Weekly Research Progress Report

Student: 丛兴 **Date:** 10/11/2023-10/25/2023

List of accomplishments this week:

学习 SuperLU 的基本实现方法和思想

一种在 SuperLU 中进行 LU 分解时，减少通信的 3D 算法

Paper summary:

Name： A Communication-Avoiding 3D LU Factorization Algorithm for Sparse Matrices

Motivation：利用高斯消元（LU 分解）直接求解线性方程 Ax=b，当系数矩阵 A 大到一定的规模并且是稀疏矩阵时，由于矩阵 A 的稀疏特性不同，而引起的复杂的数据依赖性、不规则的访存机制和高动态算术强度使得对线性方程的求解较为困难。而且与稠密矩阵相比，直接求解器中的通信即使在核数相对较少的情况下也能迅速占据主导地位。但是，在稠密矩阵领域已经有人通过冗余计算、数据复制的方式来减少数据通信，而在稀疏矩阵的求解领域，却没有人涉猎，同时，Super LU 前期的研究加快计算的方式，也仅仅是通过重叠计算的方式来从表面现象上减少通信时间，并没有实际上减少通信量。因此本文以 Super LU 为 baseline 来重新设计算法，来实际减少在进行 LU 分解时的通信量。

Solution：借鉴 2.5D dense LU 算法的思想，将二维进程 grid 映射到三维 block 上面，本方法在进行映射时，借助 eliminate-tree(e-tree)的结构进行映射，将各个子树上所需要的父节点数据进行拷贝到每一个层上面，并行计算每个层的 schur-complement，最后将计算好的所有子树进行通信，合并到根节点上，也就是说，利用空间换时间的方式，实现通信量的减少，因此每层子树所需要的数据都已经拷贝一份了。

Related to us：了解对稀疏矩阵方程 Ax=b 的直接求解器算法，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： Nested Dissection: A survey and comparison of various nested dissection algorithms

Motivation：对嵌套剖分领域的一些工作进行了调查，并试图用一个共同的框架将其整合在一起。

Solution：各种版本的嵌套剖分方法被证明是 e-tree 的不同形式的树遍历算法。

Related to us：对第一篇文章所涉及到的 *nested dissection* 进行理解和学习。

Name： Graph Fill-In, Elimination Ordering, Nested Dissection and Contraction Hierarchies

Related to us：对第一篇文章中所涉及的 e-tree 和 *nested dissection* 概念进行学习。

Name：New Scheduling Strategies and Hybrid Programming for a Parallel Right-looking Sparse LU Factorization Algorithm on Multicore Cluster Systems

Motivation：在现代高性能计算机的 NUMA 节点架构上，由于并行编程存在高度不规则的内存访问模式、任务和数据的高度依赖性以及数据分布和工作量的不平衡的问题，因此实现一种在时间和内存上都可以扩展的并行 LU 分解算法是很困难的。因此，本篇文章提出两种策略来解决并行因式分解算法的一些可扩展性问题。

Solution：在算法层面，使得可以并行进行的任务尽快执行；在编程和体系结构层面，将轻量的 OpenMP 线程引入到 MPI 进程中。

Related to us：了解第一篇文章中所提到的 look-ahead 名词概念，同时补充对 SuperLU 基本执行过程的认识。

Name：A Sparse Direct Solver for Distributed Memory Xeon Phi-accelerated Systems

Motivation：想要设计一种适用于混合多核 CPU 和 Intel Xeon Phi 协处理器组成的分布式内存系统的稀疏直接求解器。而直接求解的核心部分是 LU 分解部分，而 LU 分解中通信又是最需要解决的问题，因此，急需要提出一种算法，用来隐藏或者减少该过程中的通信。

Solution：将异步与加速卸载、懒更新和数据阴影相结合的方式，构建出一种新的算法，用来隐藏过程中的通信，从而实现加速计算。

Related to us：查看 SuperLU 中 Panel 分解和 Schur-complement 更新的具体含义。

Name：Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms

Motivation：并行化的目标包括尽量减少通信、平衡工作负载和减少内存占用。有些问题可以通过在每个处理器上复制整个输入来实现并行化。然而，这种方法使用的内存可能远远超出需要，而且需要大量冗余计算。因为，一些并行算法确实成功地利用了有限的

额外内存来提高并行性或减少通信量，而在平衡线性代数问题中内存使用和通信成本中，并没有提出新的算法。

Solution：提出一种新的 2.5D 算法，使用可用的额外内存来将它们执行的通信量减少到理论上的最小值。

Related to us：对第一篇文章，理解所谓的 3D 算法。

Work summary

对于 Ax=b 线性方程的求解来说，除了使用迭代求解的方法来进行求解之外，还存在着直接求解的方式，比如基于高斯消元思想的直接求解器方式。而如果想要快速求解出 Ax=b 方程的解，可以采用 LU 分解的方法来实现分步求解，如下所示。

$$\begin{aligned} Ax &= b \\ LUx &= b \\ \Rightarrow x &= (LU)^{-1}b \\ &= U^{-1}L^{-1}b \end{aligned}$$

