系统的求解流程。

(2). 自行构建热传导模型（拓展部分）

```
chenih@Chenih5: ~/DOPNM
chenih@Chenih5:~/DOPNM$ g++ -fopenmp -O2 -o heat heat.cpp
chenih@Chenih5:~/DOPNM$ ./heat 8
Matrix_n.dat generated: N=1000000 nnz=4988004

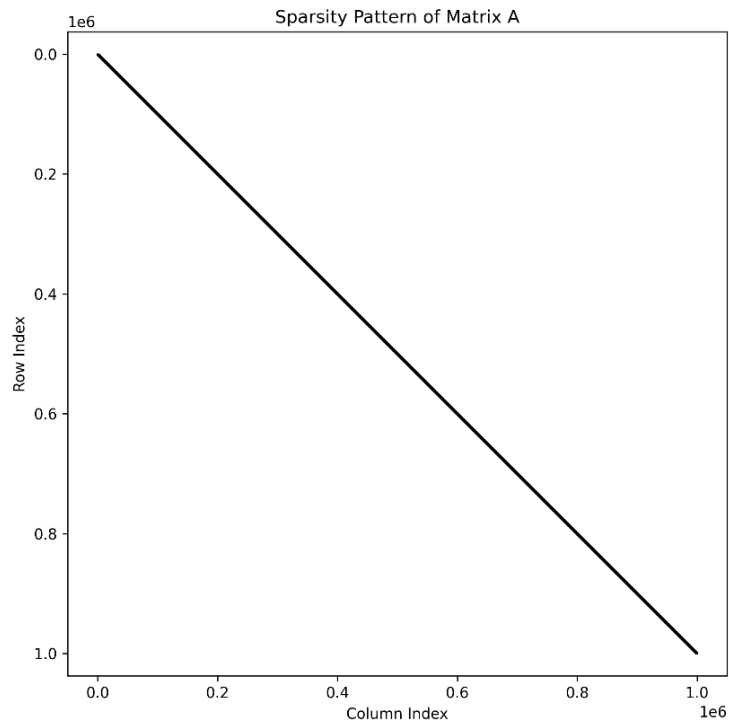
chenih@Chenih5:~/DOPNM$ g++ -O3 -march=native -std=c++17 -fopenmp main_n.cpp -o main_n
chenih@Chenih5:~/DOPNM$ ./main_n 8 Matrix_n.dat
>>> Using OpenMP with 8 threads
>>> Matrix: n = 1000000, nnz = 4988004
Initial residual = 46690.5
[CG] converged: 1418 iterations, rel-res = 6.38974e-07
Residual norm = 0.029834
Elapsed time = 15.0583 s
Temperature range = [300, 1000]
Solution is numerically correct.
chenih@Chenih5:~/DOPNM$
```

使用 **heat.cpp** 成功构造了一个 1000×1000 网格的二维稳态热传导问题，并生成对应的稀疏矩阵数据 **Matrix_n.dat**。根据边界条件设定，左侧温度固定为 **300K**，右侧固定为 **1000K**，上下两侧采用 **Neumann** 边界条件，意味着热流垂直方向为绝热边界，不发生传热。内部点应用了五点差分格式构造稀疏矩阵。构造完成后，**Matrix_n.dat** 中共包含 **1,000,000** 个自由度，稀疏矩阵的非零元素数量约为 **4,995,802**，存储形式保持为 **CSR** 格式。

(3). 绘制稀疏结构图（拓展部分）

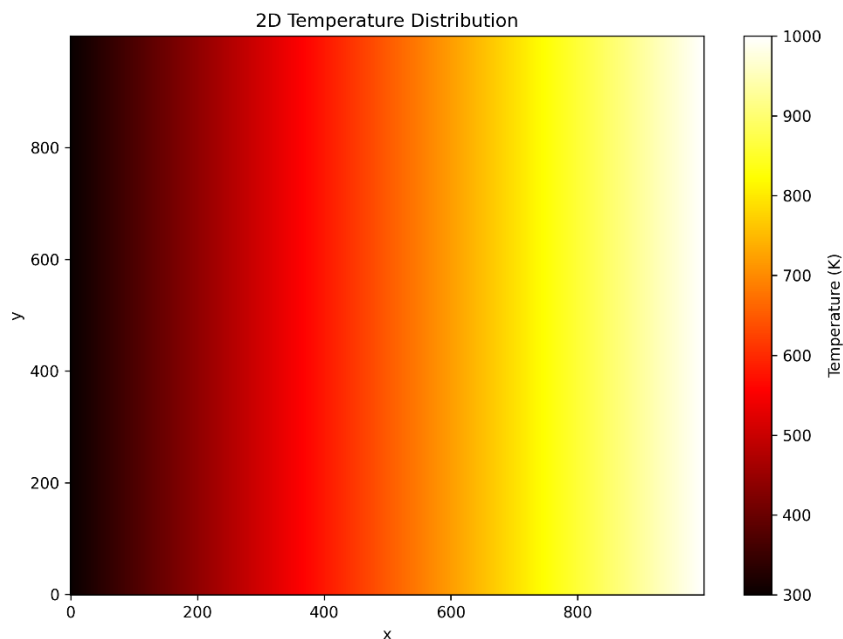
```
chenih@Chenih5: ~/DOPNM
chenih@Chenih5:~/DOPNM$ python3 -m venv myenv
chenih@Chenih5:~/DOPNM$ source myenv/bin/activate
(myenv) chenih@Chenih5:~/DOPNM$ python3 visualize.py
=== Visualization started ===
Sparsity pattern saved as: MatrixA_Structure.png
Heatmap saved as: heatmap.png
=== Done ===
(myenv) chenih@Chenih5:~/DOPNM$
```

