**注意：本文是对**[**https://arxiv.org/pdf/1907.05309.pdf**](https://arxiv.org/pdf/1907.05309.pdf)**的翻译，尝试通过翻译的方法提高学习的效率。请看原文获取更准确的信息。**

**State-of-The-Art Sparse Direct Solvers**

Matthias Bollhöfer, Olaf Schenk, Radim Janalík, Steve Hamm, and Kiran Gullapalli

# 大规模稀疏矩阵直接求解器的最新进展

**摘要：**本文章讲述最新的稀疏消除方法。这些方法被用于组合算法的预处理阶段，以提高对角优势，减少填充，提高并发性， 从而获得高效的并行计算。此外， 这些方法侦测密集子矩阵，这样可以调用Blas Level-3的密集矩阵核函数计算。我们将演示先进的稀疏直接求解法如何显著提升电路仿真问题。

## 1 简介

求解大规模线性系统是许多工程技术的核心应用问题。最新的计算机体系结构集成越来越多的内存资源，更多的计算核心，这些变化极大地推进了直接求解器的组合算法，使得求解大线性系统成为可能。

其中特别支持最新的最佳组合算法提高了对角优势，铺平了静态旋转方法的一条道路，这将显著地提高分解阶段的效率。另外，系统的划分和重排序方法可以获得很高的并发性，并减少填充。这些最新的成就可以显著地加快分解阶段，现在CPU的体系结构要求程序利用缓存执行尽可能多的操作。这个要求可以通过分解阶段的密集子块获得，这些密集子块可以被组合为密集子矩阵，能够被BLAS和LAPACK[3]的多线程、缓存优化的密集矩阵核函数处理。

1

本章将回顾大规模稀疏矩阵直接求解的基本方法，最新进展和最新的L-U分解方法。本篇文章组织如下。第二节，我们从最大权重匹配法开始，它是组合优化的一个关键工具，可以显著地提高系统的对角优势性。接着，第三节讨论多层嵌套分割，一种组合方法以对称地重排序系统，填充可以被减少，并且并行性可以被提升，同时旋转性可以被或多或少忽略。此后，我们将在第四节介绍图理论，特别是消除树，这将推导出LU分解的大部分特性。这些特性包括分解过程中密集子矩阵的预测。用这种方法，后续几个列的因子L和*U*T可以被组合为一个密集块。这是使用优化的BLAS Level-3的密集矩阵核函数的基础，第五节将条论这些核函数，它们能够使用缓存的多层结构提供快速计算。最后第六节我们将演示一个应用，电路模拟，展示最新的稀疏直接分解器如何用最新的优化技术加速计算。我们假设读者已经熟悉基本的图理论【15，21】，基于图理论的计算算法【1】

## 2 最大权重匹配

最新的稀疏消除方法获得成功的关键因素是能够产生高效的数据结构和对应的数值模板。如果我们能够越早加大对角线元素的尺寸，则更加常常不处理高斯消除中的旋转，我们可能使用静态数据结构，数值计算可以被显著地加速。获得这个目标的常用方法是最大权重匹配法【16，37】，它重新排列一个非奇异矩阵*A* ∈ Rn,n的行，使用一个重排矩阵Π ∈ Rn,n使得 ΠT *A*的*对角元素非0。*并且， 它最大化对角元素的绝对值乘积，产生两个对角线缩放矩阵, *D*r, *D*c ∈ Rn,n 使得*A*˜ =ΠT *D*r *AD*c满足对于使用的i， j = 1,…, n, |*a*˜ij|<= 1，|*a*˜ii| = 1。这个想法的原意是通过非对称排列和缩放，找到一个符合**最大权重匹配**的**二分图**。最大权重匹配是一个已知的运筹学和组合分析的分配问题。【译者： 参考附录1】。

**定义1** 一个图G= *G* = (*V*, *E*)，包含定点集V，和边集*E* ⊂ *V*2 ，如果满足下面条件被称为**二分图**，即V可以被分成两个集合，*V*l和*V*r，使得没有边*e* = (v1, v2) ∈ *E，*它的两个顶点v1, v2 都在*V*l,或者v1, v2 都在*V*r，换而言之，任何一条边的两个顶点必须有一个属于*V*l，而另外一个定点必须属于*V*r。

**定义 2** 给定一个矩阵A，我们可以把它映射到一个准确的二分图Gb(A)=(Vr,Vc,E) ,通过设置*V*r = {*r*1, . . .,*r*n}的标记为矩阵A的行索引，把列索引标记为{*c*1, . . ., *c*n}。这时E被定义为*E* = {(*r*i, *c*j)| *a*ij ≠ 0}。

对于二分图*G*b(*A*)，我们不难发现如果*a*ij ≠ 0，那么我们有定点*r*i ∈ *V*r行集合连接到一个条(*r*i, *c*j) ∈ *E* ，另一个顶点*c*j ∈ *V*c列集合。但是不存在行连接另外一个行， 也不存在列连接两外一个列。

**定义3** 图 *G* = (*V*, *E*)的一个匹配M是e ∈ E的一个子集，使得没有两个遍共享一条边*。*

如果M是一个二分图*G*b(*A*)的一个匹配，那么每条边*e* = (*r*i, *c*j) ∈ M对应于一个行Ri和一个列Cj，不存在其他边*e*ˆ = (*r*k, *c*l) ∈ M含有相同定点，也不存在Rk=Ri或者Cl=Cj.

**定义4** 图*G* = (*V*, *E*)的一个匹配是最大的，如果没有E的其他边可以被加入M。

如果对于n x n矩阵A，有一个*G*b(*A*)的匹配M，具有最大基数，那么边被定义为(*i*1, 1), . . ., (*i*n, *n*)，其中*i*1, . . .,*i*n也是数字1, . . ., *n*的某种顺序，那么我们得到*a*i1,1 ≠ 0，. . .， *a*in,n ≠ 0。此时，我们已经确定矩阵A至少是结构性非奇异，我们可以使用一个行置换矩阵ΠT，映射为行顺序*i*1, . . .,*i*n,以放置一个非0值到矩阵ΠT *A*的每个对角位置。

**定义 5** 一个完全匹配是一个基于基数n的最大匹配。

可以证明一个结构非奇异矩阵A总是存在一个完美匹配M。

## 完美匹配

在图1中， 一组边= {(1, 2), (2, 4), (3, 5), (4, 1), (5, 3), (6, 6)}表达了图*G*b(*A*)一个完美最大匹配。右边是二分图Gb(A)和完美匹配M.

