arXiv:2305.04635v1 [math.NA] 8 May 2023

使用 OpenMP 任务对带状矩阵进行并行 Cholesky 分解

Felix Liu[1], [2][0000−0001−6865−9379]、Albin Fredriksson[2],和 Stefano Markidis[1]

[1] 瑞典斯德哥尔摩皇家理工学院 felixliu@kth.se[2] 瑞典斯德哥尔摩 RaySearch Laboratories

摘要。 Cholesky 分解是求解涉及对称正定矩阵的线性方程组的一种广泛使用的方法；对于需要高度数值稳定性的应用而言，它也是一种颇具吸引力的选择。数值优化就是此类应用之一，其中广泛采用直接方法求解线性方程组，而这往往会成为显著的性能瓶颈。放射治疗计划制定就是一个典型例子，也是本研究所关注的特定优化问题：通过数值优化为患者制定个性化治疗计划。为解决这一瓶颈，我们提出一种基于任务的多线程方法，用于对中等带宽的带状矩阵进行 Cholesky 分解。我们使用 OpenMP 任务实现该算法，并将其性能与 Intel MKL 等先进库进行比较。性能测量结果表明，在两种不同的英特尔中央处理器系统上，针对各种矩阵带宽，我们的性能与 Intel MKL 相当或更优（最高可达 ˜26%）。

关键词： Cholesky 分解、基于任务的并行、线性求解器

1 引言

Cholesky 分解是一种广为人知的方法，用于求解矩阵为对称正定矩阵的线性方程；它属于一类通常称为直接 法的算法，用于求解线性方程组[5]。虽然迭代法通常被视为求解大型线性方程组的最先进方法，但由于线性方程组可能存在病态等问题，在某些应用中，直接法仍是标准做法。例如，数值优化领域就包括此类应用；在其中一些算法中，随着算法的推进，方程组会变得越来越病态 [13]。许多应用中涉及的矩阵并非稠密矩阵，而是具有某种结构。带状矩阵就是一种常见的结构，其所有非零元素与主对角线的距离不超过 k 行和列。在带状矩阵的语境中，数值 k 通常称为矩阵的带宽 。按照此定义，对角矩阵是带宽为零的带状矩阵。我们强调

这里的“带宽”一词，不应与例如指计算机硬件内存通道吞吐量时所说的带宽混淆。

本研究的具体动机是放射治疗计划中的数值优化（有关放射肿瘤学优化的背景，请参见例如 [22,1] 及其中的参考文献），该领域利用数值优化为个体患者制定专门的治疗计划（即治疗设备的控制参数）。在这类问题中，我们发现某些情形需要对带宽达数百的带状矩阵进行分解。此外，我们发现这构成了一个显著的计算瓶颈：带状矩阵分解占治疗计划优化总耗时的 50% 以上。

当然，优化问题也出现在众多不同的应用领域中，例如运筹学、模型预测控制等。模型预测控制领域的一个例子是 Wang 和 Boyd 的研究 [24], ，他们提出了一种计算方法，其中涉及对带宽约为 100 的矩阵进行分解。对于希望进一步了解优化算法的读者，我们推荐参阅 [10]中的综述。对于希望从更高层面了解优化方法和高性能计算（HPC）的读者，我们推荐参阅 [17]中的综述。

本文提出了一种基于任务的方法，用于对带状矩阵进行 Cholesky 分解的并行化。我们展示了，与当前最先进的库相比，我们的方法在具有较大带宽的矩阵上表现良好，并进一步讨论和分析了其性能。我们将所提出的贡献总结如下：

– 我们设计并实现了一种基于任务的带状矩阵 Cholesky 分解并行方法，采用 OpenMP。– 我们评估了该方法的性能，并与当前最先进的库进行比较，测试对象为带宽介于 50–2000 之间的矩阵；这类矩阵可见于放射治疗中的优化问题。– 结果表明，与 Intel MKL 等当前最先进的库相比，我们的方法平均性能最高可提升 26%。

2 背景

Cholesky 分解是一种广为人知的方法，用于求解具有对称正定矩阵的线性方程组。该方法基于这样一个观察：每个对称正定矩阵 A 都存在一种形式为 $A = LL^T$ 的分解，其中 L 是一个下三角矩阵。这种分解称为*Cholesky* 分解，并且是唯一的 [6, Ch. 2.7 p. 77]。