转化成两个方程组：

$$\begin{aligned} Ux = L^{-1}b \quad if \quad y = L^{-1}b \quad to \\ Ux = y \\ Ly = b \end{aligned}$$

其中，L 为下三角矩阵，U 为上三角矩阵。而直接求解下三角或者上三角的线性方程是很容易的。所以主要的求解问题过程就在如何将一个矩阵快速的分解为 LU 这两个矩阵的乘积。

而 Super LU 算法库就是用来将 A 矩阵分解为 L 矩阵和 U 矩阵的比较好的算法。

SuperLU 算法分为三个主要的步骤：

1. 矩阵 A 的预处理：

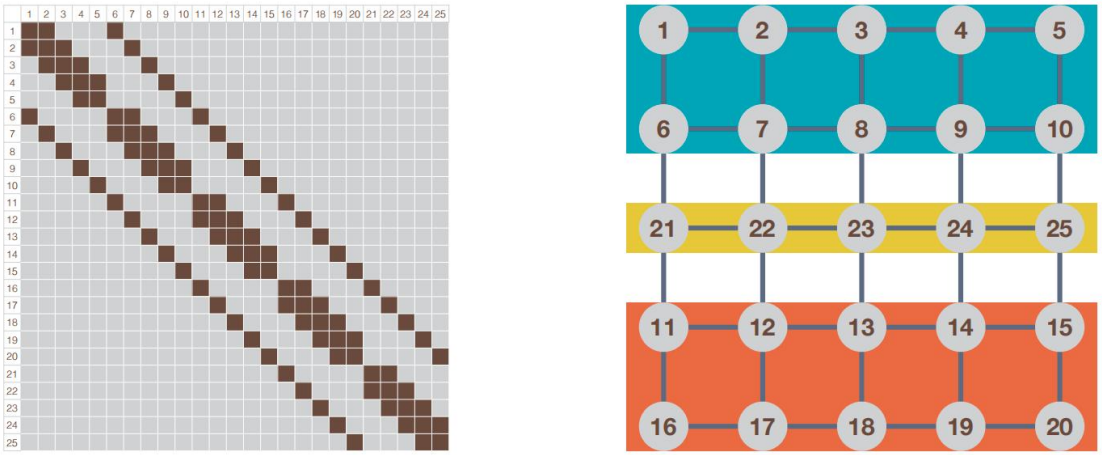
（1）通过静态旋转和矩阵平衡来提高数值稳定性：

$$P_r D_r A D_c$$

通过上述公式来使得矩阵 A 具有以下特点：非零对角线项的绝对值为 1，并且所有非对角线项的绝对值小于或等于 1。研究表明，这些预处理技术使 LU 分解在数值上与使用部分枢轴分解一样稳定，适用于广泛的问题。

(2) 对矩阵$P_r D_r A D_c$进行重新排序，使其 LU 因子保持稀疏，从而减少 *fill-in* 现象的产生，同时，重新排序还有助于减少通信，改善数值分解的负载平衡。可以使用的方法是 *Nested Dissection* 。

该方法的预处理过程例子如下：首先，将稀疏矩阵中非零点的关系转变为关联矩阵所表示的图。



(a) A 25×25 sparse matrix　　(b) The associated graph and separator

图 1 矩阵 A 的非零元素邻接关系图

比如，1-2、1-6、2-1、2-3、2-7 等。分别对上述关系以图中的边的关系表示出来。然后，选取分隔符，分隔符的选取规则如下：

- Nested Dissection Method
  - George's scheme uses the fact that removal of $2k-1$ precisely vertices from a $k \times k$ square grid leaves four square grids, each roughly $\frac{k}{2} \times \frac{k}{2}$.

而又因为例子中的矩阵是一个 5-对角矩阵，因此，为了在将来更好的处理小矩阵，对小矩阵进行 LU 分解，故将每矩阵分为 5*5 的小矩阵。因此，这个 25*25 的矩阵以小矩阵为单位，分为 5*5 的矩阵块行和列。因此，根据上面的算法，可以对上述矩阵进行操作，如下图所示：

