arXiv:2303.02352v1 [cs.DC] 4 Mar 2023

面向迭代线性求解器的多图形处理器聚合型代数多重网格预条件器

Massimo Bernaschi, Alessandro Celestini,Pasqua D'Ambra意大利国家研究委员会应用计算研究所{name.surname}@cnr.it

Flavio Vella特伦托大学{flavio.vella}@unitn.it

2023年3月7日

摘要

我们介绍并以开源形式发布了一款稀疏线性求解器，它能够高效利用异构并行计算机。该求解器可轻松集成到科学应用中，这些应用需要在由配备 NVIDIA图形处理器（图形处理器）加速器的混合节点构成的现代并行计算机上求解大型稀疏线性方程组。

本工作扩展了我们此前对单个GPU加速器利用的研究，并提出了一种基于混合MPI-CUDA软件环境的Krylov型线性求解器实现，该求解器依赖于BootCMatchG库中已有的高效代数多重网格（AMG）预条件器。我们的混合实现设计遵循了在使用多块 GPU时最小化数据通信开销的最佳实践，同时保持单个GPU内核的效率。本文讨论了该库新版本在知名基准测试用例上的强可扩展性和弱可扩展性结果。与Nvidia AmgX解决方案的比较表明，在求解阶段性能最高提升达2.0倍。

1 引言

异构性是最先进计算平台的一个特征。事实上，在同一计算节点中使用不同类型处理器的趋势已经确立 [Shalf(2019)]。从笔记本电脑到（准）百亿亿次级Top500超级计算机 [Top500(2022)], ，当前计算平台均采用与图形处理器（GPU）等加速器相结合的多核处理器。为了将这些异构方程组高效用于通用计算，需要重新设计算法及其实现，尽管一些高效且易用的编程环境能够隐藏大部分架构细节（例如，Nvidia CUDA、SYCL或OpenACC等，仅举数例）。对于呈现不规则内存访问模式的应用程序，例如稀疏线性求解器，充分利用此类混合方程组所提供的全部计算能力尤其具有挑战性。

设 A 是一个维数为 n 的实数方阵且非奇异， x 和 b 是维数相容的实数列向量。线性求解器用于求解问题 $Ax = b$，其中 A 和 b 为已知量， x 表示未知量。当矩阵 A 含有大量零元素（通常超过总元素的 99% ），且 n 非常大时，高效可靠的线性求解器往往是对复杂模型进行可靠仿真或开展大数据分析的关键。用于求解大型稀疏线性方程组的大多数方法都属于迭代 Krylov 子空间投影方法，其中的典型代表是著名的共轭梯度（CG）方法 [Saad(2003)] ，该方法适用于 A 为对称正定（s.p.d）矩阵的情况。CG 方法得以实际应用的一个基本组成部分是所谓的预条件器，即一个新的（s.p.d.）矩阵 B，它以某种方式近似于矩阵 A 的逆。将预条件器应用于预条件系统（例如 $BAx = Bb$）时，可以加快 CG 方法的收敛。

一些现有软件工具，如 Nvidia AmgX [Naumov(2015),AmgX(2020)] 和 hypre [Falgout(2006),Fal- gout(2021)], ，能够充分利用多GPU系统。具体而言，AmgX 包含基于未知量聚合的代数多重网格（AMG）预条件器，该聚合由系统矩阵邻接图中的匹配驱动；这些方法

具有灵活性，能够平衡计算复杂度与所得线性求解器的收敛特性。另一方面，我们在 *BootCMatchG* 中实现的代数多重网格方法采用一种聚合方案，利用图中最大权匹配的某些性质；对于可在单个GPU内存中求解的线性方程组，该方法具有更好的收敛特性 [Bernaschi(2020a),Bernaschi(2020b)]。本文的主要目标是提升该方案的可扩展性，使其能够求解维数远超单个计算节点内存容量的稀疏线性方程组，例如现代计算与数据科学应用中产生的方程组。为弥补这一不足，我们对原有数值算法作了一些近似，从而通过结合通信规避技术、细粒度并行以及计算与数据通信的重叠，设计出可扩展的线性求解器。本工作的贡献包括：

- 一种适用于混合架构的并行 AMG 预条件器，其收敛性能优于类似的先进 AMG 方法，并能以较低复杂度实现良好的可扩展性；
- 一种基于并行解耦聚合、具有高通信效率的 AMG 预条件器设置方法，可使我们在图形处理器上进行匹配计算的同时，重叠执行稀疏矩阵-稀疏矩阵乘积所需的数据通信，并减少总体数据通信量；
- 一种用于并行混合稀疏矩阵-稀疏矩阵乘积的非局部稀疏矩阵行的新型数据结构管理方法，使我们能够最大限度地重复利用现有的高效 GPU 内核。

我们的多图形处理器版本*BootCMatchG* 能够求解包含超过 2 亿个未知量的线性方程组，并扩展至搭载 Nvidia 图形处理器的 100 个混合节点；在求解阶段，相较于 AmgX，其加速比为 1.6x 至 2.0x。

