Weekly Research Progress Report

Student: 丛兴 **Date:** 9/13/2023-9/27/2023

List of accomplishments this week: (工作成果列表)

学习稀疏矩阵的定义和优化稀疏矩阵存储的方式

学习解决稀疏矩阵方程$Ax = b$的经典方法：最速下降法和共轭梯度法

了解优化大规模线性问题求解的 Schur-KS 架构

Paper summary:（文献总结）

Name：An Introduction to the Conjugate Gradient Method Without the Agonizing Pain

Motivation：之前学者对于 CG 的介绍过于抽象和难以理解。

Solution：通过对$Ax = b$线性方程的举例，配以大量的插图，利用由浅入深的方法，从最初的直接求解优化到 SD 方法，再对 SD 进行优化的过程中，提出 CD 方法，最后在 CD 基础上改进为 CG 方法。同时利用配图和公式推导，对 CG 的特性进行说明和阐述，最后提出现在 CG 的研究方向----利用预处理器对 CG 方法的矩阵进行优化，以使其达到更快的收敛速度。

Related to us：了解对稀疏矩阵方程$Ax = b$的典型求解算法，明白基本的过程和原理，对后面的计算架构的学习和优化奠定基础。

Name：大规模线性问题求解算法的高可扩展性研究

Motivation：科技进步使得求解问题的规模不断增大，应用问题模的模型结构越来越复杂，对计算精度的要求也在日益提高，而其精细模拟可以达到的**问题规模却严重受制于计算机体系结构和求解算法的可扩展性**。本文作者阐述了在科学工程应用的实际问题中，所涉及的三个领域----数理模型、数值求解方法、体系结构。而作者指出数理模型和数值求解方法的相互作用已经到达了相对系统的研究阶段，而**数值方法和体系结构的相互作用还在起步阶段**，并指出**实现三个领域之间的交叉与耦合是设计和实现高效数值计算算法的重点**。

因此，本文的作者立足现实，想基于**计算机体系结构和数值方法的耦合**，对大规模线性问题的迭代求解算法高校可扩展性问题进行研究。以此来设计出高可扩展性以及对典型实际应用问题具有良好求解性能的算法和架构。

Solution：作者指出，大规模线性问题的迭代并行算法，其**高可扩展性瓶颈主要体现在全局通信开销和全局通信频次上**。因此作者决定利用**降低基准通信开销**的方法来实现高可扩展性。而降低基准通信开销就意味着**需要选取通信极小化的处理器分组、矩阵分块**的策略。

作者进行优化的思想是：如果数值算法能够获得并行计算机体系结构自身存在的分层聚类的空间分布非均衡信息，然后据此信息来指导求解算法进行系数矩阵的分块策略，可能会降低基准通信开销。

作者基于**舒尔补（Schur-complement）方法**来对矩阵进行分块处理，从而实现基于区域分解的并行计算，降低其通信开销。随后，使用 **Petal-剖分方法**，对超算的节点分布进行聚类分组操作，对矩阵，同样使用 **Petal-剖分方法**，也得到相应的分块矩阵。从而使得超算处理器网络关联图的通信开销能够达到近似优化的量级。

Related to us： 本文利用了数值计算和体系结构相结合的方式来提升计算效率，从降低节点之间的通信消耗出发，利用 Petal-剖分大尺度聚类分组算法和舒尔补方法以及其他程序组件来架设解决大规模线性系统计算的计算架构，给计算优化提供了新的方向。

Work summary（工作总结）

优化稀疏矩阵存储的三种方式：CSR、CSC、COO

CSR：(压缩行进行存储)，在 CSR 格式中，稀疏矩阵由三个数组表示：值数组（V）按行优先顺序存储矩阵的非零元素、列索引数组（C）存储每个非零元素在 V 中的列索引、行指针数组（R）存储每行第一个元素前面非零元素的个数。