Figure

1

2

3

4

5

6

1

3

4

5

6

2

1

2

3

4

5

6

1

3

4

5

6

2

|  |
| --- |
| 原始矩阵 |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | 3 | 0 | 2 | 0 | 0 | | 3 | 0 | 0 | 4 | 0 | 1 | | 0 | 0 | 0 | 0 | 3 | 0 | | 2 | 4 | 0 | 0 | 1 | 0 | | 0 | 0 | 3 | 1 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 2 | |

|  |  |
| --- | --- |
|  |  |

最后是行置换矩阵ΠT A

|  |
| --- |
| 行置换矩阵 |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2 | 4 | 0 | 0 | 1 | 0 | | 1 | 3 | 0 | 2 | 0 | 0 | | 0 | 0 | 3 | 1 | 0 | 0 | | 3 | 0 | 0 | 4 | 0 | 1 | | 0 | 0 | 0 | 0 | 3 | 0 | | 0 | 1 | 0 | 0 | 0 | 2 | |

**定义6** 如果一个边*e* = (*u*, v) 在图*G* = (*V*, *E*) 连接两个顶点*u*, v ∈ *V*,那么我们把它记作符号*u*v。一个路径包含边集合*u*1*u*2,*u*2*u*3,*u*3*u*4 . . .,*u*k−1*u*k, 其中每一个(*u*i,*u*i+1) ∈ *E*, *i* = 1, . . ., *k* – 1。

如果一个图 *G*b = (*V*r,*V*c, *E*) 是一个二分图，那么可以定义一个路径，任何一个路径是连接*V*r 和 *V*c,顶点。也就是说，图Gb的路径可以是下面的实例*r*1*c*2, *c*2*r*3,*r*3*c*4, . . . .

**定义7** 给定一个图*G* = (*V*, *E*), 一个顶点被称为是自由的，当它不属于图G的一个匹配的任何一条边。一个匹配存在一条交替通路 P = *u*1*u*2,*u*2*u*3, . . .,*u*s−1*u*s,其中它的边是交替的在E\M和M。一个相对于匹配M的增广路径是一个奇数长度的交替路径，它的顶点端点都是自由的。（译者暂时无法完整翻译这段话）

## 增广路径

考虑图1.为了更好地区分行订点和列顶点，我们用1 , 2 , . . ., 6 来代表行，用 , ,. . . , 来代表列。一个不完美但是最大匹配为*M* = {( 4 , ), ( 1 , ), ( 6 , ), ( 2 , ), ( 5 , )}. 我们可以很容易找到一个行和列的增广路径， 3 , 4 , 4 , 1 , 1 , 6 , 6 , 2 , 2 , 5, 5 . 两个端点3 和 都是自由的。

一个二分图*G*b = (*V*r,*V*c, *E*)中， 任何一个增广路径的一个端点必须是在*V*r,同时另一个端点必须在*V*c。两个边集合A,B的对称性差异，*A* ⊕ *B*被定义为 (A \ B) ∪ (B \ A).

使用这些定义和记号，下面理论【5】给出一个寻找一个二分图的最佳匹配的构建算法。

定理1如果M是二分图*G*b = (*V*r,*V*c, *E*)的一个非最大匹配，那么存在一条增广路径P相对于M，使得*P* = M ⊕ M˜，而且M˜是基数|M| + 1的一个匹配。

根据这条定理，一个寻找二分图最佳匹配的方法是寻找一条增广路径。

我们到此为止讨论的最佳匹配仅仅考虑矩阵的非0元素。为达到这个目标，需要在LU分解前的静态置换操作能够使得对角线元素的乘积的绝对值最大。它被称为最大权重匹配。这时需要一个置换矩阵π，使得最大化下面公式（1）

（1）

最大化这个成绩可以被转化为下面求和的最小值。我们定义一个矩阵*C* = (*c*ij)，

其中 *a*i = maxj |*a*ij|是这个矩阵A的行row i的最大元素。一个置换矩阵π具有最小化下面的和：

这个矩阵同时能够最大化公式（1）的乘积。这个最小化问题在优化中被称为线性求和赋值问题，或者二分权重匹配问题。这个问题被Hungarian用稀疏变化求解。稀疏矩阵具有τ个元素的计算复杂度为O(nτ logn)。如果矩阵的关联图满足特殊要求，那么时间复杂度可降低为to O(*n*α(τ + *n*log*n*) ，且α < 1。所有由有限差分或有限元离散得到的图均满足上述条件【24】。我们最终可以定义一个非对称置换获得一个完美匹配。

当我们解决这个赋值问题，需要计算出两组向量*u* = (*u*i) 和 v = (vi)，并且满足：

*u*i +vj = *c*ij (*i*, *j*) ∈ M, (2)

*u*i +vj ≤ *c*ij 其他. (3)

这些向量可以使用幂函数对初始矩阵做缩放操作。

请参考下面的公式4和公式5定义对角矩阵*D*r 和 *D*c ：

|  |  |  |
| --- | --- | --- |
| *D*r = diag(*d*1r, *d*2r, . . ., *d*nr), | *d*ir = exp(*u*i), | (4) |
| *D*c = diag(*d*1c, *d*2c, . . ., *d*nc), | *d*jc = exp(vj)/*a*j. | (5) |

使用公式（2）(3)和矩阵C的定义，我们可以推导出*A*˜ =ΠT *D*r *AD*c满足下述条件（6）和（7）：

|  |  |
| --- | --- |
| |*a*˜ii| = 1, | (6) |
| |*a*˜ij| ≤ 1. | (7) |

经过置换和缩放变换的矩阵*A*˜显著地具有更好的数字特性，无论被用来做直接切结或者预设条件的迭代求解器，参考文章【4，16】。Olschowka 和 Neumaier [37]介绍了这些缩放和置换方法， 以减少整个矩阵的高斯消元法的旋转操作。Duff 和 Koster [16]第一个实现了稀疏矩阵问题的求解。对于对称阵|*A*|，这些非对称匹配方法可以被转化为一个对称置换p和一个对称缩放Ds = (DrDc)1/2，从而使得*P*T *D*s*AD*s*P*具有大部分的对角1x1块和2x2满足相似的条件（6）（7），而且实际上很少发现1x1的块等于0【17】。最近，【29，28】提出了极为成功的并行算法来计算最大权重匹配。

## 最大权重匹配

现在对本节进行总结，我们将展示最大权重匹配的有效性，使用SuiteSparse矩阵数据集的简单矩阵实例“west0479”。Matlab可以用 “load west0479” 直接加载这个矩阵。图2中我们将展示这个初始矩阵和经过最大权重匹配置换后的矩阵。为了展示更好的对角占优，我们进一步计算矩阵A的每一行*r*i = |*a*ii|/∑nj=1 |*a*ij|，和*A*˜ =ΠT *D*r *AD*s, *i* = 1, . . ., *n*。*r*i可以可以被看做一个相对于行(i)的一个相对对角占优，这样产生一个（0，1）之间的一个数，进一步，如果*r*i >1/2 ,这一行就是严格对角占优，或者说., |*a*ii| > ∑nj:j≠i|*a*ij|。在图2中，我们展示了对于矩阵根据这个值做做重排序，从而获得一个参考线。我们看到最大权重匹配极大地提高了这个矩阵的对角占优，因此找到一个不完全或者完全LU分解的旋转方法。