本文组织如下：第2节简要总结影响大规模异构系统上科学应用，尤其是稀疏线性求解器可扩展性和效率的主要问题；第3节概述*BootCMatchG* 及其组件设计的特点，这些设计旨在高效利用高度并行的处理器；第4节介绍我们在基于 Nvidia 图形处理器加速器的多GPU系统上设计分布式算法时所遵循的主要原则和指导方针；最后，第5节针对源自三维泊松方程数值求解的真实测试案例，展示强可扩展性和弱可扩展性结果。三维泊松方程是许多科学应用中普遍存在的内核。特别地，我们将结果与业界先进求解器的结果进行比较，该求解器已在 Nvidia AmgX 中实现。

2 异构高性能计算系统上的迭代线性求解器

求解大型稀疏线性方程组是计算与数据科学应用中普遍存在且耗时的任务。基于物理的模拟、用于不确定性量化的统计方法以及复杂网络分析只是其中一些应用场景。特别是，我们关注以迭代方式计算近似解的场景。当系数矩阵 A 较大且稀疏，或应用能够接受精度低于机器精度所提供精度的解时，通常会采用这种方法。在异构计算环境中，为了最大限度地提升性能，我们希望使用多核架构和图形处理器等加速器；而要有效地迭代求解线性方程组，就需要开发能够利用底层硬件和基础软件的新方法、算法和实现。事实上，无论采用何种方程组求解方法，即平稳迭代的雅可比法和高斯-赛德尔法，还是非平稳的克雷洛夫方法 [Saad(2003)], , 由于内存访问模式不规则，以及矩阵行的稀疏模式所导致的固有负载不均衡，稀疏矩阵上的计算相比其稠密对应形式都尤其具有挑战性。

图形处理器依赖细粒度并行性，以及对中等规模、高带宽存储器的访问，但其延迟也不可忽视。因此，尽管当前的软件栈提供了各种编程环境，例如 Nvidia CUDA，可为硬件特性提供清晰的接口，但要高效使用上述加速器仍然颇具挑战，尤其是对于内存受限内核，例如迭代线性求解器中涉及的稀疏矩阵-向量乘积。例如，Nvidia 图形处理器由多线程流式多处理器阵列构成，每个多处理器又由固定数量的标量处理器组成。CUDA 编程范式基于线程块共享数据这一概念。因此，矩阵各行具有规则的非零元素密度，有利于实现高吞吐量的 SIMD 操作；而一般稀疏矩阵的不规则结构则会对高效利用该架构造成一定限制，通常可通过将矩阵组织成合适的数据结构来加以平滑 [Filippone(2017)]。此外，对同一数据连续执行一系列 SIMD 操作，可以实现所谓的数据/线程局部性，从而充分发挥图形处理器的效能。出于相同原因，相较于更复杂的算法，人们更倾向于采用能够体现高度数据并行性的基础迭代算法，尽管前者可能导致数据依赖关系，且通常收敛性能较差——

尽管如此，前者通常可能具有较差的收敛特性。事实上，额外的计算往往可以被图形处理器上极为高效的执行所抵消并取得平衡 [D'Ambra(2016),Bernaschi(2020a)]。另一方面，要充分利用大型异构节点集群的可扩展性，很大程度上取决于能否妥善协调多个计算层级，从而确保数据划分、数据/工作负载均衡，以及图形处理器之间、中央处理器之间和分布式节点之间的数据通信，不会显著损害最终性能。

多年来，人们一直致力于设计和实现高效、可靠的迭代稀疏线性求解器，并将其开发为开源软件库。在使用最广泛的软件库中，我们列举 PETSc [Petsc(2020)]、Trilinos [Heroux(2005)] 和 hypre [Falgout(2006), Falgout(2021)]。正如 [Anzt(2019)]所总结的，这些软件库都经历了多次修订，以跟上技术发展的步伐。Nvidia AmgX 库 [AmgX(2020)] 提供了依赖于专门针对 GPU集群 调优的知名代数多重网格预条件器的稀疏求解器。

在这一背景下，我们提出了一个开源软件包，用于求解具有对称正定矩阵的大型稀疏线性方程组，作为高效利用当前高端超级计算机的基础组件。尽管与 Nvidia Amgx求解器相比，我们的预条件方法需要更长的设置时间，但它在多个节点上实现了更好的可扩展性，并且通常以更低的复杂度获得更快的收敛率。我们结合了预条件共轭梯度法（PCG）方法——该方法最初由 [Notay(2015)] 提出，旨在高效地应用于分布式内存机器——以及 [Bernaschi(2020a), Bernaschi(2020b)] 中提出的一些增强功能，这些功能专为图形处理器设计。

本工作面向规模极大的问题进行了扩展和改进，其维数超出了单个GPU的容量。我们采用了一种方法，在算法中引入一些近似，以在较高的可扩展性、良好的收敛性质和最终稀疏线性求解器的计算复杂度之间取得平衡。