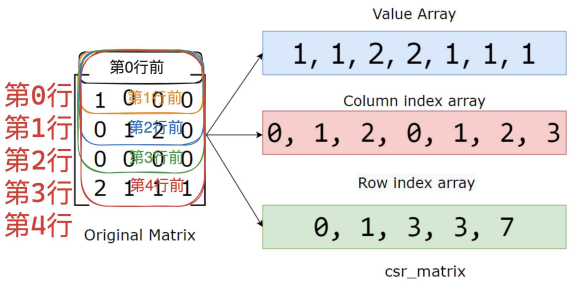

图 1 CSR 存储方式举例

通过 C++实现对 CSR 的模拟压缩存储、访问、还原：

存储：

```
for(int i=0; i<m; i++){
        row.push_back(row_count);
        for(int j=0; j<n; j++){
                if(ve[i][j]!=0){
                        value.push_back(ve[i][j]);
                        col.push_back(j);
                        row_count++;
                }
        }
}
row.push_back(row_count); // 最后要记得记录在整个矩阵中所有的非零元素的个数！
```

访问：

```
cout<<"输入要查询的坐标位置：";
int u,v;
cin>>u>>v;
int before_count = row[u]; // 前面行中非零元素的个数
int now_count = row[u+1]-row[u]; // 本行中非零元素的个数
// 定边界
for(int i=before_count; i<=now_count; i++){
        if(col[i]==v){ // 如果在区间中列号相匹配，则对应的 value 就是要读取的 value 值
                cout<<"数值为："<<value[i]<<endl;
                return 0;
        }
}
cout<<0<<endl;
```

还原：

```
// 计算二维矩阵的行数
int row_normal = row.size()-1;
// 计算二维矩阵的最小列数
int col_min_normal = 0;
for(auto e : col){
        col_min_normal = max(e,col_min_normal);
}
vector<vector<int>> ve_normal(row_normal,vector<int>(col_min_normal+1,0));
int value_index = 0; // value 起始索引
for(int i=0; i<row.size()-1; i++){
        int begin_cnt = row[i]; // 确定第 i 行所属的 col_index 数组的起始范围
        int end_cnt = row[i+1]; // 确定第 i 行所属的 col_index 数组的终止范围
        for(int j=begin_cnt; j<end_cnt; j++){
                ve_normal[i][col[j]] = value[value_index];
                value_index++;
        }
}
```

与 CSR 压缩存储相对应的是 CSC 压缩存储，CSC 压缩方式和 CSR 压缩方式是类似的。它们两者的主要区别是：**CSR 是压缩的行，而 CSC 是压缩的列**。在 CSC 中，矩阵的列被压缩存储，只有非零元素的值、对应的行索引以及每一列的起始位置之前非零元素的个数被存储。

还有 COO 压缩存储，在 COO 中，矩阵中非零元素的坐标和对应的值被分别存储在三个单独的数组中。具体来说，COO 存储方式使用三个数组：一个存储非零元素的行坐标，一个存储非零元素的列坐标，以及一个存储对应的值。

COO 的存储方式一般来说，它**适用于非常稀疏的矩阵**，也就是说其中大部分的元素都是零。除此之外，COO 存储方式支持快速的随机访问和元素插入操作。但是，如果矩阵的稀疏程度不够，则 COO 存储方式可能会浪费大量的存储空间。

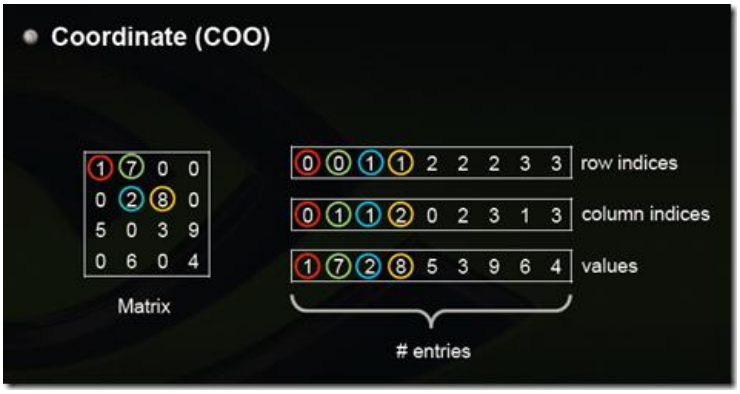

图 2 COO 存储方式示例

升维求解线性方程 Ax=b 的方法

对于 Ax=b 问题（A 矩阵是对称而且是正定的），当 A 的维度是 2 维时，它可以形象的表示为两条直线的交点。如果直接去求解的话，就是 $x=A^{-1}b$，也就是交叉超平面的解决方法，但是通常情况下， 是比较难以求得的，所以需要通过一定的转换，从而可以实现迭代求解，也就是利用--最速下降法或共轭梯度法去求解。