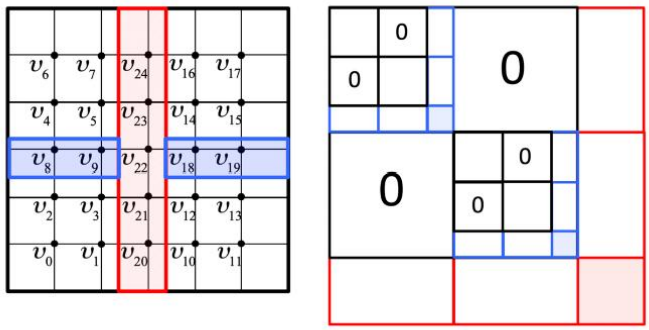
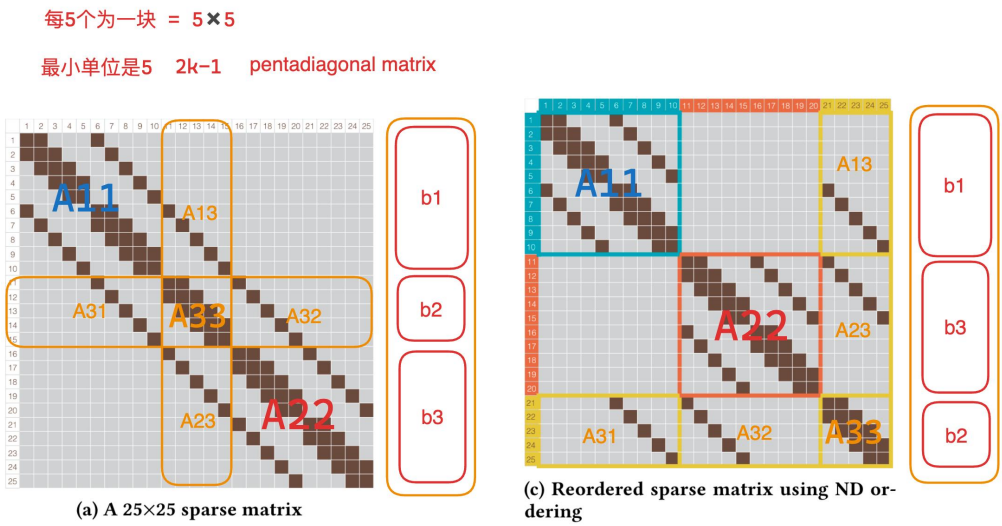


Figure 7.2.2.2 A second level of nested dissection.

图 2 Nested Dissection 示例



(a) A 25×25 sparse matrix

(c) Reordered sparse matrix using ND ordering

图 3 矩阵分块并进行 Nested Dissection 操作

因此，利用 A33 块可以将其分为三个大块，分别为 A11、A22、A33，其中 A11 和 A22 的 LU 分解将不会相互依赖，这就减少了依赖性，提高了并行性。上图所示的（c）为要进行分解的矩阵。

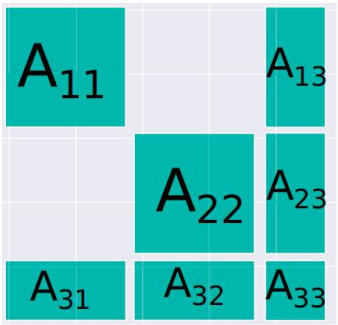


图 4 分解后的矩阵

2. 象征性分解:

静态枢轴相对于动态枢轴的主要好处是允许在数值分解之前先验地确定 LU 因子的稀疏性结构。提出了一种高效的符号分解算法，用于确定稀疏结构，建立所需的数据结构，并调度数值分解的所有通信和计算。