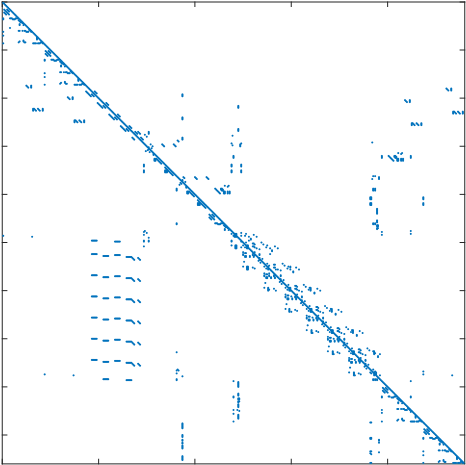
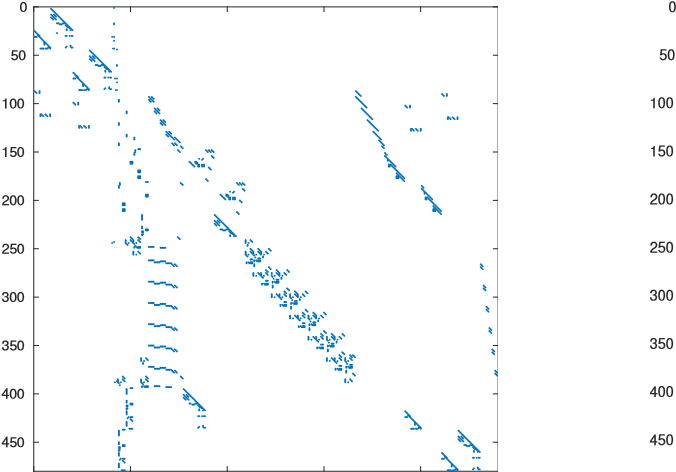


Figure 左边：原始矩阵；右边：经过最大匹配后的置换矩阵

# 3 符号对称重排序

当处理一个超大稀疏矩阵时，对该矩阵分解的计算时间的一个决定性因素是产生的填充数量。为了减少复杂度，主要采用对称重排序的方法启发式地减少填充数据时间。这里我们展示其中的一个方法，被称为嵌套分割法。

100

200

300

400

number of rows

0

0.2

0.4

0.6

0.8

r

i

100

200

300

400

number of rows

0

0.2

0.4

0.6

0.8

r

i

Figure 对角占优性。左边：ri，右边：ri A˜ =ΠT Dr ADc

选择这个方法主要是因为他可以被很容易做并行计算。

## 3.1 多层嵌套分割

多处理中最早使用直接分解法的递归多层嵌套分割。如果使用并行直接法求解稀疏方程，那么需要图分解算法来计算填充减少排序，这么可以实现分解阶段的高度并行性。

**Definition 8**对于一个矩阵A∈ Rn,n ,我们定义一个关联（直接）图 *G*d(*A*) = (*V*, *E*)，其中*V* = {1, . . ., *n*}和它的边集合*E* = (*i*, *j*)| *a*ij ≠0。非直接图*G*d(|*A*| + |*A*|T ) ，我们简单标记为*G*(*A*)。

使用图术语，稀疏矩阵A的任何一个非零元素aij有Gd(A)的一个直接边(i, j)，而在G(A)中忽略边的方向。

20世纪90年代对于图分割方法的研究产生了一大批高质量的软件包，例如Metis [25]。这些方法通常计算重排序，一方面使得分解方法产生较小的填充，同时能够提供很好的并行性。我们简要回顾使用图分解的多层嵌套分割的主要思想。

**定义9**设 *A* ∈ Rn,n 和它的图*G*(*A*) = (*V*, *E*)。一个K路图分割包含分割V为多个不连续的子集*V*1,*V*2, . . .,*V*k ， *V*i ∩*V*j = ∅ 其中*i* , *j* ∪i*V*i = *V*。这个子集*E*s = *E* ∩Ui,j(*V*i ×*V*j) i被称为边分离项.

典型地，我们希望一个k路分割是均衡的，也就是说*V*i应该满足|*V*i| ≈ *n*/*k*。边分离*E*s包含那些从图中拿走的边，这样可以产生k个离散的子图映射到*V*1, . . .,*V*k，*E*s的元素数目被称为裁剪边数。

**Definition 10**对于给定 *A* ∈ Rn,n,*G*(*A*) = (*V*, *E*)的一个**顶点分离项***V*s是一组顶点，使得存在V的一个k路分割*V*1,*V*2, . . .,*V*k ，对于任何i，j不存在边 *e* ∈ *V*i × *V*j 。

一个有效的分割*V*s不仅将*G*(*A*)分割为映射到*V*1, . . .,*V*k的k个独立的子图，而且它应试图使得变得∪ik=1|{*e*is ∈ *V*i, *s* ∈ *V*s}|数目比较小。

嵌套分割将递归构造**顶点分离项***V*s，从而将图*G*(*A*) = (*V*, *E*) 分割为近似相同的子图，最终将获得目标k路划分。如果K是2的幂次方，那么获得顶点分割的自然方法是先获得改图的一个2路分割，它被乘坐关联边*E*s分割的一个图二分。计算出*E*s的一个**顶点分离项**Vs后，它就获得V \ Vs的一个2路划分V1,V2。重复这个处理过程，直到最终获得k=2^m路分割。对于这个矩阵A的顺序，V1的顶点排在第一，然后是V2,Vs。这个重排序过程和每个Vi的二分过程类似，通常， 较小尺寸的**顶点分离项**具有较低的填充。

### 顶点分离项

为了讲述顶点分离项，我们考虑图1中经过匹配处理的重排序后的矩阵ΠT *A，图4我们展示了一个图G*(ΠT *A*)*，它忽略边的方向。可以获得一个两路分割，V*1 = {1, 3}, *V*2 = {5, 6}，和一个顶点分割*V*s = {2, 4}。对应重排序矩阵ΠT *A*的行列顺序为1, 3, 5, 6, 2, 4。

**图4请看原文。**

计算一个图的递归二分计算通常都是非常耗时的，因此长采用多层的图二分法计算这个过程。基本结构很简单，多层方法包含三个计算：首先是一个粗化阶段，压缩整个图到为下一层，它的大小是上层的一半大小。这个粗粒度的图被压缩到几百个顶点的时候，进入第二阶段被称为二分，它是一个高质量的分割方法。然后，第三阶段反粗化，这个二分比成功地延伸到原图。

### 粗化阶段（粗粒度化？ 图像处理术语：下采样更好？）

粗化处理将*A* ∈ Rn,n的初始图*G*0 = (*V*0, *E*0) = *G*(*A*变换为一系列子图*G*1, *G*2, . . ., *G*m ，而且他们的缩小尺寸为|*V*0| |*V*1| |*V*2| · · · |*V*m|。给定图*G*i = (*V*i, *E*i),下一个粗化图*G*i+1 is 可以通过折叠相邻顶点获得。例如使用Gi最大匹配Mi,下一个粗化图*G*i+1 is可以通过折叠Gi的顶点为多个节点，例如Mi的元素可以和Gi没有匹配的元素变成Gi+1的新顶点Vi+1.新的边集合*E*i+1 是Ei和折叠顶点连接而成。有许多构造最大匹配的不同实现，参考 [25, 9]。其中应用最广而且最有效的方式是重边匹配 [25].