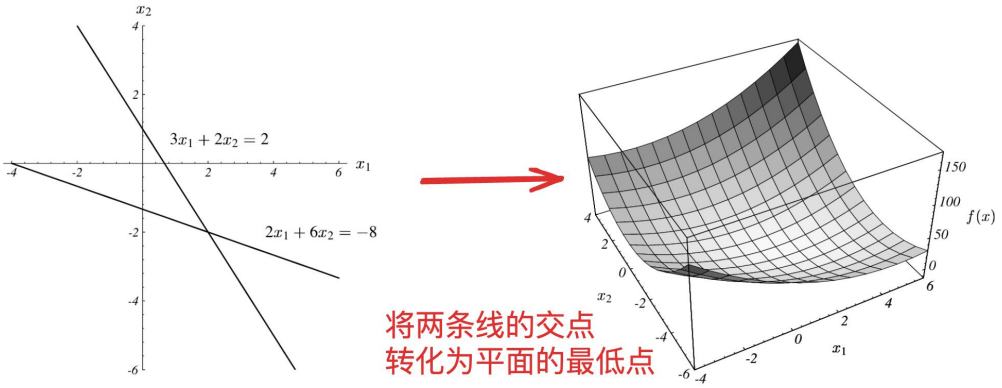

图 3 升维转换图

升维求解的理论依据：

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c$$

$$f(x)' = \frac{1}{2}(A + A^T)x - b$$

$$because\ A\ is\ symmetric\ A = A^T$$

$$f(x)' = Ax - b$$

$$f(x)' = 0$$

$$Ax = b$$

升维后使用最速下降法求解最终解

最速下降法基于以下这个公式：

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} r_{(i)}$$

最速下降法的大致步骤如下：

【第一步】设置初始点，确定最快下降方向

最快下降的方向是函数在该点的梯度的反方向

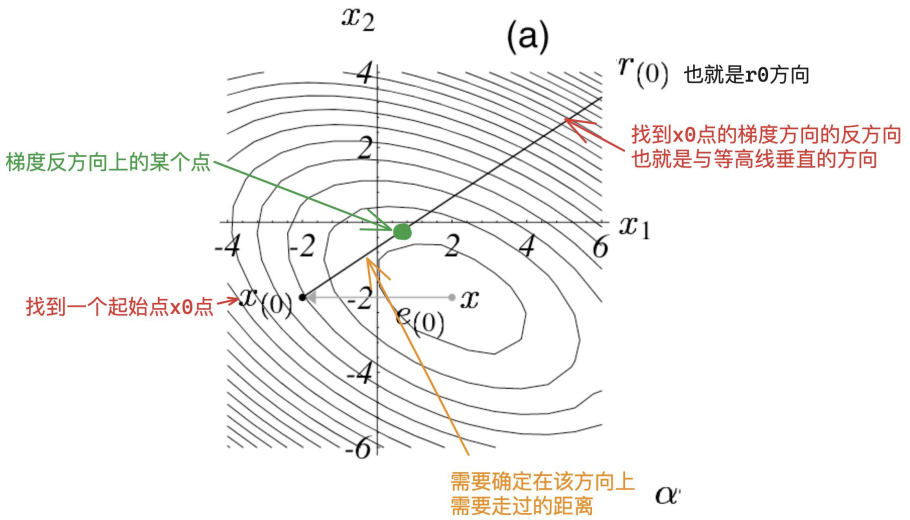

图 4 定点找方向

【第二步】确定在最速下降方向上的步长

在确定步长的时候，使用到求导的方法，从而证明了两个相邻点的梯度方向是相互正交的，也就是两个点的梯度乘积为 0，利用公式替换，从而确定 a 的值。

$$\begin{aligned}
r_{(1)}^T r_{(0)} &= 0 \\
\left(b - Ax_{(1)}\right)^T r_{(0)} &= 0 \\
\left(b - A\left(x_{(0)} + \alpha r_{(0)}\right)\right)^T r_{(0)} &= 0 \\
\left(b - Ax_{(0)}\right)^T r_{(0)} - \alpha\left(Ar_{(0)}\right)^T r_{(0)} &= 0 \\
\left(b - Ax_{(0)}\right)^T r_{(0)} &= \alpha\left(Ar_{(0)}\right)^T r_{(0)} \\
r_{(0)}^T r_{(0)} &= \alpha r_{(0)}^T \left(Ar_{(0)}\right) \\
\alpha &= \frac{r_{(0)}^T r_{(0)}}{r_{(0)}^T Ar_{(0)}}.
\end{aligned}$$

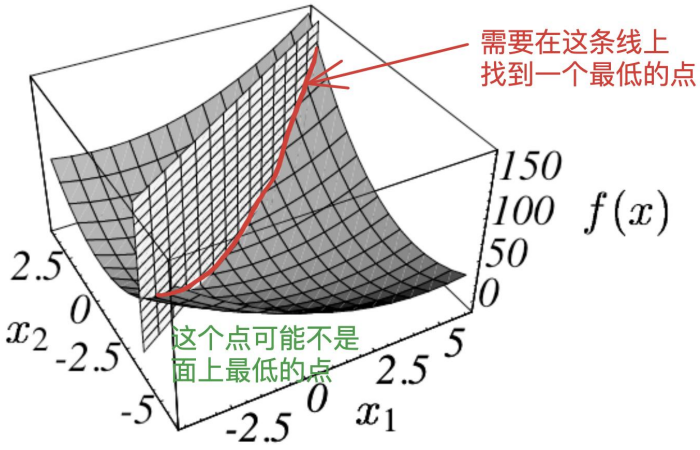

图 5 确定步长

【第三步】进行循环，直到找到最小值

通过上面的步骤，就可以源源不断的推算出下一个点的位置，直到找到最小的点。

循环过程如下：

$$r_{(i)} = b - Ax_{(i)},$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}},$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} r_{(i)}$$