3. 数值分解（真正开始分解）：

在这个部分分为 3 个步骤：（这三个步骤的）

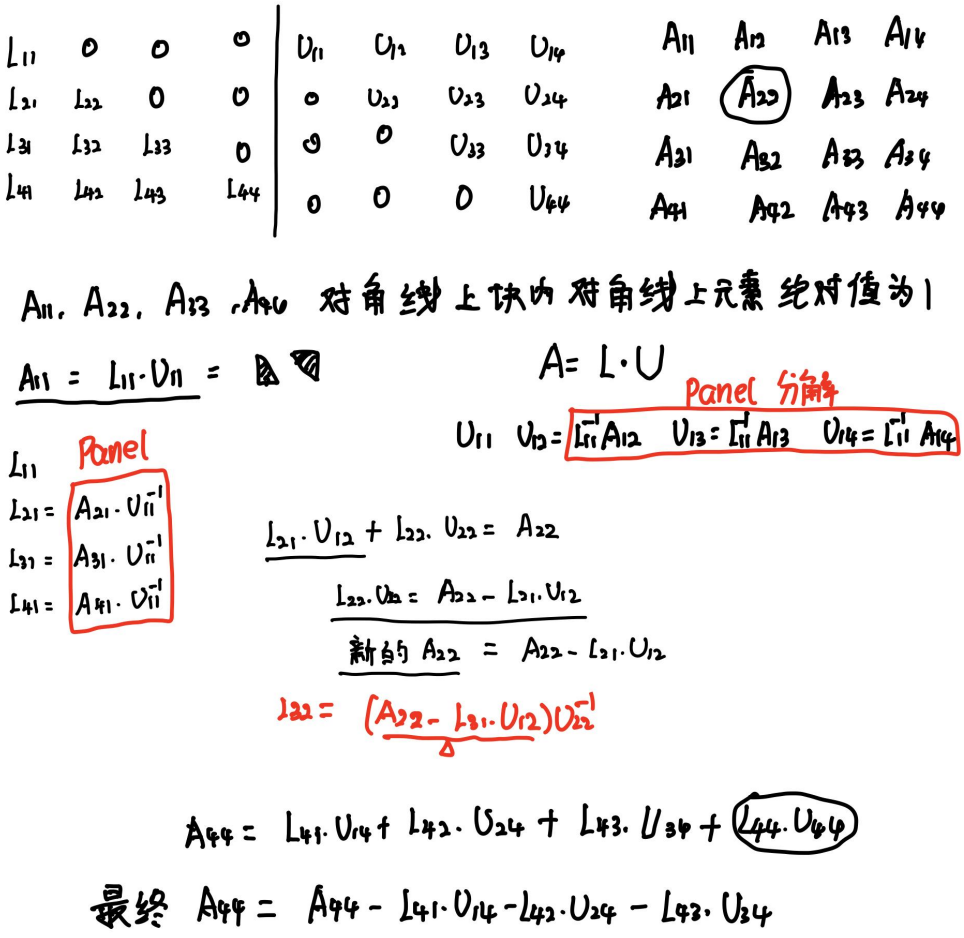


图 5 LU 更新过程原理分析

1) *Diagonal factorization*: $A_{ii} \rightarrow L_{ii}U_{ii}$
2) *Panel update*: $U_{ij} = L_{ii}^{-1}A_{ij}$ and $L_{ji} = A_{ji}U_{ii}^{-1}$
3) *Schur-complement update*: $A_{jk} = A_{jk} - L_{ji}U_{jk}$

图 6 LU 分解的主要步骤

而在分解时，确定分解的顺序是通过 etree 的遍历顺序来进行的，本篇文章采用的是从叶子节点到根节点的顺序进行顺序并行分解的。etree 的构建过程如下：

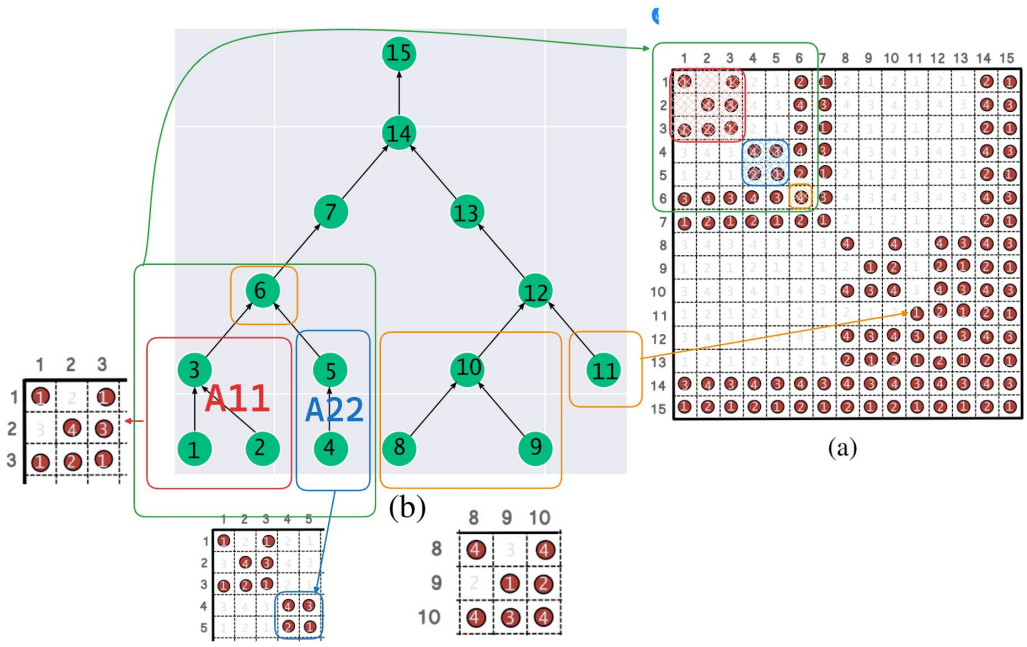


图 7 依赖关系图

所以，数值分解的过程就是按照上述三个步骤，首先对角线上的块进行分解，然后对该块正下方的列和该块正右方的行进行 panel 更新，然后再进行 schur-complement 更新操作。