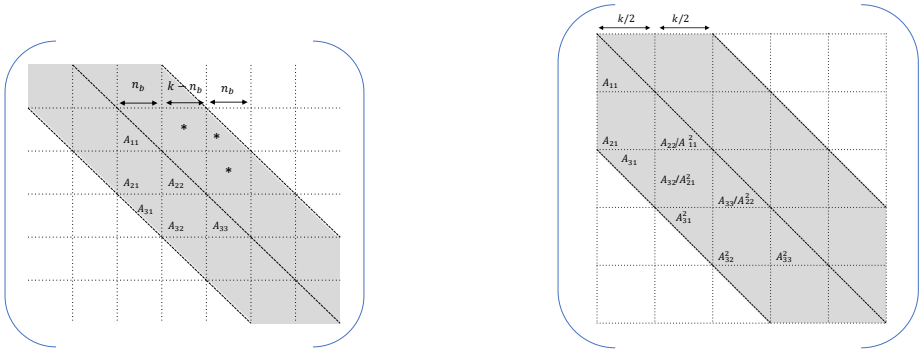
带宽非常大的带状线性系统可能出现的一个应用领域是数值优化（例如参见 [25,24]），其中许多算法在每次迭代中都需要求解一个块结构线性方程组。当前最先进的优化求解器通常依赖矩阵分解算法来完成这一步求解。例如，在内点法[10] 中——这是一种广泛用于处理带约束优化问题的算法

——需要求解的线性方程组会因优化问题的目标函数和约束结构而具有特定结构。在某些情况下，通常会利用这种结构，例如采用分块消元；这可能导致需要求解具有特定结构的线性方程组，其中一种可能的结构就是带状结构 [24]。

2.1 Cholesky 分解

为我们的方法提供初始灵感的实现方案是 Du Croz 等人在 [4] 中提出的方法。其核心思想是将分解过程中的计算组织为针对稠密子块的运算，从而能够使用三级 BLAS 内核。Du Croz 等人的方法将矩阵非零带中的当前活动窗口划分为一个 3×3 分块网格，如图 1a 所示。需要注意的是，只有该 A_{31} 块的上三角部分位于矩阵带内。在实现中，通过使用一个与 A_{31} 具有相同维度的内部方形工作数组，并将其下三角部分置零，来处理这一问题。该方法的基本步骤如下：

1. 使用稠密 Cholesky 分解将 A_{11} 分解为 $L_{11}L_{11}^T$ 2. 使用 DTRSM 计算 $L_{21} = A_{21}(L_{11}^T)^{-1}$ 3. 使用 DSYRK 计算 $A'_{22} = A_{22} - L_{21}L_{21}^T$ 4. 将 A_{31} 的上三角部分复制到方形工作数组中 5. 使用 DTRSM 计算 $L_{31} = A_{31}(L_{11}^T)^{-1}$（将 L_{31} 存储到工作数组中，覆盖 A_{31}） 6. 使用 DGEMM 计算 $A'_{32} = A_{32} - L_{31}L_{21}^T$ 7. 使用 DSYRK 计算 $A'_{33} = A_{33} - L_{31}L_{31}^T$ 8. 将 L_{31} 的上三角部分复制到主矩阵中。



(a) 展示块的图形-算法 for 带状的 矩阵来自 Du Croz 等人 [4]。

（b）两次迭代的示意图我们的 3×3 子并行算法窗口。

图 1：分块算法示意图。

3 方法

通过检查不同步骤之间的依赖关系，可以发现，列表 ?? 中概述的计算方案很自然地适用于基于任务的并行化。例如，我们可以看到，步骤 (2) 在计算 L_{11}^T 时依赖步骤 (1)，而步骤 (3) 在计算 L_{21}时依赖步骤 (2)，依此类推。这种并行化方案的一个主要缺点是，3×3 个活动窗口可能无法提供足够的并行度，因而无法充分利用硬件。

在这项工作中，我们扩展了基于块的 3×3 算法，使其能够将活动块进一步细分为更细的 $n \times n$ 子块。这样做的好处是在并行化算法时可以获得更细的任务粒度（因为每个任务都包含对网格中单个单元格的操作）。我们方法的一个优势是采用标准的 LAPACK 存储格式，这降低了现有代码采用我们方法的门槛，并避免了将矩阵转换为专用内部存储格式时可能产生的额外开销。需要指出的是，我们的代码中不使用任何显式屏障，而是由 OpenMP 运行时处理所有任务依赖和调度。我们的算法的伪代码实现如算法 1 所示。

算法 1 细粒度 Cholesky 分解

要求： $n \geq 0$. $\{n$ *is the dimension of the* $n \times n$ *grid in each iteration*$\}$