### 分割阶段

在粗化阶段，计算*G*m = (*V*m, *E*m)的一个二路分割*V*m,1∪*V*m,2 = *V*m, 它们其中任何一个包含大约Gm的一半顶点。Gm的这个分割可以有许多算法，例如光谱半分割 [19]， Kernighan-Lin [27, 18]. 相对于光谱法，后者通常计算更小的边切割分割项。

然而，由于粗化图Gm的尺寸通常很小(|*V*m| < 100)，这个步骤相对于整个计算时间可以忽略不计。

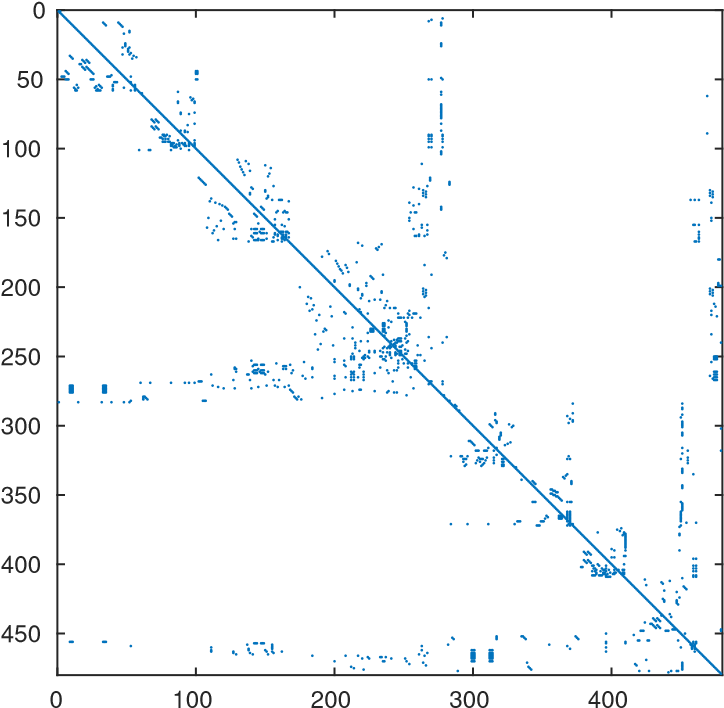
### 逆粗化阶段

假设粗化层*m*, Gmd的一个边分割项*E*m,s ，它关联一个计算好的二路分割，它产生Gm的一个足够好的边切割，使得Vm,1 和Vm,2具有几乎相同尺寸。 那么Em,s将被反向传播到Gm-1，使用折叠匹配顶点的逆过程。这样产生Gm-1的一个原始边分离项*E*m−1,s 。由于Gm-1是具有更好粒度，Em-1,s是半优化的，通常使用局部启发式犯法例Kernighan–Lin【27】、Fiduccia–Mattheyses【18】来减少分割中的边切割。继续向上层重复这个细化过程，我们得到一些列的边分割项*E*m,s, *E*m−1,s, . . ., *E*0,s ，那么最终获得原始图*G*(*A*)的边分割项*E*s = *E*0,s 。如果有人寻求G(A)的一个边分割项， 那么通常最终用Es计算Vs。

有大量的方法计算图分割，例如Metis [25],并行MPI版本ParMetis [26], 最近有一个多线程方法MT-Metis[30]. 还可以参考另外一个并行分割算法Scotch[9].

### 多层嵌套风格

我们继续例2中据矩阵*A*˜ =ΠT *D*r *AD*s t，它已经经过了最大权重法的缩放和置换。图5展示了一个使用多层嵌套风格后的图样*A*ˆ = *P*T *AP*˜，其中p指*G*(*A*˜)分割后关联的置换矩阵。

**Fig. 5 使用最大权重匹配并完成缩放于置换的矩阵被执行多层嵌套分割**

## 3.2 其他重排序方法

一个早期的重排序方法是逆Cuthill–McKee (Rcm) 法[10, 34]，它试图减少一个矩阵需要的带宽。尽管这个方法对于顺序解法和不完全分解法仍然很有吸引力， 对直接求解法而言已经过世。另外一个与嵌套分割法不同的重排序方法是**最小度**（MMD）【40，20】以及它的变种，在实践中**近似最小度算法**（AMD）【2，12】，该变种可以包含限制条件。**最小度**算法的主要目标是模式高斯消元过程，它研究图论方法的更新过程*a*ij → *a*ij − *a*ik*a*kk−1*a*kj ，该方法使用于非直接图。这里的**度**指一个顶点连接的边的数量，以及该图和**度**如何在分解过程中变化。过去几年，最小度算法的不断革新，使用**外度**来选择要旋转的顶点，包括最新的就是例如**不完全度**更新，元素吸收，多消除，以及新的数据结构变化。【20】给出这类算法的总结。最小度算法最大的进展就是更新度的计算。不是计算准确的**外度**，AMD[2]给出了近似计算外度的方法，在LU分解的填充过程能够显著地节省计算。

我们现在来对本章做个总结，我们讲到如果*G*(*A*) = (*V*, *E*)的嵌套分割计算产生一个顶点分离项Vs和剩余的顶点*V* \*V*s对应的一个k路分割,V1…Vk,，那么可以继续宁并行计算，对于人一个Vi中的元素可以以任意顺序排列。或者说Vi中也可以使用嵌套分割，这也是嵌套分割可以进行多层分割的精华。然而， 一旦粗化图Gm足够小（通常小于100个顶点），不仅能够计算分割器，并且Gm的剩余元素也被排序为一个简化填充的顺序。对于这些情况，Gm和V1…Vk都可以使用不同的排序方法， 例如MD的变种算法。事实上，Metis软件就是这么处理Gm的。此外，受限的AMD算法或者类似的排序算法也可以用作V1…Vk的局部排序算法，以替代嵌套分割，可以考虑连接到顶点分离项Vs的边集合(参考HALO结构)，请参考文章[38]。

# 4 稀疏LU分解

本章我们假设一个给定矩阵是*A* ∈ Rn,n 是非奇异的，这样它可以被分解为*A* = *LU*，其中L是下三角对角矩阵，U是上三角矩阵。参考著名文章 [21], 如果 *A* = *LU*, 其中*L* 和 *U*> 是下三角矩阵， 通常我们有*G*d(*L*+*U*) ⊃ *G*d(*A*),也就是说我们仅仅得到一些多余的边除非有些元素在消除过程中被意外删除。通常我们忽略这些意外删除。在本节，我们会经常假设A的对角元素也是非0。我们假设*G*d(*A*) 是联通的。