3 预条件共轭梯度法

简要地说，我们回顾 Notay [Notay(2015)] 提出的预条件 CG（PCG）方法，该方法特别适合在分布式机器上进行扩展。它是原始 CG 的一种灵活版本，支持使用可变预条件器；该方法依赖于对基本操作的重新组织，即将内积序列和形如 $y = \alpha x + y$ 的向量更新（通常分别称为 点积和 *axpy* 操作）进行重新组织，以减少全局同步和通信延迟带来的开销。该方法在算法 1 中进行了描述，其中我们重点列出了以下主要计算内核:

1. 在第 2 行和第 9 行应用由矩阵 B 表示的预条件器；2. 在第 3 行和第 10 行进行涉及原始矩阵 A 的稀疏矩阵-向量乘法（稀疏矩阵-向量乘法）；3. 从第 11 行到第 13 行执行一系列点积指令；4. 从第 15 行到第 18 行执行一系列*axpy* 操作。

上述版本的预条件共轭梯度法在点积序列和*axpy* 操作序列中都体现了数据局部性；正如我们在 [Bernaschi(2020a)] 中所述，并在 [Bernaschi(2020b)]中实现的那样，还可以在节点级利用这种局部性来减少全局内存访问。

3.1 代数多重网格预条件器

预条件共轭梯度法中最重要的操作之一是应用预条件器，其目的是在给定精度下减少求解该系统所需的迭代次数。具体而言，预条件器是一个矩阵 B ，旨在改善系统系数矩阵的谱性质。对于对称正定矩阵，可以证明，共轭梯度法的收敛速率取决于系统矩阵 [的特征值分布]Saad(2003)。预期预条件矩阵 BA 相对于矩阵 A具有更小的条件数，即 $\kappa(BA) = \|BA\|\|(BA)^{-1}\| < \kappa(A)$，其中 $\|\cdot\|$ 是矩阵范数。

从计算角度来看，有两个主要问题会限制预条件技术的可扩展性。第一个问题取决于应用预条件器的开销：如算法 1 所示，该过程包括将稀疏矩阵B 与当前残差向量 r_i相乘；也可以采用更一般的过程，以隐式方式计算乘积。因此， B 的设置尤为重要；理想情况下，其开销应随问题规模线性增长（以浮点运算量和内存占用衡量）。其操作应尽可能局部化，以减少并行任务之间的数据通信，并使并行性能能够随着任务数量的增加而近似线性提升。我们将这些特性称为实现可扩展性。

算法 1预条件柔性共轭梯度

1: 给定 u_0 并设置 $r_0 = b - Au_0$
2: $w_0 = d_0 = Br_0$
3: $v_0 = q_0 = Aw_0$
4: $\alpha_0 = w_0^T r_0$
5: $\beta_0 = \rho_0 = w_0^T v_0$
6: $u_1 = u_0 + \alpha_0/\rho_0 d_0$
7: $r_1 = r_0 - \alpha_0/\rho_0 q_0$
8: **对于** $i = 1, \ldots$ **执行**
9: $w_i = Br_i$
10: $v_i = Aw_i$
11: $\alpha_i = w_i^T r_i$
12: $\beta_i = w_i^T v_i$
13: $\gamma_i = w_i^T q_{i-1}$
14: $\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$
15: $d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$
16: $u_{i+1} = u_i + \alpha_i/\rho_i d_i$
17: $q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$
18: $r_{i+1} = r_i - \alpha_i/\rho_i q_i$
19: **结束循环**

第二个方面与迭代次数有关，它取决于预条件器的最优收敛或算法可扩展性。理想的预条件器会生成一个条件数非常小（$\kappa(BA) \approx 1$）且与问题规模无关的预条件矩阵，从而使预条件共轭梯度法的迭代次数几乎保持不变。

本文聚焦于一类名为代数多重网格（AMG）方法的多重网格预条件器。之所以将其视为纯代数方法，是因为它们用于构建 B 的信息仅来自系统矩阵 A，而非来自应用领域的特征。这类方法在某些情况下能够同时实现计算复杂度和收敛性的最优性，因此适合进行可扩展的并行实现 [Xu(2017), Vassilevski(2008)]。

这类预条件器面临的挑战在于：需要在纯代数环境下，根据原始系统矩阵（细矩阵）建立粗网格指标空间和矩阵的层次结构；这些空间和矩阵必须足够稀疏，同时还要能够在层次结构的每个粗网格层次上，良好地表示导致预条件共轭梯度法收敛变慢的误差向量分量。

具体而言，任意代数多重网格的构建过程都会生成一个 nl 矩阵层次结构，即 $A^k = (a_{ij}^k)_{i,\ j=1,\ldots,n_k}, \quad k = 1, \ldots nl$，其中$A^1 = A$通过粗化算法生成。对于所有 $k < nl$，都会构建一个维数为 $n_k \times n_{k+1}$的延拓矩阵 P^k，并根据通常的三矩阵 Galerkin 乘积计算矩阵 $A^{k+1} = (P^k)^T A^k P^k$。