对矩阵中的每个活动窗口**执行**
  使用稠密 Cholesky 分解将 A_{11} 分解为 $L_{11}L_{11}^T$
  对 $i \leftarrow 2$ 至 $n-1$ **执行**
    使用 DTRSM 计算 $L_{i1} = A_{i1}(L_{11}^T)^{-1}$
    对 $j \leftarrow 2$ 至 $i-1$ **执行**
      使用 DGEMM 计算 $A'_{ij} = A_{ij} - L_{i1}L_{j1}^T$
    结束循环
    使用 DSYRK 计算 $A'_{ii} = A_{ii} - L_{i1}L_{i1}^T$
  结束循环
  $\{$*The last row requires special handling, since the bottom-left block is cut off by the banded structure of the matrix.*$\}$
  将 A_{n1} 复制到工作数组（一个显式存储左下三角的方阵设为零）。
  使用 DTRSM 计算 $L_{n1} = A_{i1}(L_{11}^T)^{-1}$ （覆盖工作区中的值数组）。
  对 $k \leftarrow 2$ 至 $n-1$ **执行**
    使用 DGEMM 计算 $A'_{nk} = A_{nk} - L_{n1}L_{k1}^T$ （其中 L_{n1} 存储在工作数组中）
  结束循环
  使用 DSYRK 计算 $A'_{nn} = A_{nn} - L_{n1}L_{n1}^T$ （其中 L_{n1} 存储在工作数组中）
结束循环

算法 1 中所示不同步骤之间的依赖关系分析

当仅考虑一次外层迭代 （即算法 1 中第 1 行循环的一次迭代）内部的操作时，分析相对直接。每个作用于 A_{i1} 分块的 DTRSM 操作都依赖于 A_{11} 分块的分解（对于左下分块，还依赖于向工作数组的复制操作）；DGEMM 操作依赖于两个使用 DTRSM 执行的更新操作；而作用于 A_{ii} 分块的 DSYRK 操作则依赖于同一块行中最左侧分块的 DTRSM 操作。由于需要使用临时数组来存储 A_{n1} 分块的右上三角，因此对于最底部的块行，需要特别处理。为了进一步减少所需的同步操作，我们还将任务依赖分析扩展到包含多个外层迭代的情况。这可以通过确保当前迭代的分块网格与前一次外层迭代的块网格部分重叠来实现，对于 $n = 3$ 情况，图 1b 给出了示意图。因此，每个与前一次迭代重叠的 A_{ij} 分块上的更新操作，也依赖于前一次迭代中 $A_{i+1、\ j+1}$分块上的操作。不过，这种重叠方式需要假设在我们当前的实现中，每个 $n \times n$ 子网格的维度 $n - 1$ 都能整除带宽 k 。

最后，算法中如何选择将每个 $n \times n$ 活动窗口划分成的子块数量，仍是一个待解决的问题。回顾一下，我们要求 $n - 1$ 必须整除矩阵的带宽 k 。作为初步方案，我们实现了一种启发式方法，用于选择合适的值 n ，既满足整除要求，又能为算法提供足够的并行度和合适的块大小。该启发式方法通过选择一个值 n 来平衡以下要求：

- $n - 1$ 能整除矩阵的带宽 k
- n 的选取应使三级 BLAS 操作的块大小约为 50×50。
- n 不超过系统的物理核心数。

这些标准是基于我们在系统上调优性能的实验经验制定的。不过，这种启发式方法仍较为粗略，未必能在所有配置和规模下实现最佳性能。因此，对于使用场景的矩阵带宽和硬件配置已有先验了解的用户，可以单独调整块数以获得更高性能。

3.1 使用 OpenMP 任务实现

我们使用 C++ 实现了该方法的原型，其中依靠高效的 BLAS 实现来执行算法 1 中的分块计算。为了实现基于任务的带状矩阵并行 Cholesky 分解，我们使用带有任务依赖的 OpenMP 任务，这是 OpenMP 4.0 的一项特性。为了指定任务之间的数据依赖，OpenMP 为任务编译指令提供了 in、inout 和 out 子句，我们利用这些子句定义上一节所述的任务依赖。如清单 1.1 所示，我们使用了一个哑数组来

指定任务依赖，该数组不会被实际任务直接访问。之所以可行，是因为 OpenMP 实现并不关心任务是否实际访问和修改了任务依赖，而只是构建依赖图，就好像数据已被修改一样。因此，哑数组中索引为 $(i,\ j)$ 的元素在逻辑上表示当前活动窗口中行索引为 i、列索引为 j 的分块。对于想了解其他基于任务的并行编程框架的读者，我们推荐参阅 [19] 中的研究。其余内容

```
1 char 任务依赖[块维度][块维度]; 2 #pragma omp 并行 3 #pragma omp 单个4 {5 for (int i = 0; i < 矩阵维度; i += nb) {6 #pragma omp task依赖(out:任务依赖[0][0]) depend(in: 任务依赖[1][1]) 7 dpotrf(...); 8 9 for (int blk i = 1; blk i < 块维度−1; ++块 i) {10 #pragma o mp task depend(in: 任务依赖[0][0], 任务依赖[块 i + 1][1]) \11 depend(out:任务依赖[块 i][0]) 12 cblas dtrsm(...); 13 for (int 块 j = 1; 块 j <= 块 i; ++块 j) {14 #pragma omp task依赖(in: 任务依赖[块 i][0], 任务依赖[块 j][0]) \15 depend(out: 任务依赖[块 i][块 j]) 16 cblas dgemm(...); 17 }18 #pragma omp task depend(in: 任务依赖[块 i][0], \19 任务依赖[块 i + 1][块 i + 1]) \20 depend(out: 任务依赖[块 i][块 i]) 21 cblas dsyrk(...);22 } 23 }24 }
```