前面章节我们讲到最大权重匹配会产生一个经过缩放和置换的矩阵，能够产生静态的旋转，或者说旋转可以被限制在LU分解的稠密块。此外，重排序车裂例如多层嵌套分割能够进一步对称地置换矩阵， 使得高斯消元过程的填充减少到一个可忍受水平，并且这个重排序可以并行执行。 本节假设A不需要重排序，而且A=LU是需要解决的真实问题。

## 4.1 消元树

三角分解L和U的填充的基本方法是高斯消元，归纳如下，更详细的可以参考[23]。

**定理2** *设A* = *LU符合前面章节的假设，存在一个边* (*i*, *j*) *属于 G*d(*L* + *U*)，当且仅当存在*G*d(*A*)的一个路径

*ix*1, *x*2*x*3, . . ., *x*k *j*

*使得x*1, . . ., *x*k < min(*i*, *j*)*.*

换言之，高斯消元法的过程，对每一个从i到j满足的路径，如果它的顶点小于min(*i*, *j*)我们都获得一个填充边(*i*, *j*) 。

### 填充

我们使用例3.1的矩阵ΠT *A* ，把它高斯消元中获得的填充扩张矩阵参考原文Figure 6.

高斯消元过程的常见问题是如何预测和快速计算填充图*G*d(*L* + *U*) 。如果图是无向的，那么问题将变得简单。对于无向图，我们可以不考虑边的方向，可以简单考虑无向图 *G*(*A*) 和*G*(*L* + *U*)。

**定义11** 对无向图*G*(*A*)使用定理2可以获得无向图*G*(*L* + *U*) ，可称为填充图*G*f (*A*).

### 无向图的填充

当我们考虑例子4.1的无向图*G*(*A*), 数据模板of |ΠT *A*| + |ΠT *A*|T ，它的填充图*G*f (*A*)也等价于*G*(*A*) (cf. Figure 7).

无向图的预测填充图的关键方法是消除树[33].

一个无向连接图被称为树，当它不存在任何一个。此外，有一个顶点被定义为根顶点。通常，如果存在一个边(i,j), 称一个顶点j的父顶点，那么j更靠近根节点。，我们称i为j的子节点。从顶点j开始的子树被标记为*T*(*j*) ，这个子树的顶点被称为j的后代，j是 他们的祖先。最初我们使用深度优先算法定义消元树[1]. 最后我们描述一个简单算法。定义 **12 的填充图***G*f (*A*) 的消元树*T*(*A*)可以用下述算法构造。

从顶点n对*G*f (*A*)执行一个深度优先搜索算。

如果已经访问了顶点m, 从它的未访问的邻居*i*1, . . .,*i*k 选取*j* ，其中最大值*j* = max{*i*1, . . .,*i*k} ，然后继续用j搜索. 当所有邻居都被访问，那么我们获得一个树的叶节点。

再次，我们特别指出，使用对图*G*f (*A*)的深度优先算法和其他图的表现玩完全不同。通过定理2，每当我们访问顶点m, 所有邻居j > m必须提前于m被访问。因此，当我们到达一个叶节点，顶点是被降序排列。

### 深度优先搜索

我们使用例子4.1来展示填充图的深度优先算法。参考图8，多余的边用粗线标记。

深度优先算法访问顶点顺序，6 → 5 → 4，当4的所有邻居都被访问，算法回溯到顶点5，继续用5 → 1的顺序访问，定义1的所有邻居被访问，算法返回顶点5，然后回到顶点6，继续按照顺序6 → 2。这时算法终止。由于顶点3是独立的，因此图A是不连接的，算法必须对每一个连接子图做单独处理。

1

5

2

6

4

3

6

1

5

3

4

2

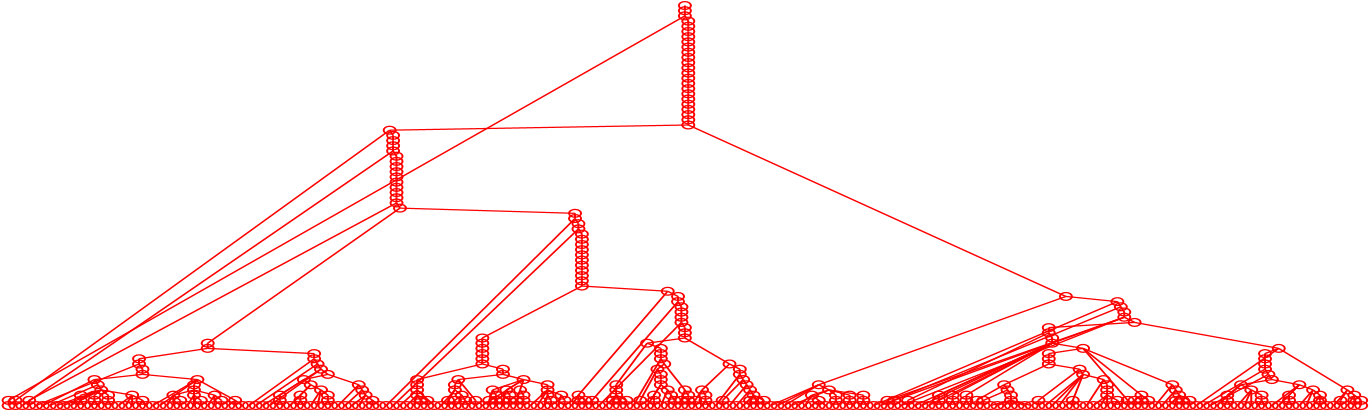
**Fig. 8** Filled graph (left) and elimination tree(s) (right).