而在分解过程中，主要的通信过程主要有四个部分，如下图所示：

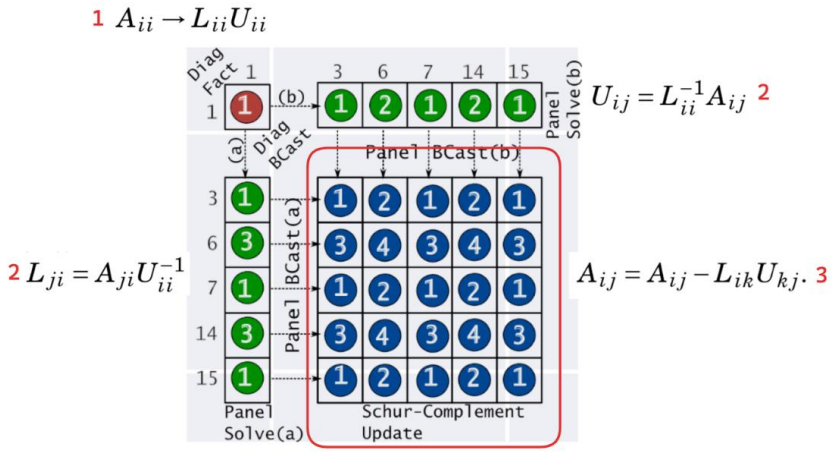


图 8 分解过程中通信的四个主要部分

在分解时，由于依赖关系的存在，所以可以将其过程视为流水线操作，具体流程图为：

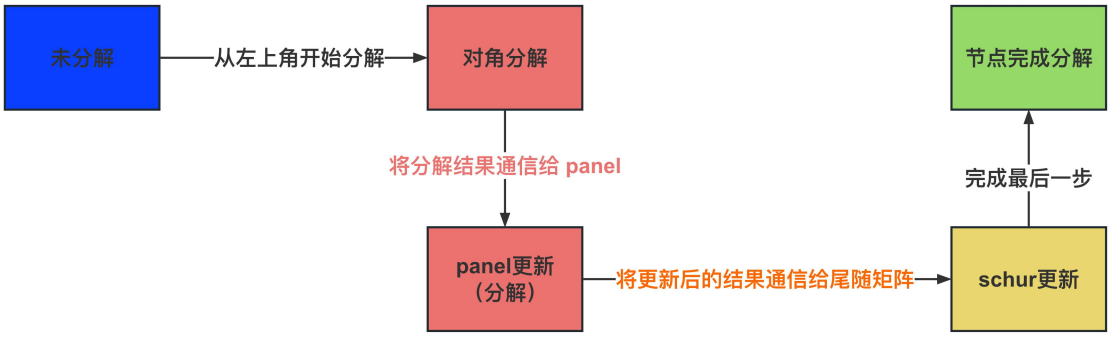


图 9 分解流程图

Algorithm 1 SUPERLU_DIST Sparse LU Factorization

Input: Distributed sparse matrix A; n_s: number of supernodes; p_{id} : my process rank; $P_r(k)$: k-th process row; $P_c(k)$: k-th process column.
On each MPI process p_{id} **do in parallel:**
for $k = 1,\ 2,\ 3 \ldots\ n_s$ **do**
  Synchronize all processes
  Panel Factorization
  if p_{id} owns $A(k,k)$ **then**
    Factor $A(k,k)$ and send $L(k,k)$ to $P_r(k)$ who need it
    Send $U(k,k)$ to $P_c(k)$
  if $p_{id} \in P_c(k)$ **then**
    Wait for $U(k,k)$
    Factor the block column $L(k)$
    Send $L(k)$ blocks to needed processes in $P_r(:)$
  else
    Receive $L(k)$ blocks if needed
  if $p_{id} \in P_r(k)$ **then**
    Wait for $L(k,k)$
    Compute the block row $U(k)$
    Send $U(k)$ blocks to required processes in $P_c(:)$
  else
    Receive $U(k)$ blocks if required
  Schur-complement update
  if $L(:,k)$ and $U(k,:)$ are locally non-empty **then**
    for $j = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
      for $i = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
        if $p_{id} \in P_r(i) \cap P_c(j)$ **then**
          $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$

图 10 LU 分解基本算法过程

所以，当第一个节点完成分解后，各个部分所处在流水线的阶段如下所示：

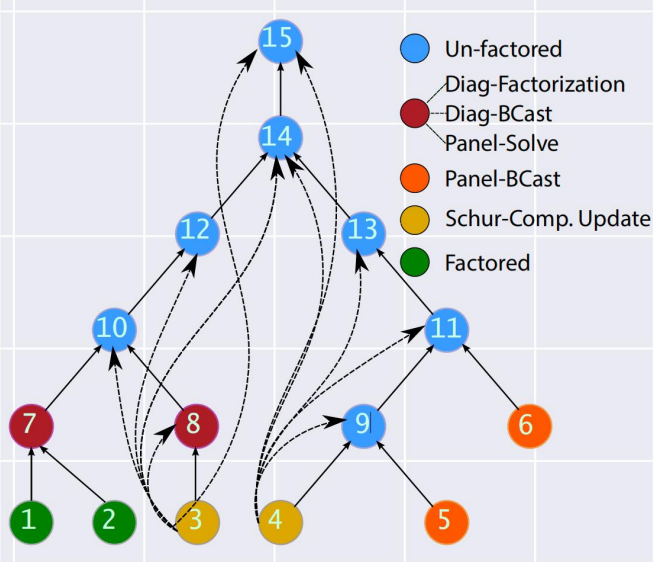


图 11 各个节点的处于流水线上的位置

所以，通过上述过程，可以看到 2D 的 LU 分解算法存在问题：2D 算法扩展到一定程度后，数据传输成本开始主导计算成本。而且，在大量的进程中，负载不平衡的影响变得更加突出。所以在一定数量的进程之后，添加更多的进程会导致分解时间变慢。从根本上说，二维算法存在以下两大局限性：

1. 顺序 Schur -complement 更新:

对于给定的块，在 2D 算法中只有一个进程可以执行 Schur -complement 更新。因此，尽管有丰富的树级平行级别，2D 算法必须按顺序执行所有 Schur 补码更新。

2. 固定的延迟成本

几乎所有进程都参与所有超级节点的分解。因此，各种通信内核的延迟并不会随着处理器数量的增加而减少。

所以，对于以上局限性，可以通过数据的拷贝，从而将 Schur-complement 更新分为多个部分，从而使得多层可以并行进行 Schur-complent 更新。