清单 1.1：展示如何使用 OpenMP 实现任务化处理的精简 C++ 代码片段。为清晰起见，省略了函数参数。

我们的实现还依赖某种 BLAS 实现，以提供列表 ?? 中各步骤所用的 3 级 BLAS 内核；此外，还依赖 LAPACK 中的 dpotrf 实现（用于步骤 (1)）。需要注意的一点是，我们的实现假设带宽 k（定义为矩阵上、下超对角线的数量）可被 $n-1$ 整除，其中 n 是每次迭代中 $n \times n$ 分块网格的维度。我们的启发式方法（见第 3 节）会在可能的情况下选取这样的 n。当然，在某些情况下，这一要求无法满足（例如，带宽可能是素数）。因此，我们的最低要求是输入矩阵的带宽至少为偶数，这样才能有效地划分为 个块；用户可以在分配内存时通过对矩阵进行零填充来确保这一点。

此外，我们的实现已在 Github 上以开源形式提供[3]

3.2 性能模型

本文讨论的带状矩阵 Cholesky 分解算法主要由对三级 BLAS 内核的调用构成，而这些操作通常受计算能力限制。要评估计算密集型算法实现的性能，一种简单的方法是考察对给定维度和（矩阵）带宽的矩阵进行分解所需的浮点运算次数。由于 Cholesky 分解无需选主元即可保持稳定，因此，对给定维度的任意对称正定矩阵进行分解所需的最少浮点运算次数是固定的，且可以相当精确地计算出来。通过考察乔列斯基分解的左看算法，可以计算所需的浮点运算次数；该算法中的部分循环会因带状结构而从稠密算法中截断。算法的伪代码如算法 2 所示。计算这

算法 2 带状矩阵的左看型 Cholesky 分解

输入： 矩阵 A，维度为 N，带宽为 k
输出： 因式分解后的矩阵 L

对于 $i \leftarrow 1$ 至 N **执行**
  对于 $j \leftarrow \max(1,\ i-k)$ 至 i **执行**
    $t \leftarrow 0$
    对于 $l \leftarrow \max(1, i-k)$ 到 j **执行**
      $t \leftarrow t + L(i,\ l) * L(j,\ l)$
    结束循环
    如果 $i == j$ **那么**
      $L(i,\ i) \leftarrow \sqrt{A(i,\ i) - t}$
    else
      $L(i,j) \leftarrow (A(i,j) - t)/L(j,j)$
    结束 if
  结束循环
结束循环

分解所需的浮点运算次数可以用一种直接的方法计算：只需将算法中的计算语句替换为其浮点运算次数，再求和得到总数。令 $r = \max(1, i-k)$，则所得总和为

$$\sum_{i=1}^{N}\sum_{j=r}^{i}\sum_{l=r}^{j} 2 + \sum_{i=1}^{N}\sum_{j=r}^{i} 2, \tag{1}$$

[3] https://github.com/felliu/BandCholesky

这里将平方根视为一次浮点运算。虽然可以在计算机上精确计算这个总和（这也是我们得出基准测试所用精确浮点运算次数的方式），但也可以得到所需运算次数的数量级近似值。我们有

$$\sum_{i=1}^{N}\sum_{j=r}^{i}\sum_{l=r}^{j}2+\sum_{i=1}^{N}\sum_{j=r}^{i}2\approx\sum_{j=i-k}^{i}\sum_{l=i-k}^{j}2N+2Nk=$$
$$\sum_{j=i-k}^{i}2N(j-(i-k)+1)+2Nk=2N\sum_{l=1}^{k}l+2Nk$$
$$\approx Nk^2+2Nk=\mathcal{O}(Nk^2),$$