*推论1*树*T*(*A*)的构造，和定理2中Gf (A)的边，如果它们不能被校覆盖，他们只能是一个点或者他们的祖先。换种说法，不存在不相连顶点的跨接边。

*推论2* 根据推论1，三角分解可以从加节点开始独立计算，直到他们相同的父辈节点，也就是说L的列j ，和*U*T 仅仅依靠这些列s，如果*s* of *L* and *U*T 在消元树*T*(*A*)中是j和后代。

### 消元树

我们使用例子3.1矩阵 “west0479”, 它已经经过最大权重匹配和多层嵌套分割，我们使用Matlab的etreeplot来展示消元树 (see Figure 9).经过嵌套分割后，消除树具有高度的兵法性，根据推理2，每个叶节点都可以独立计算直到一个相同的父节点。



**Fig. 9** Elimination tree of “west0479” after maximum weight matching and nested dissection are applied.

*推论3* 设 *k* ∈ {1, . . ., *n*}. T那么存在一个填充边 (*j*, *k*)满足*j* < *k，* 当且仅当存在一个相同的后代i,使得*a*ik , 0. This follows from the fact that once *a*ik ≠0, 根据定理2，这个填充边(*j*, *k*) 位于消除树i,k之间，或者说所有i的祖先都是k的后代。那么，i可以把填充边信息沿着i到k的树干传播，信息*a*ik ≠0,也可以做路径压缩信息。

### 路径压缩

考虑一个图和它的消元树，例如图8. 由于图G(A)有一个边（1，6），同时另一个（填充）边(5,6)也必须存在，相似地，也必须存在边（4，6），由于a1,6 !=0,那么我们可以绕过5 而直接子节点1到节点6。

消除树可以比较容易地使用一个长度为n的向量p来表达， 使得对于任何i< n, Pi标记为它的父节点，Pn=0对应于根节点。对于一个步长k,对于i< k如果aik !=0,根据推论3，i必须是K的后代，而且有另外一个后代j,也是k的后代。也许不是所有i的祖先被指派，我们可以替换i使用j=Pi直到Pi=0或者Pi>K。这样我们从i到K遍历T(A)直到我们发现k的子节点j。如果J的父节点没有指派，那么Pi=0和K就是J的父节点。如果i < k 是J的父节点，我们可以指派I是J的父节点。简言之，设置*p*j ← *k* 否则如果Pi > K, 我们已经设置J的父节点在更早的一步 I < K 。

### 计算父节点

考虑图8的消元树T(A),除非是K=5, 没有父节点被设置，对于所有i都有Pi=0。

对于K=5, 我们有ai,5!=0, 而且P1=0,我们设置*p*1 ← 5，对于a4,5!=0,P4=0，*p*4 ← 5。

最后对于K=6, a1,6~=0,此前我们已知p1=5,由于P5 = 0, 因此我们设置 *p*5 ← 6，下一个a2,6 !=0,p2=0,我们设置 *p*2 ← 6。A4,6 != 0,从4遍历到p4=5, p5=6,不需要更多操作。A5,6 也不需要更一步操作，因为我们已经有p5=6。

最终我们有*p* = [5, 6, 0, 5, 6, 0] ，它完美地揭示了图8的消元树特性。

从推论3， (cf. [43, 12]), 我们也可以充分使用路径压缩。 因为我们的目标是最快遍历从i到k的消除树，i的 任何祖先节点j = ai都可以。我们已经说过，任何一个祖先节点ai=0指一个没有父节点的顶点。这时可以设置I *p*j ← *k*. 而且*k* 总是ai的一个父节点。

关于路径压缩的算法总结参考 (see also [33, 12]).

### 计算消除树

**Input:** *A* ∈ Rn,n such that *A* has the same pattern as |*A*| + |*A*|T.

**Output:** vector *p* ∈ Rn such that *p*i is the parent of *i*, *i* = 1, . . ., *n*−1, except *p*n = 0. 1: let *a* ∈ Rn be an auxiliary vector used for path compression.

2: *p* ← 0, *a* ← 0

3: **for** *k* = 2, . . ., *n* **do**

4: **for** all *i* < *k* such that *a*ik , 0 **do**

### 5: while *i* , 0 and *i* < *k* do

6: *j* ← *a*i

7: *a*i ← *k*

8: **if** *j* = 0 **then**

9: *p*i ← *k*

10: **end if**

11: *i* ← *j*

### 12: end while

### 13: end for

### 14: end for

## 4.2 超节点方法

我们已经看到消元树展示了并行性。L，U填充可以充分这一信息。决定是否进行填充的信息也来源于推论3。假定我们对L和UT的矩阵j的非0元素感性其。那么对J的所有后代，或者说从顶点j为根节点的子树T(J)的所有节点，一个非0元素 *a*ik , 也意味着 *l*kj  !=0. 因此，从任何一个也节点I,我们获得它的所有填充*a*ik ！=0 ，使得所有K > I , 当我们从I遍历到它的父节点J,顶点J将继承从节点I的填充，当且仅当k> j，而且Ai,k非0.当我们到达一个多子节点的共同父节点，我们可以用同样的填充方法。我们总结如下简单方：

### 计算填充

**Input:** *A* ∈ Rn,n such that *A* has the same pattern as |*A*| + |*A*|T.

**Output:** sparse strict lower triangular pattern *P* ∈ Rn,n with same pattern as *L*, *U*T.

1: compute parent array *p* of the elimination tree *T*(*A*)

2: **for** *j* = 1, . . ., *n* **do**

3: supplement nonzeros of column *j* of *P* with all *i* > *j* such that *a*ij , 0

4: *k* = *p*j

### 5: if *k* > 0 then

6: supplement nonzeros of column *k* of *P* with nonzeros of column *j* of *P* greater than *p*

### 7: end if

### 8: end for

算法4.2 仅仅处理填充图案。另外它能够提效能和显著地加速数值分解，能检测分解中的密集矩阵。块结构允许组合矩阵部分元素为密集子矩阵，从而可以使用密集矩阵的核心函数，例如第三级 BLAS and LAPACK [13, 14].

密集块可以从算法4.2的消除树读取。

**定于13 定义Pj为P的列j用算法4.2计算的非0索引，一个序列***k*, *k* + 1, . . ., *k* + *s* − 1 被称为大小为S的节点， 如果Pj = Pj+1 ∪ {*j* + 1} for all *j* = *k*, . . ., *k* + *s* − 2.

简而言之，定义13指出对于一个超节点，它们的列可以被组合为一个密集块，一个密集三角对角块，因为他们完美组成一个梯形形状。我们可以对算法4.2做扩展以检测一个超节点。

### 填充和超节点的计算

**Input:** *A* ∈ Rn,n such that *A* has the same pattern as |*A*| + |*A*|T.

**Output:** sparse strict lower triangular pattern *P* ∈ Rn,n with same pattern as *L*, *U*T as well as column size *s* ∈ Rm of each supernode.