$$r_{(1)}^T r_{(0)} = 0$$

每次迭代的方向都是垂直的

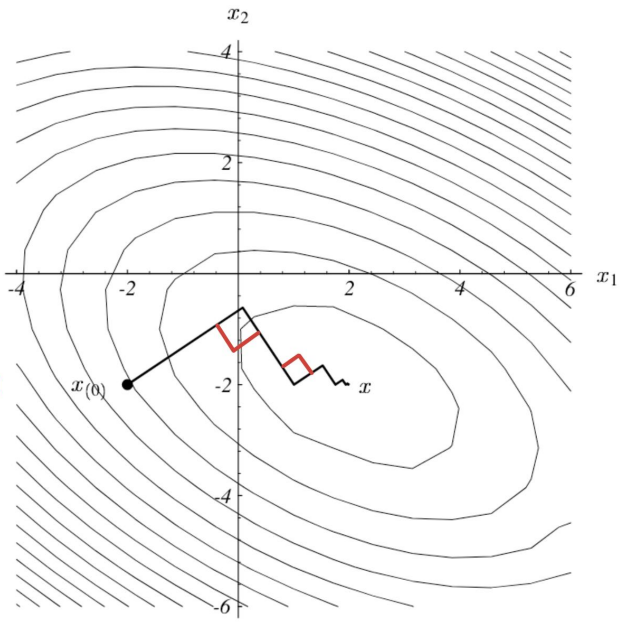

图 6 最速下降法的求解路径

观察最速下降法的曲线路径可以得到，最速下降法的前进方向会有重复的现象。

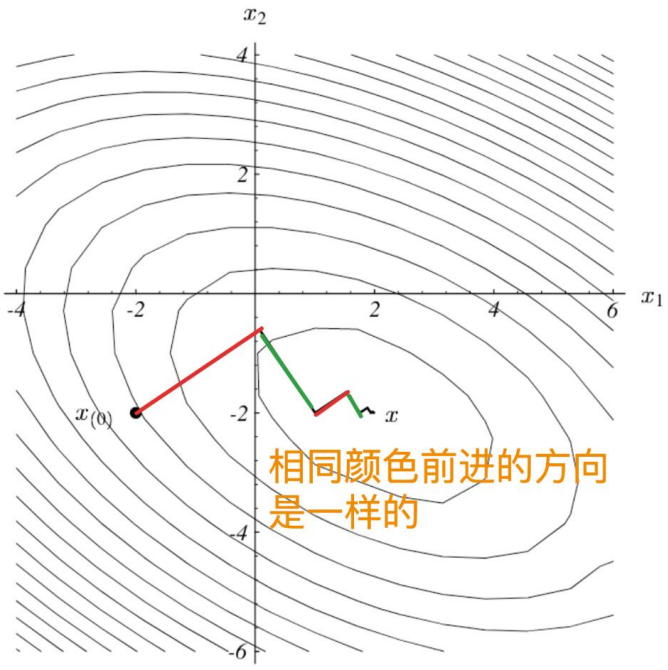

图 7 最速下降法前进曲线

如果在每个特定的方向上只前进一步，也就是说，对于一个 n 维的线性方程，只需要对多 n 次迭代，就可以达到最终的解。该种方法是存在的，就是共轭方向法。

共轭梯度法的由来

在共轭方向法中，需要一组不相关的变量来构造一组搜索向量，也就是在某一个搜索方向上，仅仅前进一步的那个方向向量。而共轭梯度法就是将每个点的梯度的反方向值作为这一组不相关的构造向量来进行使用，因此被叫做共轭方向法。

我所理解的选用每个点的梯度反方向（residuals）值作为构造向量来进行使用的原因是：residuals 是梯度的反方向，它在具有一个很好的性质，就是 residual 向量和之前的搜索向量相互正交，而之前的搜索向量都是通过之前的 residual 构造而来的，所以新的 residual 与之前的 residual 都相互正交，也就是乘积为 0，因此，在计算新的搜索方向向量时，就没有必要再存储之前旧的梯度向量了，这就克服了共轭方向法中构造搜索向量 d 需要大量存储之前搜索方向向量的缺点。

因此，共轭梯度法的求解过程如下：

$$
\begin{aligned}
d_{(0)} &= r_{(0)} = b - Ax_{(0)}, \\
\alpha_{(i)} &= \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}} \\