其中，第一次近似忽略了 max(1，$i-k$) 中的截断，而第二次近似（倒数第二步）利用了第一项为等差数列这一观察。

4 实验设置

我们在随机生成的正定带状矩阵上评估所提出的方法（通过使矩阵对角占优来确保正定性），因为矩阵元素的取值不会影响乔列斯基分解所需的运算次数（只要矩阵保持对称正定即可）。在以下所有实验中，我们将矩阵的维度（行数和列数）固定为 100,000，并改变矩阵带宽。

4.1 基准测试系统

下面列出本研究中使用的基准测试系统。

– **Coffee Lake 工作站** 是一款工作站笔记本电脑，配备六核 Intel Xeon E-2186M（Coffee Lake）中央处理器，运行 Ubuntu 22.04 LTS。– **Kebnekaise** 是位于瑞典于默奥（Ume˚a）的 HPC2N 高性能计算集群。Kebnekaise 集群的每个节点配备两颗 Intel Xeon E5-2690v4（Broadwell）处理器。这些节点运行 Ubuntu 20.04 LTS。

为了运行基准测试，我们使用 Google Benchmark[4] 套件，这是一个 C++库，提供了用于运行（微型）基准测试的各种实用工具。我们让基准测试套件决定运行基准测试的迭代次数（通常约为 10 次），然后将每次运行重复 10 次，以收集统计数据并估计噪声，所有这些都使用 Google Benchmark 的内置功能完成。在下面的所有图表中，我们报告中位时间，以排除受系统噪声等因素影响的异常值的影响。

对于我们的任务实现，我们使用英特尔的 OpenMP 运行时库，并将其与 GCC 编译的代码链接（英特尔的 OpenMP 运行时库有一个

[4] https://github.com/google/benchmark

兼容层，包含 GNU OpenMP 符号）。我们使用 GCC 11.2.0 和 CMake 编译代码，并将 CMake 构建配置为 Release 模式（这意味着 GCC 的优化级别为 -O3）。实验中使用了以下软件版本：

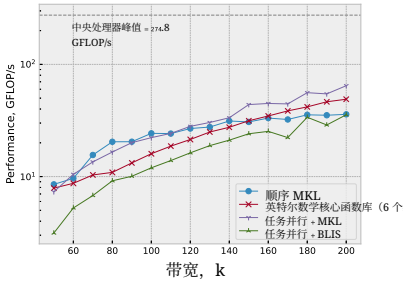
– 英特尔数学核心函数库 2022.1.0 版本（Coffee Lake 工作站）和 2022.2.0 版本（Kebnekaise）– BLIS 0.9.0 版本– OpenBLAS 0.3.20 版本– PLASMA 21.8.29 版本

5 结果

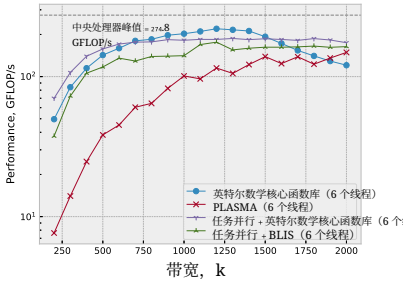
实现	Kebnekaise			Coffee Lake 工作站	
问题设置	低带宽 （50-200）	高带宽 （200-2000） 整节点	高带宽 （200–2000） 单路插槽	低带宽 (50-200)	高带宽 (200-2000)
任务并行 +英特尔数学核心函数库	**22.554**	**218.574**	**269.133**	**32.158**	**168.402**
任务并行 +BLIS	12.224	176.513	186.601	17.9	139.82
MKL 多线程	17.256	161.078	249.022	25.089	162.2
MKL 顺序执行	17.852	-	-	25.731	-
PLASMA	-	127.361	128.636	-	88.705
相较于 最佳基线	**26.283%**	**35.694%**	**8.076%**	**24.978**%	**3.824%**

表 1：不同实现的平均性能（以 *GFLOP/s* 计），涵盖不同的矩阵带宽范围、测试系统和实现。测试的具体矩阵带宽与图 2 和图 3 中的相同。性能最佳的实现以粗体显示，相对于性能最佳的基线实现（即顺序版和多线程英特尔数学核心函数库及 PLASMA）的提升显示在底行。

下一节中，我们将展示性能结果，对比我们的并行 Cholesky 分解与不同的先进库（英特尔 MKL 和 PLASMA）及其设置。请注意，我们的实现依赖于 BLAS 实现以及 LAPACK 中 dpotrf 的实现。在部分结果中，我们会在算法中使用英特尔 MKL 来满足这些依赖项。这与单独的 MKL 结果不可混淆；后者使用 MKL 的 dpbtrf（用于带状矩阵 Cholesky 分解的 LAPACK 内核）。在下文所有涉及我们基于任务的实现的实验中，我们使用第 3 节所述的启发式方法来确定每次迭代中分块网格的维度 n 。