1: compute parent array *p* of the elimination tree *T*(*A*)

2: *m* ← 0

3: **for** *j* = 1, . . ., *n* **do**

4: supplement nonzeros of column *j* of *P* with all *i* > *j* such that *a*ij , 0

5: denote by *r* the number of entries in column *j* of *P*

6: **if** *j* > 1 and *j* = *p*j−1 and *s*m + *r* = *l* **then**

7: *s*m ← *s*m + 1 . continue current supernode

### 8: else

9: *m* ← *m* + 1, *s*m ← 1, *l* ← *r* . start new supernode

10: **end if** 11: *k* = *p*j

### 12: if *k* > 0 then

13: supplement nonzeros of column *k* of *P* with nonzeros of column *j* of *P* greater than *p*

### 14: end if

### 15: end for

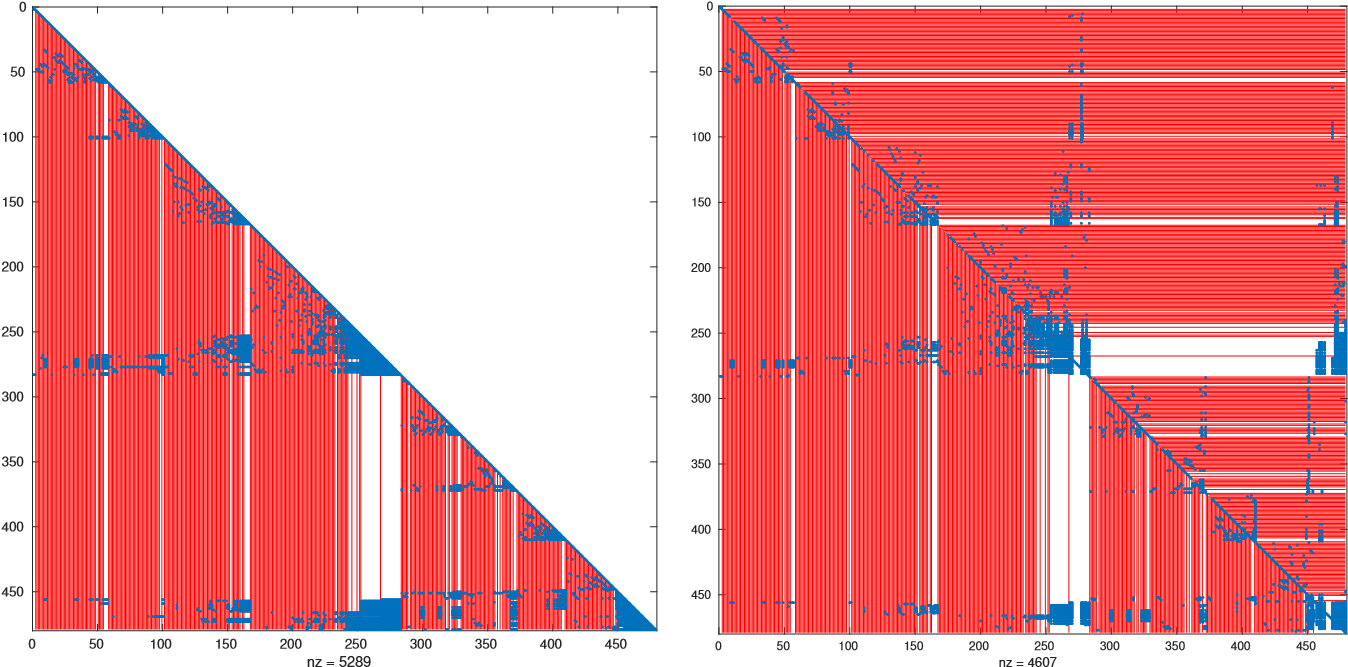
### 超节点计算

为了图示超节点，我们考虑图7的矩阵图案，用图10展示对应的密集块结。列1，2，3对应为标量列， 列4-6对应为一个单一超节点。

超阶段称为加速计算的重要基础，例如超节点可以被存储为一个或者多个密集矩阵。除了存储为密集矩阵外，这些块的非0行索引只需要存储一次。此外这些稠密子矩阵可以使用BLAS求解 [13, 14].

### 超节点

我们使用例子3.1的矩阵 “west0479” ，经过最大权重匹配和多层嵌套分割 。我们它的无向图计算超节点结构。当然，因为块结构是非对称的，那么块结构是近似优化的。我们展示 Cholesky 分解对应的块结构，因为Cholesky 分解对称正定矩阵使用同样的无向图，参考 (see left part of Figure 11). 此外， 我们展示了L和U的超节点结构，而且不是用旋转 (see right part of Figure 11).



**Fig. 11** Supernodal structure. Left: vertical lines display the blocking of the supernodes with respect to the associated Cholesky factor. Right: vertical and horizontal lines display the blocking of the supernodes applied to L and U.

构造对称矩阵的超节点非常简单，而其他情况则异常困难。因为我们必须处理高斯消元中的每一步旋转。请参考列消元树 [22].

# 5 稀疏矩阵直接求解器— 超节点数据结构

高性能稀疏矩阵求解数学函数库对于科学和工程非重要，而且随着计算机微结构的发展， 有些微处理器体结构变得更加复杂，数学库需要变得更加容易被集成到应用程序中（原始英文好别扭）。尽管有需许多的科学于工程应用，但是基础支撑算法具有高度相似性，特别是这些算法需要完美并行执行。我们不是说计算科学的这些高性能计算的成功都来源于这些软件库（那作者你在说啥？）。本节我们将使用【42】的PARDISO稀疏矩阵求解软件包展示超节点数据结构的好处。我们将使用三角阵处理过程。列L的前向与后向替换使用列方向，参考图12 .

数据依赖性允许存储向量y, *z*, *b*, 和*x* 在一个向量r. 当遇到向量j, *r*j包含yj的解。L所有元素， 或者用*L*ij 表示，其中 *i* = *j* +1, . . ., *N*, 被用来更新r的其余元素，参考公式（8）

*r*i = *r*i − *r*j*L*ij. (8)

*L*T 的后向替换发生在行方向。相对于列的前向替换，它将从最后一个列开始循环到第一列。到达列J的时候， 计算*r*j, 就可使用Lij的元素机型计算

*r*j = *r*j − *r*i*L*ij. (9)

当所有的列都被处理了，那么r就包含x需要的解。图12的第5行代表了两个操作，索引的DAXPY和索引的DDOT核函数，他们需要在计算向量r的流操作和L的列j数值分解过程中计算。当我们处理稀疏矩阵是，把下三角阵保存为密集阵并无意义。那么PARDISO 使用自己的数据结构保存*L*, 参考Figure 13.

1: **procedure** Sparse forward substitution

j

j+1

j-1

L

r

iteration direction

j

j+1

j-1

r

L

L

T

2: **for** j = 0; j < n; j++ **do**

3: **for** i = xl[j]; i < xl[j+1]; i++ **do**

4: row = id[i]

5: r[row] -= r[j] \* l[i] . indexed DAXPY

6: **end for**

7: **end for**

8: **end procedure**

1: **procedure** Sparse backward substitution

2: **for** j = n; j > 0; j - - **do**

3: **for** i = xl[j]; i < xl[j+1]; i++ **do**

4: row = id[i]

5: r[j] -= r[row] \* l[i] . indexed DDOT

6: **end for**

7: **end for**

iteration direction 8: **end procedure**

**Fig. 12** Sparse triangular substitution in CSC format based on indexed DAXPY/DDOT kernel operations.

panels

separator

parts

t

r

L

**Fig. 13** Sparse matrix data structures in PARDISO. Adjacent columns of L exhibiting the same structure form panels also known as supernodes. Groups of panels which touch independent elements of the right hand side r are parts. The last part in the lower triangular matrix L is called separator.