在层次结构的每一层，还定义了一个平滑器 M^k，它表示简单松弛法的迭代矩阵。在我们的方法中，我们采用加权版的雅可比迭代法，称为 ℓ_1–雅可比法。该方法具有高度并行性，并且对称正定矩阵均能保证收敛；此外，对于对角占优矩阵，它还具有良好的平滑性质 $M^k = diag\ M_{11}^k\ \ldots\ M_{n_k n_k}^k$，$M_{ii}^k = a_{ii}^k + \sum_{j \neq i} a_{ij}^k$ [2013] Brannick ( )。其迭代矩阵定义为对角矩阵 (,), 其中为 , 其逆矩阵可直接求得。

在设置阶段生成的组件可以通过多种方式组合，以得到不同类型的多重网格循环，并在算法 1（第 9 行）中作为一个过程应用，该过程计算预条件算子作用于残差向量所产生的效果。

这种组合的一个例子称为 V循环，见算法 2。在这种情况下，在对 V循环进行递归调用之前和之后（即在预平滑和后平滑阶段），都使用同一种平滑器进行一次迭代。在最粗层，即对于 $k = nl$，通常采用直接求解器。然而，在

多块 GPU 上的代数多重网格方法中，最粗层使用直接求解器有两个显著缺点：(i) 最粗层矩阵往往规模较小，因此在多节点环境中，数据通信的开销会超过本地计算的开销；(ii) 直接求解器需要求解三角系统，而由于稀疏三角结构所固有的强数据依赖和负载不均衡，这在 GPU 上尤其具有挑战性 [Yamazaki（2020）]。在这项工作中，我们采用最粗层系统的分布式迭代求解器，以减少并行处理器之间的数据依赖；具体而言，我们对平滑器采用的同一雅可比方法版本进行固定次数的迭代，且该次数足够大。这种方法能够重复利用为平滑步骤开发的内核，其优势已在该方法针对单个 GPU 的并行版本中得到验证 [Bernaschi(2020a)]。算法 2 对应于应用一个由以下序列隐式定义的预条件矩阵 B

算法 2 *V*循环

过程 V循环(k, nl, A^k, r^k)
  如果 $k \neq nl$**则**
    $r^k = (M^k)^{-1} r^k$
    $r^{k+1} = (P^{k+1})^T r^k$
    $r^{k+1} =$ V循环$\left(k+1,\ nl,\ A^{k+1},\ r^{k+1}\right)$
    $r^k = P^{k+1} r^{k+1}$
    $r^k = \left(M^k\right)^{-T} r^k$
  else
    $r^k = \left(A^k\right)^{-1} r^k$
  结束条件判断
  返回 r^k
结束过程

以下矩阵：

$$B^k = (M^k)^{-T} + (M^k)^{-1} - (M^k)^{-T} A^k (M^k)^{-1} + (I - (M^k)^{-T} A^k)(P^k B^{k+1} (P^k)^T)(I - A^k (M^k)^{-1}) \quad \forall k,$$

假设 $B^{nl} \approx (A^{nl})^{-1}$ 是最粗层矩阵的逆的一个近似。

最近，我们提出了一种代数多重网格方法，该方法依赖于一种基于未知量成对聚合的粗化算法，并利用系统矩阵邻接图中的最大权匹配（以下简称 *MWM*）。图权重由问题对应的光滑向量（即 w）和矩阵元素通过适当的函数计算得到。该过程递归应用，以获得一系列更粗的矩阵，直到最粗层矩阵的维数足够小（$coarsesize$）（见算法3）。

算法3 基于*MWM*的基本粗化

过程 COARSENING$(A,\ w,\ coarsesize)$
  $k = 1, A^1 = A, w^1 = w$
  当 $size\ (A^k) > coarsesize$**时执行**
    应用 *MWM*$(A^k,\ w^k)$
    成对延拓算子的设置 P^k
    计算 $R^k = (P^k)^T$
    计算 $A^{k+1} = R^k A^k P^k$
    计算 $w^{k+1} = R^k w^k$
    $k = k + 1$
  结束循环
  $NL = k$
  返回 $NL\ , A^2, \ldots, A^{NL}$
结束过程

匹配后构建的 $NL - 1$ 成对延拓算子对应于分片常数插值器，每行仅有一个非零项，每列有两个非零项，其值取决于所选的光滑向量。通过合并多个聚合对，使聚合体的大小几乎可以任意增大（对于给定的 $s > 1$，大小为 2^s），可以实现更激进的粗化；这可以通过组合多个基本成对聚合步骤来实现，也就是计算 s 个连续成对延拓算子的乘积；随后，最终的 $nl - 1$ 延拓算子的每列最多有 2^s

个非零项。这种分片常数算子的选择也称为未平滑或普通延拓算子，使我们能够保持粗层矩阵的稀疏性，因此最终的预条件器内存需求很小，并且可以在并行环境中高效实现每次预条件共轭梯度法迭代中的应用。关于本方法的构造及其数学原理，读者可参阅 [D'Ambra(2013)、D'Ambra(2018)、Bernaschi(2020a)、D'Ambra(2019)]。这里仅指出，AMG预条件器设置中的主要计算构件包括：