x_{(i+1)} &= x_{(i)} + \alpha_{(i)} d_{(i)}, \\
r_{(i+1)} &= r_{(i)} - \alpha_{(i)} A d_{(i)}, \\
\beta_{(i+1)} &= \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}}, \\
d_{(i+1)} &= r_{(i+1)} + \beta_{(i+1)} d_{(i)}.
\end{aligned}
$$

共轭梯度法的收敛性分析

$$
\begin{aligned}
\|e_{(i)}\|_A^2 \leq & \min_{P_i} \max_{\lambda \in \Lambda(A)} [P_i(\lambda)]^2 \sum_j \xi_j^2 \lambda_j \\
= & \min_{P_i} \max_{\lambda \in \Lambda(A)} [P_i(\lambda)]^2 \|e_{(0)}\|_A^2.
\end{aligned}
$$

具体过程已记录在笔记中，通过上面的公式可以得出，当右面的等于 0 时，相当于达到了收敛。通过分析前 3 维的收敛效果，可以得到如下结论：

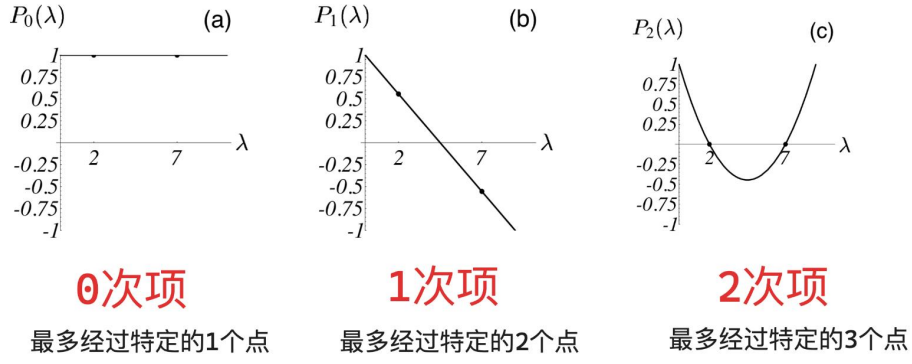

图 8 前 2 次迭代收敛分析

根据上述的公式，当没有迭代时，也就是 i=0 时，误差范围并不会降低，当第一次迭代时，也就是 i=1 时，特征值 2、7 对应的$P_1(\lambda)$变小，所以误差范围会进行缩小，当第二次迭代时，也就是 i=2 时，特征值 2、7 对应的$P_2(\lambda)=0$，此时根据上述公式，误差范围缩小到 0，因为最大值特征值对应的值为 0，所以此时收敛到最终的结果。