(a) 较小的矩阵带宽。



(b) 更大的矩阵带宽。

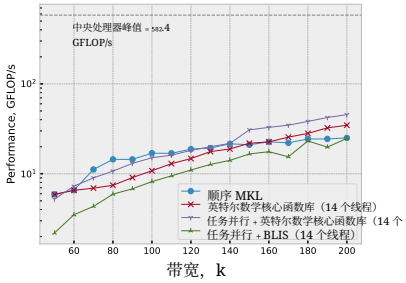
图 2：Coffee Lake 工作站上不同 Cholesky 分解的性能比较。性能以 GFLOP/s 表示。左图显示较小矩阵带宽下的性能，右图显示较大矩阵带宽下的性能。请注意，y 轴采用对数刻度。

在每次迭代中。在所有图表中，性能以 GFLOP/s 表示，所需浮点运算次数按照第 3.2 节所述计算（尤其是，用于计算 FLOP/s 的浮点运算次数对所有基准测试都相同）。图表还展示了中央处理器（或部分 Kebnekaise 基准测试中的节点）以双精度执行时的峰值性能，单位为 GFLOP/s。峰值性能数值取自英特尔为其中央处理器提供的导出合规性指标（可在线获取[5]）。

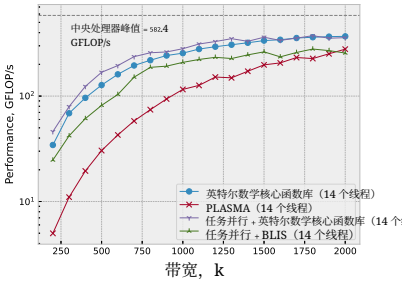
Coffee Lake 工作站上的性能图如图 2 所示，Kebnekaise 上的结果如图 3 所示。在 Kebnekaise 上，计算节点具有两个位于不同 NUMA 域的 CPU 插槽。对于较大的矩阵带宽，我们分别展示了使用单个 CPU 插槽（从而避免 NUMA 影响）的结果（图 3b），以及使用整个节点的结果（图 3c）。当 Google Benchmark 报告的运行时间标准差超过 5% 时（这种情况仅出现在图 3b 的结果中），我们在图中以阴影区域显示样本标准差的（对称）偏移量。不同矩阵带宽范围内的平均性能（低范围为 50–200，高范围为 200–2000）以 GFLOP/s 为单位汇总于表 1。

我们发现，在考虑整个矩阵带宽范围内的平均性能时，使用英特尔数学核心函数库的 BLAS 后端的基于任务的实现表现最佳，而英特尔数学核心函数库的 dpbtrf 在大多数情况下表现次佳。不过，正如图中所示，在某些配置和带宽下，英特尔数学核心函数库的表现更好。性能差异

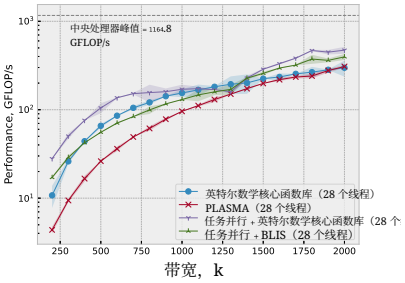
[5] https://www.intel.com/content/www/us/en/support/articles/000005755/processors.html)



(a) 较小矩阵带宽，单个 CPU 插槽



(b) 较大矩阵带宽，单个 CPU 插槽。



(c) 较大矩阵带宽，整节点。

图 3：Kebnekaise 上不同 Cholesky 分解实现的性能比较。性能以 GFLOP/s 表示（越高越好）。图中展示了两种运行方式：一种使用包含两颗英特尔 Xeon E2690v4 CPU（28 个物理核心）和对应的两个 NUMA 域的整节点；另一种使用单路插槽（14 个物理核心）和单个 NUMA 域。

对于不同矩阵带宽，我们基于任务的实现可能会受到用于选择子块大小的启发式方法（第 3 节中所述）的影响，而该方法目前仍相当粗糙。此外，我们发现，与使用英特尔数学核心函数库作为 BLAS 后端相比，我们基于任务的方法使用 BLIS 作为 BLAS 后端时，性能下降相当显著。一个可能的原因是，英特尔数学核心函数库的 BLAS 三级内核针对较小矩阵的调优可能更好（每次 BLAS 调用中的矩阵大小通常约为 50×50）。先前的工作 [11] 也观察到了这种较小矩阵的性能差异。在我们的实验中，PLASMA 的性能低于英特尔数学核心函数库，我们认为这是因为将基准测试中使用的标准 LAPACK 格式的矩阵转换为 PLASMA 的内部存储格式会产生开销。平均而言，对于较小的矩阵带宽，我们相较于英特尔数学核心函数库的性能提升更大。平均性能增益】【。