1. 递归应用*MWM*，构建基本成对延拓算子序列 P^k，用于$k = 1$、.....、 $NL - 1$；2. 通过三次（稀疏）矩阵乘法（以下简称稀疏矩阵-矩阵乘法）计算每个层级 $k = 1$、.....、 $NL - 1$上的粗矩阵 $A^{k+1} = (P^k)^T A^k P^k$；3. 通过（稀疏）矩阵-（稠密）向量乘法（以下简称稀疏矩阵-向量乘法）计算每个层级 $k = 1$、.....、 $NL - 1$上的粗层平滑向量 $w^{k+1} = R^k w^k$；4. 计算一系列形如 $P^1 = P^1 \cdots P^s$、$P^2 = P^{s+1} \cdots P^{2s}$、.....、 $P^{nl-1} = P^{\lfloor NL/s \rfloor s+1} \cdots P^{NL-1}$的稀疏矩阵-矩阵乘法，将基本成对聚合合并为大小至多为 2^s 的聚合体，并获得由 $nl - 1 = \lceil NL/s \rceil - 1$ 延拓算子构成的最终层次结构；5. 通过涉及最终延拓算子序列 P^k的Galerkin型稀疏矩阵-矩阵乘法计算新的 $nl - 1$ 粗矩阵。

—

对于基于基本成对聚合的最终AMG 层次结构，最后两个步骤可以不执行。

4 多图形处理器方程组上的BootCMatch

为实现上一节所述计算过程的混合MPI-CUDA可扩展实现，我们不得不对单GPU版本进行重大重新设计。遵循最佳实践，我们的指导原则是：通过数据和线程局部性高效利用GPU，尽量减少主机与GPU加速器之间以及多个MPI任务之间的数据交换，并尽可能利用分布式MPI任务间计算与数据通信的重叠。

在AMG预条件器设置操作中引入并行性的需求，促使我们采用了一些算法选择，从而影响了算法3中所述粗化算法的质量，具体如下文所述。然而，正如我们在第5节中所证明的，尽管在其设置过程中引入了某些形式的近似，我们的方法仍能够充分保持AMG预条件器的收敛特性。在混合实现中，与原始串行代码和单GPU版本一样，分布式稀疏矩阵采用广泛使用的压缩稀疏行（CSR）存储格式表示。此外，矩阵采用行块分布，其中每个任务拥有系统矩阵的一个连续行块，而连续的行块则分配给秩连续的MPI任务。

4.1 解耦聚合

我们的预条件器设置依赖一种计算匹配的算法，其时间复杂度接近线性，计算得到的匹配对应于次优最大权重。通常，计算最大权重匹配（最优或精确匹配）的算法本质上是串行的，且具有超线性计算复杂度 [Pothen(2019)]。在 [Bernaschi(2020a)] 中，我们讨论了将现有 GPU 实现的*Suitor* 算法改造为适用版本的工作。该算法 [Naim(2015], 计算得到的匹配权重至少是最优值的一半。尽管*Suitor*旨在利用局部主导边原则缩短关键路径，但它采用竞价策略来完成匹配。在包含多块 GPU 的分布式框架中，这种竞价需要进行多轮通信，并伴随 CPU 与 GPU 之间代价高昂的内存复制；通信轮数基本上等于任务数。显然，虽然可以开发 *Suitor* 的多图形处理器版本（我们确实开发了该版本，以比较结果质量，见下文），但它无法在实践中使用，因为会严重损害可扩展性。事实上，目前有许多研究工作正致力于为超大型图设计并行次优匹配算法，详见 [Acer(2021)]。据我们所知，目前尚无能够在分布式环境中利用多块 GPU 的高效算法和软件；开发此类算法和软件也超出了本文的范围。不过，正如下文所述，我们的目标并不需要采用多图形处理器版本的MWM计算。

一种在分布式内存编程模型中广为研究的粗化并行化策略，已被证明在实践中能够生成质量良好的粗矩阵 [Tuminaro(2000)、D'Ambra(2010)、Notay(2015)],，其基础是

所谓的未知量聚合解耦方法。具体而言，每个并行任务都会独立地对当前层矩阵行块分布中分配给自己的那部分未知量进行聚合。这种解耦聚合具有令人乐见的并行性，因为它不需要并行任务之间进行任何数据通信。然而，它也有一个可能影响粗化质量的缺点：它可能会导致边界未知量的聚合体不均匀，而边界未知量指的是由某个任务持有、且与其他任务持有的未知量相连的未知量。此外，这种方法还依赖于并行任务的数量以及系统矩阵的初始划分。我们采用算法3中的并行解耦方法，该方法仅将*MWM*内核应用于与局部未知量对应的局部子图，而忽略不同任务所持有未知量之间可能存在的连接。这样做使我们能够利用现有的单图形处理器版本*Suitor* 算法。尽管进行了上述简化，正如第5节所讨论的结果所示，解耦方法在最终预条件共轭梯度法的收敛性与预条件器设置的并行效率之间实现了良好的权衡。我们注意到，完成解耦*MWM* 计算后，由于我们的聚合体仅包含图形处理器上的局部未知量，聚合算法定义了一个分布式分块延拓算子（进而定义限制算子），其各个块对应不同的任务，可在本地构建。下一节将详细说明，延拓/限制矩阵的这种特殊结构也会影响完成AMG预条件器设置所需的后续计算，即并行稀疏矩阵-矩阵乘法 操作，这些操作用于可能需要的连续延拓算子乘积，以及用于计算更粗层矩阵所需的稀疏矩阵三重伽辽金乘积。