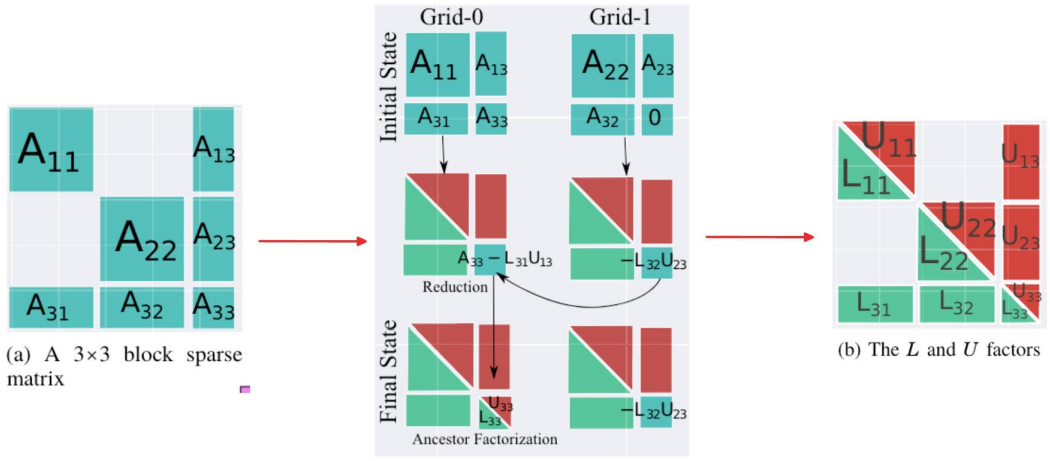


图 12 并行执行 Schur 更新示例

当矩阵更大时，可以增加分层，从而实现更多的并行性。

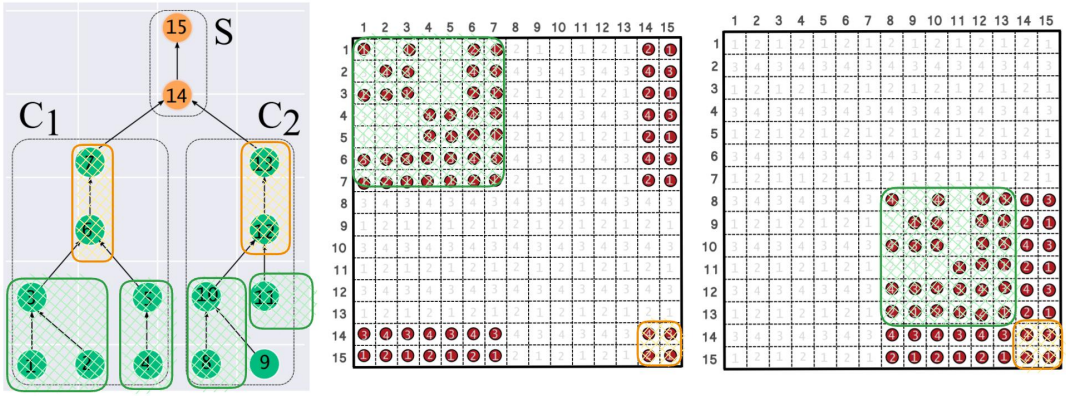


图 13 更多层的分层示例

同时，利用该分层的思想，也可以很容易实现，节点之间的负载均衡，比如下面的例子：

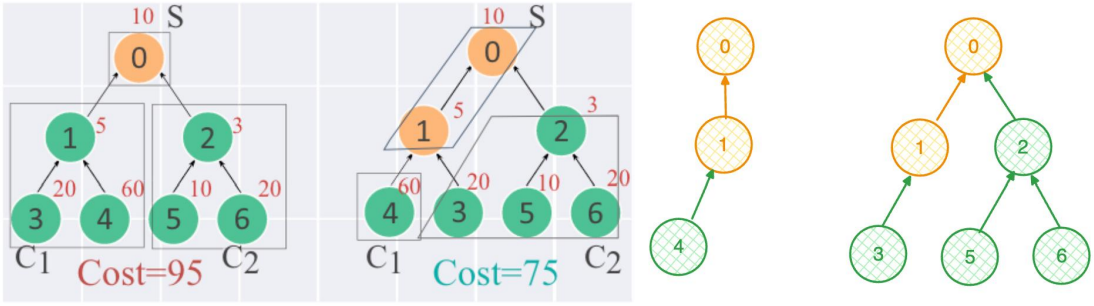


图 14 利用分层的分割点不同实现负载均衡

当节点 4 进行 schur 更新很慢时，如果采用 2D 进行更新，则所有的程序都会由于都等待节点 4 的更新完成而被阻塞，因此，如果采用 3D 算法，则可以将节点 4 进行单独的划分，从而实现负载的均衡，获得整体流水时间的减少。

所以，这个算法实际上就是 e-tree 的合并算法，不断将叶子节点合并到该叶子节点的父亲节点上，直到到根节点结束，此时，A 矩阵就已经被分解成为 LU 型矩阵了。