在各种矩阵带宽下，性能都远未达到中央处理器的峰值性能；随着矩阵带宽减小，性能差距显著扩大。对于最大的矩阵带宽，性能最佳的实现达到中央处理器峰值性能的约 70%。

6 相关工作

与我们采用类似思路的带状矩阵并行 Cholesky 分解的早期研究包括 Quintana-Ortí 等人 [20],的工作，该工作是在 SuperMatrix [3] 框架中实现的。我们的工作与 Quintana-Ortí 等人的工作不同之处在于，我们对带状矩阵采用标准的打包 LAPACK 存储格式，并使用 OpenMP 进行任务调度（ Quintana-Ortí 等人的工作发表时，OpenMP 任务的实现尚处于起步阶段）。我们认为，这降低了现有代码采用该方法的门槛，并消除了将矩阵转换为内部存储格式可能带来的额外开销。

多线程 Cholesky 分解这一主题此前已在文献中得到广泛研究。例如，Remon 等人 [21]研究了带状矩阵 Cholesky 分解的多线程性能。在他们的工作中，Remon 等人提出了一些性能优化方法：通过对 LAPACK 中带状矩阵的存储方案进行少量修改，将 List ?? 中的若干计算步骤合并为对 BLAS-3 内核的单次调用，从而提高每次调用 BLAS 内核时可用的并行度。Gustavson 等人 [15] 也研究了对 LAPACK 存储方案进行小幅修改，以合并内核并提高效率。

作为更广泛努力的一部分，任务并行带状 Cholesky 分解此前也得到过研究；这些努力旨在利用基于任务的并行性处理不同的计算内核，例如 FLAME [23]和 PLASMA [7] 项目。这项工作的一部分，便是前文提到的 Quintana-Ortí 等人在 [20]中的研究。在分布式计算方面，Gupta 等人 [14]等也研究了带状 Cholesky 分解的并行算法。对于一般的稠密矩阵，Dorris 等人 [8], 研究了这一主题，并探讨了 Cholesky 分解算法的不同变体及其对基于任务的并行性的适用性。对于稀疏矩阵，Liu [18], 以及 Geist 和 Ng [12], 在 20 世纪 80 年代研究了基于任务的并行 Cholesky 分解，Hogg 等人 [16]则在近年进行了相关研究。

7 结论与未来工作

本文介绍了我们对一种基于任务的并行算法在带状矩阵 Cholesky 分解中的性能评估工作。结果表明，对于维度和带宽与前文所述优化问题可能产生的矩阵相似的矩阵，我们的方法平均而言优于 Intel MKL 等最先进的库。

不过，要实现最佳性能可能取决于多种因素，包括所用中央处理器硬件的具体情况、输入矩阵的维度等。

此外，鉴于图形处理器在高性能计算中的使用率迅速上升，研究我们的算法是否适合移植到图形处理器上，是一个值得探讨的问题。在实现方面，可以使用基于任务的并行编程模型来面向图形处理器，例如使用 OmpSS 编程模型[9,2]。另一种可能的方法是使用 CUDA 10 引入的 CUDA Graph 功能，显式构建由内核及其依赖关系组成的图，并在图形处理器上执行。无论采用哪种具体实现方式，我们认为面临的主要挑战之一是：当带宽尺寸适中时，带状矩阵 Cholesky 分解所能提供的并行度有限（我们的结果表明，即使矩阵带宽较小，其性能也远未达到中央处理器的峰值性能）。图形处理器通常需要大量可用并行度才能达到峰值性能。因此，移植到图形处理器所带来的性能提升可能有限，但这一问题有待未来研究。

总之，对于我们的应用场景，我们的实现具有与 Intel MKL 相竞争的性能，同时矩阵存储方案仍与 LAPACK 兼容。最后，我们希望未来能够在实际的放射治疗问题优化流程中评估我们的算法，以衡量其在这类场景中的性能提升。