4.2 并行混合稀疏矩阵乘积

一般而言，稀疏矩阵-矩阵乘法 是多个科学计算应用中的基础构件，这些应用涉及稀疏矩阵和图，不仅限于线性求解器中的代数多重网格预条件器。近年来，在人工智能领域（例如通过将层因式分解为稀疏矩阵乘积来进行神经网络压缩 [Good- fellow(2016),Zachariadis(2020, Giffon(2021]）中，人们对在采用高吞吐量加速器的并行计算机上实现高效稀疏矩阵-矩阵乘法 内核表现出了浓厚兴趣。
稀疏矩阵-矩阵乘法 的主要难点在于，结果矩阵乘积的非零元素数量及其稀疏模式无法预先确定。通常会执行所谓的符号 阶段，计算结果矩阵中的非零元数量，并将实际数值计算推迟到后续的数值 阶段。对于*BootCMatchG*的单个GPU版本，我们采用了*nsparse* [Nagasaka(2017)] 软件包；事实证明，与英伟达 *cuSparse* 库提供的任何原语组合相比，它的效率高得多。然而，*nsparse* 方案有两个缺点： *i*) 它依赖于CUDA*shuffle* 原语的旧版行为； *ii*) 它假定两个矩阵都位于本地GPU上（不支持多GPU执行）。我们通过根据 CUDA-9 中引入的新定义简单更新 shuffle 原语，解决了第一个问题；第二个问题则值得更详细地讨论。

假设两个操作数均采用连续的行块布局，并分配给具有连续 MPI 进程的并行任务，则一种直接将乘积计算分配到各图形处理器的方法是：每个图形处理器计算乘积矩阵中相应的行块。例如，在仅包含两个图形处理器的最简单配置中，第一个图形处理器负责计算乘积矩阵前一半的行，第二个图形处理器负责计算后一半的行。下面我们描述计算乘积矩阵前一半行所需执行的步骤（即从第一个图形处理器的角度出发），但显然，同样的方法也可应用于任意行子集。每个图形处理器都拥有两个操作数中属于其行块的非零元；因此，为了完成其第一个矩阵各行与第二个矩阵各列的标量积计算，通常需要获得第二个矩阵中与其行索引范围以上或以下的行索引相对应的非零元，具体取决于第一个矩阵每一行的稀疏结构。在我们的示例中，第一个图形处理器仅拥有第二个矩阵每一列前一半的非零元。如果属于第一个矩阵相应子集的某一行，其非零元的列索引大于 $n/2$，其中 n 为总行数（我们假设 n 是偶数），则它需要这些非零元素来完成乘积。在下面这个由两个非常小的（4×4）矩阵组成的示例中（水平线表示两个图形处理器之间的分界），第一个图形处理器为了完成其

乘积部分，需要第二个图形处理器负责的最后一行中的元素 b_{41} 和 b_{44} .

$$\left[\begin{array}{cccc} a_{11} & 0 & 0 & a_{14} \\ 0 & a_{22} & 0 & a_{24} \\ \hline 0 & 0 & a_{33} & 0 \\ a_{41} & a_{42} & 0 & a_{44} \end{array}\right] * \left[\begin{array}{cccc} b_{11} & 0 & 0 & b_{14} \\ 0 & b_{22} & b_{23} & 0 \\ \hline 0 & b_{32} & b_{33} & 0 \\ b_{41} & 0 & 0 & b_{44} \end{array}\right] =$$

$$\left[\begin{array}{cccc} a_{11}b_{11}+a_{14}b_{41} & 0 & 0 & a_{11}b_{14}+a_{14}b_{44} \\ a_{24}b_{41} & a_{22}b_{22} & a_{22}b_{23} & a_{24}b_{44} \\ \hline \cdots & & & \\ \cdots & & & \end{array}\right]$$

在开始计算之前，我们会在每个图形处理器上交换完成乘积计算所需的所有数据，因此从 *nsparse* CUDA 内核的角度来看，该乘积仿佛完全是本地计算的。我们之所以这样选择，是因为 *nsparse* 的效率很高，以至于调用一次它比先计算乘积的本地部分、同时交换所需数据，再计算乘积剩余部分并将两部分结果相加更为高效。另一种方案可能因本地部分的计算与非本地部分所需数据的交换可以重叠而带来优势。然而，该方案需要执行两次*nsparse* 内核（采用两个不同的符号阶段），还需要额外执行一个内核来对两个部分乘积求和。总之，我们最终采用的多图形处理器 稀疏矩阵-矩阵乘法 流程由以下步骤组成，如算法 4 所示:

1. 每个图形处理器检查第二个矩阵中需要从其他图形处理器接收的行（第 2 至 6 行）。2. 中央处理器使用以下 MPI 集合通信原语交换每个所需行的非零元素数量信息：MPI 全收集、MPI 全交换、MPI 全交换变长（第 7 行）。3. 中央处理器在分配适当的内存缓冲区（其大小在上一步中确定）后，使用 MPI 点对点通信（MPI 点对点通信）非阻塞通信原语交换非局部行的索引及其对应的值（第 8 和 9 行）。4. 每个图形处理器通过将其已拥有的行与上一步中从其他图形处理器接收的行合并，构建其所需的第二个矩阵的行子集。为了尽量减少内存复制操作，我们引入了一种新的稀疏矩阵表示，称为分段 CSR。其思想是将矩阵的本地部分保存在其原始 CSR 数据结构中，并将来自其他任务的非本地部分存储在辅助 CSR 数据结构中。辅助 CSR 的数据结构通常应包含在矩阵乘法计算过程中确定使用本地部分还是远程部分所需的全部信息。对于本文采用的一维分解（即每个图形处理器拥有一个连续行块），只需知道哪些行子集属于该图形处理器的本地部分即可。该范围内的每个列索引（及其对应的值）通过原始 CSR 结构访问，而其他任何列索引（及其对应的值）则通过辅助 CSR 访问。原始 *nsparse* 已被修改，加入了这一简单的选择准则；与其带来的时间节省相比，该准则的成本完全可以忽略不计。5. 每个图形处理器执行其第一矩阵部分与上一步构建的第二矩阵适当子集之间的乘法（参见算法 4 第 11 行）。

我们注意到，在专门针对 AMG预条件器设置进行优化的实现中，步骤 1)、4) 和 5) 作为 GPU内核执行，而步骤 2) 和 3) 则在 CPU 上完成。作出这一设计选择，是为了能够让计算与数据通信重叠；事实上，当图形处理器执行*MWM* 计算时，步骤 2) 和 3) 可以在 CPU 上执行。正如我们将在第 5.2 节讨论的那样，当并行任务数量足够少时，这种重叠可以完全隐藏通信开销（见图7）。然而，随着任务数量增加，这种重叠将不再平衡，因为步骤 4) 和 5) 所需的时间相较于*MWM* 计算所需的时间变得占主导地位（后者始终保持不变，与任务数量无关，见第 4.1 节）。

我们最后注意到，由于第4.1节所述的解耦聚合方法，我们能够完全避免在预条件器层次结构的每个层级计算粗矩阵时所需的三矩阵 Galerkin 乘积中的一次 稀疏矩阵-矩阵乘法 操作所产生的通信。实际上，任务之间的数据通信仅在第一个乘积 $C = A^k P^k$ 中是必需的，而第二个 稀疏矩阵-矩阵乘法 乘积（$R^k C$）则完全是局部的，因为 R^k 的行不包含列索引对应于其他任务所拥有行的非零元，如图1所示。

算法 4并行稀疏矩阵-矩阵乘法

输入： 两个矩阵 ces A 和 B，每个进程仅拥有每个矩阵中的一个连续行块（从 h 到 k）.
输出： 矩阵 $C = AB$，每个进程仅计算其连续行块（从 h到 k）。

1： $rowsToReceive \leftarrow 0$**2：对于** $a_{ij} \in A_{local}$ **执行 3：如果** $j < h$**或** $j > k$ **则 4：** $rowsToReceive \leftarrow j$ **5：结束 if 6：结束 for 7：** $nnzPerRow \leftarrow MPI_{coll}(nnz(B(rowsToReceive)))$ **8：为结合** $nnzPerRow$ **和**$rowsToReceive$**中所含信息的** $B(rowsToReceive)$ **分配内存 9：** $B(rowsToReceive) \leftarrow MPI_{p2p}(B_{remote})$ **10：** $B_{segmented} \leftarrow merge(B_{local}, B(rowsToReceive))$ **11：** $C \leftarrow sparseProduct(A_{local}, B_{segmented})$ **12：返回** C