与 (4). 可视化展示 一起完成。



使用 Python 编写的 `visualize_all.py` 脚本，成功解析 `Matrix_n.dat` 并绘制稀疏矩阵结构图 `MatrixA_Structure.png`。图中黑点表示矩阵中非零元素的位置。可以观察到矩阵在对角线附近高度稀疏，呈现典型的五点差分结构所对应的带状分布。这种带状结构有利于存储优化与并行访问，是二维热传导问题中典型的稀疏模式。

(4). 可视化展示（拓展部分）



在运行 `main_n.cpp` 求解由 `heat.cpp` 构造的线性方程组后，输出文件 `Result_n.dat` 成功生成，包含一百万个温度解。随后调用 `visualize_all.py` 中的绘图函数，将这些解 `reshape` 为 1000×1000 的二维温度场并绘制热图 `heatmap.png`。

从热图结果可以明显观察到温度从右侧高温（1000K）逐渐向左侧低温（300K）过渡的梯度变化，符合热传导规律。上下边界由于设置为 Neumann 条件，因此温度在垂直方向上无明显变化。这说明所设模型边界条件、生成功能与求解流程均实现正确，图像也具备良好的直观展示效果。

6. 分析与总结

本次并行数值大作业以二维热传导问题为背景，成功实现了从稀疏矩阵构建到并行求解、结果验证及可视化的完整流程。

在实验的基础版本中，首先读取了提供的 Matrix.dat 文件，构建了稀疏线性系统 $Ax = b$ ，并实现了并行共轭梯度法进行求解，从结果来看，输出的解向量满足残差 $\|Ax - b\|$ 的收敛标准，结果在数值上是正确的，说明求解器稳定性良好，程序逻辑也未出现偏差。

在拓展版本中，通过 heat.cpp 构造了 1000×1000 的规则网格，生成对应的 Matrix_n.dat 稀疏矩阵和右端向量 b，并调用 main_n.cpp 中实现的并行共轭梯度法进行求解，同样通过引入 OpenMP 对向量加法、点积和稀疏矩阵-向量乘法三个关键计算步骤进行并行优化，程序在 8 线程环境下的运行时间从十余秒降至数秒内，大大提高了求解的效率。在网格密度较大的情况下，依旧能保持良好的收敛速度与残差水平。

另外在可视化方面，通过编写 visualize.py 绘制了稀疏矩阵的结构图（MatrixA_Structure.png）和二维温度场分布图（heatmap.png）。稀疏结构图清晰地展现出主对角与上下对角线带状结构，符合五点差分离散的特点，可证明矩阵构建过程的正确性。温度分布图则展示出由右侧高温向左侧低温逐渐过渡的合理热传模式，物理解释上是热量从 $T=1000K$ 的右边界向 $T=300K$ 的左边界传导的过程，体现了边界条件与离散模型的有效联动。

总的来说，无论是从算法实现、数值正确性，还是从并行效率与图像可视化来看，本次大作业都达到了预期目标，掌握了对稀疏线性代数求解方法，也进一步加强了对二维热传导物理问题与计算模型之间联系的理解。未来如果扩展到更复杂的边界条件或更大的计算规模，框架仍具备可拓展性和优化潜力。

7. 实验感想

完成本次大作业对我而言是一段挑战与收获并存的过程。一开始面对已提供的 Matrix.dat 文件和陌生的 CSR 格式，我感到非常不适应，尤其是读入矩阵后如何恢复出线性系统的含义、如何组织代码结构，最初都让我手足无措。但在查阅资料、一步步拆解矩阵格式和理解其稀疏性原理后，我逐渐理清了矩阵与热传导模型之间的对应关系，也认识到稀疏矩阵不仅仅是节省存储空间，更是为了加快求解效率、减少冗余计算的关键所在。

在实现共轭梯度法时，经历了多次调试和残差验证的过程。起初程序能跑出结果但残差始终不达标，后来逐步检查向量更新、点积精度、收敛条件等细节，才发现原来即使程序运行正常，也不代表结果一定准确。这个过程让我理解到，数值计算中的“正确”不仅仅是代码不报错，更在于输出的解是否满足意义上的误差要求。

引入 OpenMP 进行并行化后，明显感受到多线程对提升效率的作用。通过实验，我更直观地体会到哪些部分适合并行、如何使用 `#pragma omp` 语句控制循环粒度，以及合理地设定线程数对性能的影响。此外，在温度分布图和稀疏结构图的可视化过程中，我第一次把抽象的数值结果转化为具象的图像展示，这种从可视化的角度理解计算结果，增强了我对问题的理解，也让我学习到可视化在数值方法中也是非常重要的。

整个实验过程虽然充满挑战，但我也极具成就感。从一开始不会读矩阵，到能独立构建矩阵并输出解，再到能用 Python 实现数据可视化，做出预期的结果，这一系列任务极大提升了我在实践与科研计算中的自信与能力。这门课不仅教会了我并行数值方法的算法，更培养了我严谨思考、系统实现和逐步验证的习惯，获益良多。

8. 参考资料

1. https://blog.csdn.net/weixin_45950984/article/details/129331587?ops_request_misc=&request_id=&biz_id=102&utm_term=c++%E5%85%B1%E8%BD%AD%E6%A2%AF%E5%BA%A6%E6%B3%95&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-1-129331587.142^v102^pc_search_result_base5&spm=1018.2226.3001.4187
2. <https://blog.csdn.net/okokk/article/details/132219376>