Algorithm 1 3D Sparse LU factorization algorithm

```
function dSparseLU3D(A, E_f):
# All process grids execute this function in parallel. P_z is the number of 2D grids
  P_z ← 2^l. Each process-grid has a unique
  p_z ∈ {0,... P_z − 1}. E_f is the local elimination
  tree-forest (section III-C).
for lvl in l:0 :
   if p_z = k2^(l−lvl) for some integer k:
#At lvl-th level the only grids that participate are those numbered as a multiple of 2^(l−lvl).
  The following call factors all supernodes of
  this level E_f[lvl] in the 2D grid, and
  performs the Schur-complement update on
  their copy of ancestor blocks.
      dSparseLU2D(A,E_f[lvl])
      if lvl > 0:
         if k(mod 2) ≡ 0:        # Note p_z = k2^(l−lvl)
            dest = p_z
            src = p_z + 2^(l−lvl)
         else:
            src = p_z
            dest = p_z − 2^(l−lvl)
         for l_a in lvl − 1: 0:
# Any supernode s in E_f[l_a] consists of blocks
  A_s = {A_ss ∪ A_{s, s+1:n} ∪ A_{s+1:n, s}}. If any process
  with co-ordinate (p_x,p_y,src) owns any block of
  A_s, then it will send that block to the
  process with coordinate (p_x,p_y,dest), which
  then reduce the two copies.
            for s ∈ E_f[l_a]:
               if p_z = src:
                  Send A_s^src to dest
               else :
                  Receive A_s^src from src
                  A_s^dest = A_s^dest + A_s^src
```

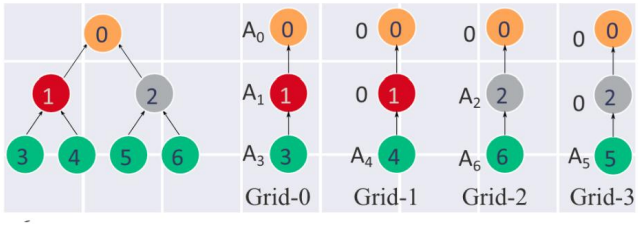


图 15 在 2D 基础上改进的 3Dsuper_LU 算法

基于 2D PDE 和 3D PDE 分别对 2D 算法和 3D 算法进行理论上的分析，结果如下：

Table II: Asymptotic memory, communication and latency costs for 2D and 3D Sparse LU algorithm

	2D PDE			3D PDE	
Parameter	dSparseLU2D	dSparseLU3D	dSparseLU3D $P_z = \mathcal{O}(\log n)$	dSparseLU2D	dSparseLU3D
Per-process Memory (M)	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n}{P}(\log(\frac{n}{P_z})+P_z)\right)$	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n^{\frac{4}{3}}}{P}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{P}\left(\kappa P_z+\frac{1}{P_z^{1/3}}\right)\right)$
Pre-process Communication(W)‡	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\log n\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}(2\sqrt{P_z}+\frac{\log n}{\sqrt{P_z}})+\frac{nP_z}{P}\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)^{\dagger}$	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\left(\kappa_1\sqrt{P_z}+\frac{1-\kappa_1}{P_z^{4/3}}\right)\right)$
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z}+\sqrt{n}\right)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z^{2/3}}+\kappa_0 n^{2/3}\right)$

† when $P \gg \log n$

‡ on the critical path of factorization. Average per-process communication is $\mathcal{O}\left(\frac{n\log n}{P_z\sqrt{P_{XY}}}\right)$. For $P_z = \mathcal{O}(\log n)$, both are asymptotically same and equal to $\mathcal{O}\left(\frac{n}{\sqrt{P_{XY}}}\right) = \mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$.