参考文献

1. Baumann, M., Krause, M., Overgaard, J., Debus, J., Bentzen, S.M., Daartz, J., Richter, C., Zips, D., Bortfeld, T.：精准医学时代的放射肿瘤学。Nature Reviews Cancer **16**(4), 234–249 (2016) 2. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguade, E., Labarta, J.：使用 OmpSs 对 GPU 集群进行高效编程。载于：2012 年 IEEE 第 26 届国际并行与分布式处理研讨会。第 557–568 页。IEEE (2012) 3. Chan, E., Van Zee, F.G., Bientinesi, P., Quintana-Orti, E.S., Quintana-Orti, G., Van de Geijn, R.：Supermatrix：一种用于分块算法的多线程运行时调度系统。载于：第 13 届 ACM SIGPLAN 并行编程原理与实践研讨会论文集。第 123–132 页 (2008) 4. Croz, J.D., Mayes, P., Radicati, G.：使用三级 BLAS 对带状矩阵进行分解。载于：CONPAR 90—VAPP IV，第 222–231 页。Springer (1990) 5. Davis, T.A., Rajamanickam, S., Sid-Lakhdar, W.M.：稀疏线性方程组直接法综述。Acta Numerica **25**, 383–566 (2016) 6. Demmel, J.W.：应用数值线性代数。SIAM (1997) 7. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., YarKhan, A., Abalenkovs, M., Bagherpour, N., 等：PLASMA：使用 OpenMP 面向多核的并行线性代数软件。ACM 数学软件汇刊 (TOMS) **45**(2), 1–35 (2019) 8. Dorris, J., Kurzak, J., Luszczek, P., YarKhan, A., Dongarra, J.：使用 OpenMP 在 Knights Corner 上进行基于任务的 Cholesky 分解。载于：国际高性能计算会议。第 544–562 页。Springer

9. Duran, A., Ayguad´e, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSS：一种用于异构多核架构编程的方案。《并行处理快报》**21**(02), 173–193 (2011) 10. Forsgren, A., Gill, P.E., Wright, M.H.: 非线性优化的内点法。《SIAM评论》**44**(4), 525–597 (2002) 11. Frison, G., Sartor, T., Zanelli, A., Diehl, M.: BLASFEO 的 BLAS API：针对小型矩阵优化性能。ACM 数学软件汇刊 (TOMS) **46**(2), 1–36 (2020) 12. Geist, G., Ng, E.: 并行稀疏 Cholesky 分解的任务调度。《国际并行编程杂志》**18**(4), 291–314 (1989) 13. Gondzio, J.: 25 年后的内点法。《欧洲运筹学杂志》**218**(3), 587–601 (2012) 14. Gupta, A., Gustavson, F.G., Joshi, M., Toledo, S.: 面向分布式内存并行计算机的对称带状线性求解器的设计、实现与评估。ACM 数学软件汇刊 (TOMS) **24**(1), 74–101 (1998) 15. Gustavson, F.G., Quintana-Ortı, E.S., Quintana-Ortı, G., Rem´on, A., Wasniewski, J.: 用于对称正定带状矩阵分解的更清晰、更简单、更高效的 LAPACK 例程 (2008) 16. Hogg, J.D., Reid, J.K., Scott, J.A.: 使用有向无环图设计多核稀疏 Cholesky 分解。《SIAM科学计算杂志》**32**(6), 3627–3649 (2010) 17. Liu, F., Fredriksson, A., Markidis, S.: 面向大规模基于梯度的非线性优化的高性能计算算法与框架综述。《超级计算杂志》**78**(16), 17513–17542 (2022) 18. Liu, J.W.: 并行稀疏 Cholesky 分解的计算模型与任务调度。《并行计算》**3**(4), 327–342 (1986) 19. Podobas, A., Brorsson, M., Fax´en, K.F.: 常见且流行的以任务为中心的编程框架的比较性能研究。《并发与计算：实践与经验》**27**(1), 1–28 (2015) 20. Quintana-Ort´ı, G., Quintana-Ort´ı, E.S., Rem´on, A., Geijn, R.A.: 一种用于超矩阵带状 Cholesky 分解的分块算法。载于：国际计算科学高性能计算会议。第 228–239 页。Springer (2008) 21. Rem´on, A., Quintana-Ort´ı, E.S., Quintana-Ort´ı, G.: 使用多线程 BLAS 对带状矩阵进行 Cholesky 分解。载于：国际应用并行计算研讨会。第 608–616 页。Springer (2006) 22. Unkelbach, J., Bortfeld, T., Craft, D., Alber, M., Bangert, M., Bokrantz, R., Chen, D., Li, R., Xing, L., Men, C., 等：容积调强弧形放射治疗计划的优化方法。《医学物理学》**42**(3), 1367–1377 (2015) 23. Van Zee, F.G., Chan, E., Van de Geijn, R.A., Quintana-Orti, E.S., Quintana-Orti, G.: 用于稠密矩阵计算的 libflame 库。《科学与工程计算》**11**(6), 56–63 (2009) 24. Wang, Y., Boyd, S.: 使用在线优化实现快速模型预测控制。《IEEE控制系统技术汇刊》**18**(2), 267–278 (2009) 25. Wright, S.J.: 将新的优化算法应用于更具预测性的控制。技术报告，阿贡国家实验室（ANL），伊利诺伊州阿贡（美国）(1996)