根据上述公式，也可以进一步推断出，**如果存在重复的特征值的话，CG 将会收敛的更快**。

因为，一个 n 次项的多项式可以限制符合 n+1 个点的坐标，除了初始点值外，还可以最多符合 n 个不同的特征值。而 CG 又是必定在 n 次内进行收敛的，所以，最坏的情况就是，A 具有 n 个各不相同的特征值，只有当迭代 n 次时，最大的特征值对应的 n 次多项式才能取到 0 值。但是如果说，**特征值有重复的话，也就是说具有 n-k 个不同的特征值，也就是说最多迭代 n-k 次就一定能保证最大的特征值对应的 n-k 次多项式取到 0 值，也就是能够收敛到最终的结果**。因此也可以得出这个结论：计算精确解所需的迭代次数最多可以等于不同特征值的数量(因为上述公式中需要取最大值的缘故)

更近一步的说：当特征值分布越密集时，收敛的速度也会越快。

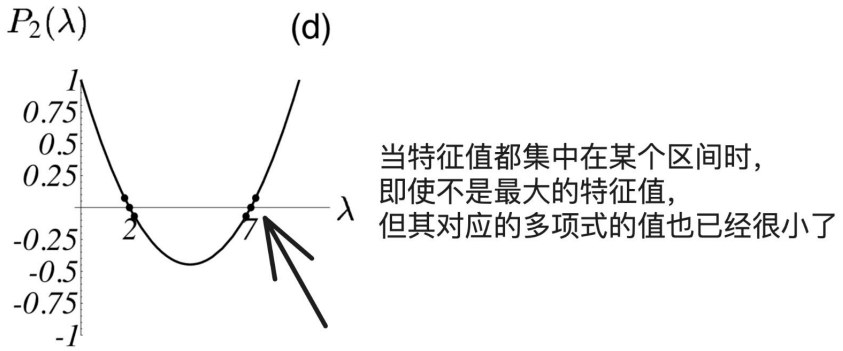

图 9 特征值分布密集

当特征值分布越密集时，CG 收敛的速度越快，因为在这种情况下，即使不是最大特征值，但都集中在最大特征值的左侧，其对应的多项式的值也一定已经很小了，所以误差值 e 也会变得很小，也就是已经很接近正确的结果 x 的值了。

最后，给出共轭梯度法收敛的具体数学表达式从而进一步证实上面的说法：

$$\|e_{(i)}\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \|e_{(0)}\|_A.$$

其中，k 表示矩阵 A 的**条件数**，它的值等于矩阵 A 的特征值的最大值除以矩阵 A 的特征值的最小值。

大规模线性问题求解算法的高可扩展性研究

三个领域：

在科学工程应用的实际问题当中，数理模型、数值方法、体系结构等领域具有自身的特点：特定的分层耦合和非均衡性。

图 10 三个领域

三个领域之间的交叉和耦合是设计和实现数值计算算法的重点。

实现高可扩展性的三种途径

实现高可扩展性的三个途径是：降低求解器的整体迭代次数----需要选取高逼近精度的预条件子、降低基准通信开销----选取通信极小化的处理器分组、矩阵分块策略、改进 Krylov 子空间迭代算法。而大规模线性问题的迭代并行算法，其高可扩展性瓶颈，主要体现在全局通信开销和全局通信频次上。（基准通信开销：（也就是每个迭代步的通信开销）与逆矩阵的非零元个数成正比，全局通信频次：与处理器群组构成的连通子图之间的割边数量成正比）