一个面板内相邻的列表现出相邻的稀疏结构，或者说超节点。一个面板的列数被称为面板尺寸Np. 面板的尺寸被一次保存在内存中，但不包括0元素。一个面板的列都被用0在前面填充，这样他们获得和该面板的第一列同样的长度。PARDISO使用这种填充采用调用BLAS和LAPACK获得最大性能。此外，面板被依次保存到一个数组I. 行和列信息被存储到一个伴生数组中。数组xsuper 保存每个面板的第一个列索引。同时请注意这个列索引是非0列的运行数。列索引被用在查找在数组I的第一列的位置。为了计算一个列元素对应行索引，我们需要使用另外一个数组，这个数据包含每个面板对应的行索引。数组xi对应面板的ID.一个面板的第一行索引被记做id[xid[p]]. 上述数据结构可以满足顺序执行所需信心。但是， 对于并行执行前向和后向替换，必须设法避免并行更新r的同一个元素。Parts数据结构包含了面板的开始、结束索引，它能够独立地更新而不会处理r的相同元素。请看，图13的两个部分，蓝色和橙色。最后一个部分，L的最右下角被称为分离，标注为绿色。

分离奔赴的处理将他们的结果写入临时数组t。在分离部分被处理前，临时部分数组可以被集中返回到r.后向处理和前向相同， 在Pardiso中是算法1的逆过程。注意，实际上也可以把临时数组的顺序处理(10-13)使用6-9的循环处理。

算法 1

1: **procedure** Forward

|  |  |  |
| --- | --- | --- |
| 2: | **for** part o in parts **do** | . parallel execution |
| 3: | **for** panel p in part p **do** |  |
| 4: | **for** column j in panel **do** | . unroll |
| 5: | i = xid[p] + offset |  |
| 6: | **for** k = xl[j] + offset; k < sep; ++k **do** |  |
| 7: | row = id[i++] |  |
| 8: | r[row] - = r[j] l[k] | . indexed DAXPY |
| 9: | **end for** |  |
| 10: | **for** k = sep + 1; k < xl[j+1]; ++k **do** |  |
| 11: | row = id[i++] |  |
| 12: | t[row,p] -= r[j] l[k] | . indexed DAXPY |
| 13: | **end for** |  |
| 14: | **end for** |  |
| 15: | **end for** |  |
| 16: | **end for** |  |
| 17: | r[i] = r[i] - sum(t[i,:]) | . gather temporary arrays |
| 18: | **for** panel p in separator **do** | . serial execution |
| 19: | **for** column j in panel **do** | . unroll |
| 20: | i = xid[p] + offset |  |
| 21: | **for** k = xl[j] + offset; k < xl[j+1]; ++k **do** |  |
| 22: | row = id[i++] |  |
| 23: | r[row] -= r[j] l[k] | . indexed DAXPY |
| 24: | **end for** |  |
| 25: | **end for** |  |
| 26: | **end for** |  |

27: **end procedure**

**Algorithm2** Backward substitution in PARDISO. Separator (sep.), parts, and panels are iterated over in reversed (rev.) order.

1: **procedure** Backward

2: **for** panel p in sep. rev. **do** . serial execution

3: **for** col. j in panel p rev. **do** . unroll

4: i = xid[p] + offset

5: **for** k = xl[j] + offset; k < xl[j+1]; ++k **do**

6: row = id[i++]

7: r[j] -= r[row] l[k] . indexed DDOT

8: **end for**

9: offset = offset - 1

10: **end for**

11: **end for**

12: **for** part in parts **do** . parallel execution

13: **for** panel p in part rev. **do**

14: **for** col. j in panel p rev. **do** . unroll

15: i = xid[p] + offset

16: **for** k = xl[j] + offset; k < xl[j+1]; ++k **do**

17: row = id[i++]

18: r[j] -= r[row] l[k] . indexed DDOT

19: **end for**

20: offset = offset - 1

21: **end for**

22: **end for**

23: **end for**

24: **end procedure**

# 6 应用-电路仿真

本节我们将演示稀疏矩阵求解器如何帮助集成线路仿真。集成线路包含互联的晶体管，主要有电阻，电容和电感器。这些互连通过电路传递信号和提供能量。电路方程源自Kirchhoff 电流原理，对一个节点使用电路方程， 那么就成为一个微分方程组。在电路的时间尺度进行模拟，可以用电路的隐式积分的时间离散求解微分。非线性方程可用[拟牛顿法](https://baike.baidu.com/item/%E6%8B%9F%E7%89%9B%E9%A1%BF%E6%B3%95/9340944) 或者连续法求解，那么可以将非线性问题变为一些列的线性方法求解。电路中的每个组件都会变成几个方程。因此整个线性方程非常稀疏，非常适合直接稀疏矩阵技术。参考【11】，电路仿真的稀疏矩阵包含一下特性：

* 不对称，近似结构对称；
* 通常有几个密集行和列，例如电源或者接地；
* 非常稀疏，直接使用BLAS库效率很低， (as in SuperLU[32])
* LU分解也很稀疏，即使具有很好的顺序
* 使用通常的重排序策略具有高填充；
* 无结构，具有不规则的内存读写，而导致分解只能在局部获得很高的峰值计算能力；

电路仿真矩阵可能是正定的，也可能是非常病态的，需要使用旋转保证稳定的收敛。当电路规模增加，依据连接的规模建模，稀疏矩阵分解是瞬态分析最重要的计算开销。

为了减低矩阵分析的复杂性，1990s出现了一种新的模拟方法，被称为快速SPICE[39]。这个模拟器把电路划分为许多子电路，以减小矩阵规模。但是，这种模拟方法会导致模拟电路和紧耦合部件带来不可接受的错误。随着精度要求提升，这些方法变得比传统SPICE方法更慢。尽管如此，还是有许多研究如何用快速SPICE模拟器，使用快速计算而能产生某种可接受的精度。这些研究不需要模拟整个系统而且能在限定时间完成模拟，他们并没推进稀疏矩阵的发展。

从2000s，随着对精度需求提升，特别是现先进导体技术，迫使模拟器回归传统SPICE.多核CPU的发展帮助很大。 并行电路模拟，此前并不流行。随着新技术发展，减少缓存丢失，重写并行计算的代码，新的稀疏据矩阵求解器， KLU[11]成为SPICE工具库最主要的加速依靠。

为使用稀疏矩阵模拟大规模电路仿真，而且完全具有SPICE精度，瞬态分析的稀疏矩阵软件包应该具备以下特性：

* 必须支持并行计算must be parallel;
* 快速矩阵重排序；
* 只需要少量的非0变换就可以做LU分解的渐进更新
* 快速求解逆矩阵的对角元素
* 快速求解一个子集的Schur-complements
* 允许使用相同的存储结构对多个LU分解；
* 允许使用迭代？
* 可以运行子多种不同硬件平台

后面章节停止翻译。