图 16 理论分析结果

实验验证：4 组平面的 PDE 和 6 组非平面的 PDE

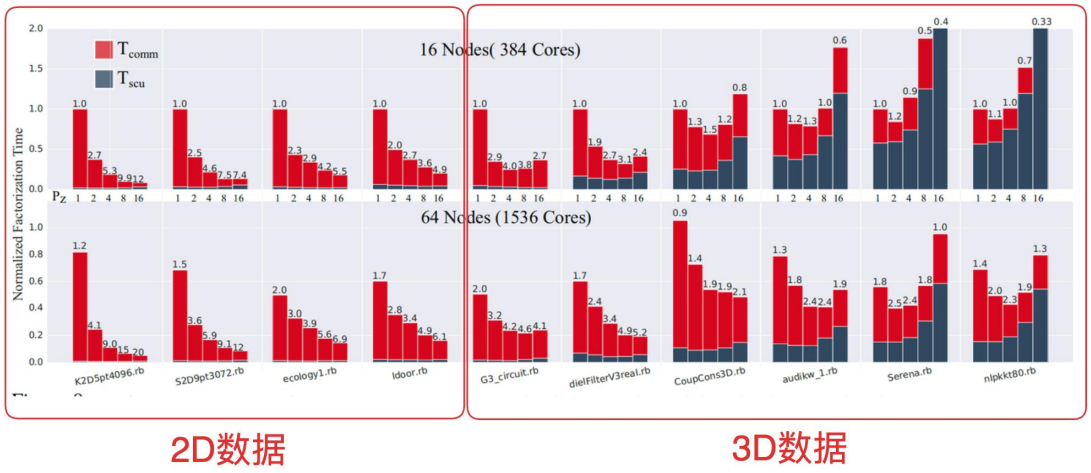


图 17 不同分层、节点数量、数据下的 schur 通信量和同步通信量

可以看到，在 2D 数据上，随着节点数的增加，2D 算法所用于通信的时间反而会增加很多，但是当采用 3D 算法时，当分层不变时，节点数的增加并不会使得通信时间增加很多，节点数不变时，分层的数量变多会使得通信的时间减少。在 3D 数据上，使用的节点数越多会使得通信的时间越少，但是当分层大到一定程度，且 2Dgrid 又比较小时，由于测试数据都集中在 etree 的上部进行计算，因此，此时顶部的 schur 更新会比较多，所以使得通信时间变大。

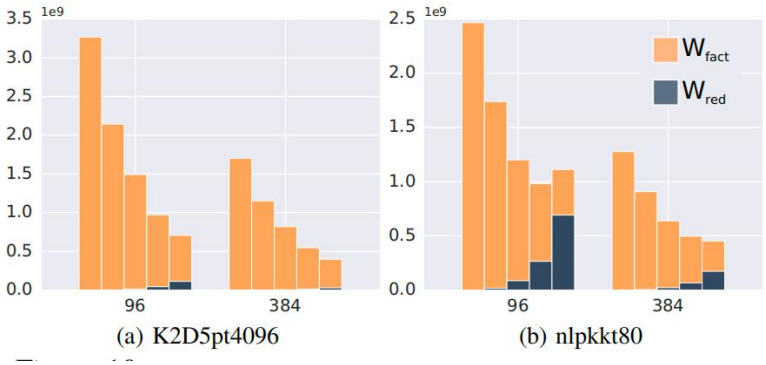


图 18 不同数据集、节点下的平面内通信量和层与层之间的通信量

上图是各个进程之间的通信量，黄色代表一个平面上进程的通信量，而灰色代表数据沿着 z 轴进行传递的通信量。当层数一定时，核心增加时，每个进程的通信量都会进行减少。当核心数量一定时，分层越多，则平面上进程的通信量一定会减少，反而增加的是沿着 z 轴进行合并的通信量。

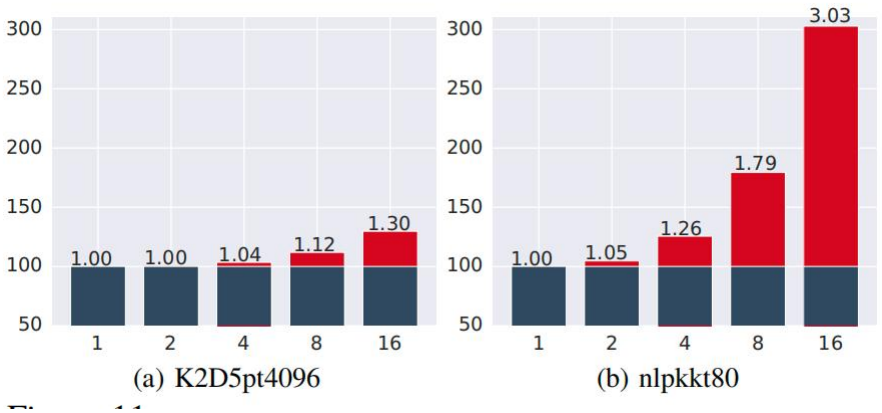


图 19 内存开销对比

3D 算法所需要的内存开销一定会比 2D 算法所需要的要大，而且是随着分层的层数的变大而变大的。

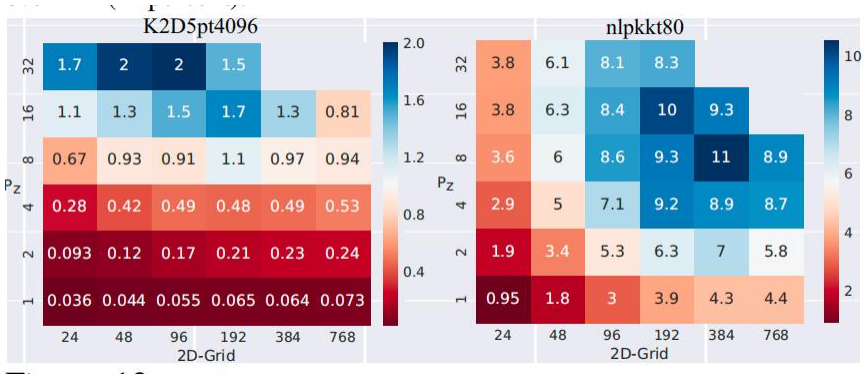


图 20 运行速度对比

运算速度和很多因素都有关系，但是可以看出，3D 算法的运算速度在相同节点下是大于 2D 算法的。

Next（下一步）：

想动手开始复现某篇论文或开始学习并行编程