本文思路以及实现方式

如果数值算法能够获得并行计算机体系结构自身存在的分层聚类的空间分布非均衡信息，然后据此信息来指导求解算法进行系数矩阵的分块策略，可以获得提高。所以文章提出**高可扩展性的线性问题求解架构（Schur-KS）**，该架构是基于 **Petal-剖分**和**舒尔补方法**而设计的。

第一部分是 Petal-剖分，**剖分子集之间的割边集小，意味着全局通信的代价低廉**。而本文设计的高可扩展性的求解架构，其并行部分是分级进行的。

第一级是大尺度的聚类分组

将现有的空闲处理器资源分解为若干个处理器群组（clusters），将现有的计算区域分解为若干个计算子区域，第一级的聚类分组不需要做负载均衡上的考虑，它所要实现的是聚类分组的通信极小化，**通信极小化问题在第一级剖分中对算法整体的可扩展性影响最大**，应该作为唯一的约束，进行剖分算法的近似优化。

第二级是中小尺度级别的并行分解

在每个处理器群组和计算子区域的内部，进行中小粒度的处理器分组和计算区域的分解。在足够多的未知量和处理器的情况下，负载均衡问题放在第二级剖分中进行实现。

在 Petal-剖分算法的并行化设计中，文章还采用了**分布式存储的图数据格式**和**并行化的 dijkstra 最短路径算法、分布式的广度优先搜索算法等。**

第二部分是舒尔补方法，它通过图剖分的策略将原始的线性系统 Ax=b 重新排序为如下结构：

$$\begin{pmatrix} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ F_1 & F_2 & \cdots & F_k & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \\ g \end{pmatrix}$$

因此，在并行计算过程中，矩阵主对角子块在相应的处理器群组中进行运算，其存放于相应处理器的局地内存中，非主对角的矩阵子块构成了不同处理器之间的相互作用（interaction），需要调用 MPI 通信库函数进行消息传递。在这个视角下，舒尔补方法形成了权重连通图的聚类剖分。

实现结果

核数（万核）	0.8	1.6	3.3	6.6	13.3	26.6	53.2	106.5
MPI 进程数（万）	0.8	1.6	3.3	6.6	13.3	26.6	53.2	106.5
非零元数（万）	67	133	267	535	1070	2141	4283	8567
迭代时间（秒）	140.4	169.1	193.8	219.1	288.4	313.3	533.0	858.9

图 11 大规模环境下高扩展性测试

数据名称	非零元数 (万)		核数				
			256	512	1024	2048	4096
			MPI 进程数				
			256	512	1024	2048	4096
		求解方法	并行效率 (%)				
Maxwell 方程	494	Schur+Petsc	100	99.9	99.4	90.7	81.3
		直接调用 Petsc	100	106.4	98.2	79.8	56.2
Helmholtz 方程	367	Schur+Petsc	100	99.6	99.6	95.3	95.0
		直接调用 Petsc	100	106.3	92.3	81.7	75.5
扩散对流反应方程	367	Schur+Petsc	100	97.6	96.0	87.0	81.5
		直接调用 Petsc	100	97.2	88.7	78.1	63.3
Flange	234	Schur+Petsc	100	95.2	90.4	79.9	70.2
		直接调用 Petsc	100	101.3	89.8	66.0	44.6
Enginefan	430	Schur+Petsc	100	97.5	97.8	84.5	73.9
		直接调用 Petsc	100	92.6	83.1	68.0	39.7

图 12 中小规模下的强扩展性测试

从图 11 结果可以看出，当非零元的个数以 n 倍增加时，迭代次数的增加倍数却逐渐趋缓且低于 n 倍，故在大规模计算上就有较高的可扩展性。

从图 12 结果可以看出，随着核数的进一步增加，受到**通信量**等因素的影响，直接调用 Petsc 的求解方式的并行效率下降较多。通过 Schur-KS 架构调用 Petsc 的求解方式比直接调用 Petsc 的求解方式，其并行效率方面有较为明显的效果提升，其强可扩展性也相对较好。但是由于缓存的缘故，当核数不是很多时，并行效率具有超线性加速比的效果，这是因为高速缓冲的存在，由高速缓存减少的计算耗时弥补了因数据通信、负载均衡等所造成的额外时间开销.

Next（下一步）：

继续阅读并行计算优化方面的文章