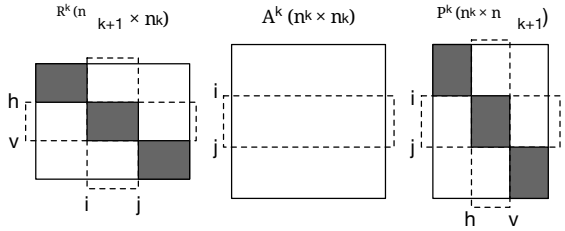

图1：解耦聚合中的三矩阵 Galerkin 乘积：矩阵 R^k 和 P^k 仅在灰色块内包含非零元。

4.3 并行混合稀疏矩阵-向量乘积

众所周知，稀疏矩阵-稠密向量乘积是线性求解器迭代方法中的关键操作。在*BootCMatchG*中，该操作主要用于预条件共轭梯度法和V循环。正如我们在 [Bernaschi(2020a)],中所述，根据矩阵的稀疏模式，我们采用不同的方案在图形处理器上执行该乘积。无论采用哪种具体方案，在使用多块 GPU 的分布式实现中，都需要在各图形处理器之间交换数据，才能完成该乘积。这个问题与第4.2节中描述的问题类似：任何一个图形处理器都可能需要其他图形处理器所拥有的向量中的某些条目。在下面的示例中，第一个图形处理器需要条目 x_4，而第二个图形处理器需要条目 x_1 和 x_2，才能完成相应的稀疏矩阵-向量乘法。

$$\left[\begin{array}{cccc} a_{11} & 0 & 0 & a_{14} \\ 0 & a_{22} & 0 & a_{24} \\ \hline 0 & 0 & a_{33} & 0 \\ a_{41} & a_{42} & 0 & a_{44} \end{array}\right] * \left[\begin{array}{c} x_1 \\ x_2 \\ \hline x_3 \\ x_4 \end{array}\right] = \left[\begin{array}{c} a_{11}x_1 + a_{14}x_4 \\ a_{22}x_2 + a_{24}x_4 \\ \hline a_{33}x_3 \\ a_{41}x_1 + a_{42}x_2 + a_{44}x_4 \end{array}\right]$$

显然，与稀疏矩阵-矩阵乘法相比，图形处理器之间需要交换的数据量非常有限。每个图形处理器最多需要向其他图形处理器发送 n/np 个条目（其中 np 是并行任务的数量），最多需要从其他图形处理器接收 $\frac{np-1}{np}n$ 个条目。然而，在稀疏矩阵-向量乘法中，无需执行符号阶段，因此，将乘积拆分为本地部分和非本地部分是合理的：本地部分仅涉及每个图形处理器已持有的向量条目，非本地部分则涉及每个图形处理器需要从其他图形处理器接收的向量条目。这种拆分使本地计算能够与必要向量条目的通信重叠，从而隐藏通信开销。尽管据我们所知，这种方法与 [Kreutzer(2012)、Naumov(2015)], 提出的方法类似，但由于实现起来并不简单，实际中并未广泛采用。要发挥实际优势，该方法需要若干机制： *i*) 异步内存复制； *ii*) CUDA流； *iii*) 非阻塞MPI原语。合理组合这些机制后，可以支持内存复制、中央处理器之间的数据通信以及图形处理器对本地数据的计算并发执行。我们的方案如算法5所示。重叠发生在乘积本地部分的执行（第9行）与通信（第10行至第12行之间的代码块）之间。随后，执行过程将

通过计算需要与其他任务通信的向量乘积条目来完成。与 [Kreutzer(2012)] 的方法相比，我们预先执行本地部分，利用在异步模式下运行第9行内核的机会；前提是输出向量 y 与输入向量 x 不同（如果输出向量必须替换输入向量，则需要预先复制该向量）。

算法5并行稀疏矩阵-向量乘法

输入： 一个矩阵 A 和一个向量 x，每个进程仅拥有矩阵 A中的一个连续行块（从 h 到 k）以及向量 x中对应的元素。

输出： 向量 $y = Ax$，每个进程仅计算其所负责的 x 连续元素（从 h 到 k ）.

1: $elToReceive \leftarrow 0$**2: 对于** $a_{ij} \in A_{local}$ **执行 3: 如果** $j < h$**或** $j > k$ **则 4:** $elToReceive \leftarrow j$**5: 结束如果 6: 结束循环 7:** x**(**$elToSend$**)** $\leftarrow MPI_{coll}$**(**x**(**$elToReceive$**)) 8: 为** x**(**$elToReceive$**) 和** x**(**$elToSend$**) 分配内存 9:** $y \leftarrow SparseMatrixVectorProduct$**(**$A_{local}$**,** x_{local}**) 10: 将** x**(**$elToSend$**) 复制到 CPU 11:** x**(**$elToReceiveCPU$**)** $\leftarrow MPI_{p2p}$**(**$x_{elToSendCPU}$**) 12: 将** x **(**$eToReceiveCPU$**) 复制到图形处理器 13:** $y \leftarrow SparseMatrixVectorProduct$**(**$A_{local}$**,** x_{remote}**) 14: 返回** y

在预条件共轭梯度法的迭代过程中，稀疏矩阵-向量乘法所涉及的稀疏矩阵不会改变，因此，第 11 行中用于标识待交换向量元素的索引始终保持不变。最后，通信操作所需的预处理（即选择图形处理器之间要交换的元素，对应于算法5中第 2 至 8 行的代码块）只需在求解阶段之前执行一次，并在每个 AMG 层次结构中的矩阵进行预条件共轭梯度法的所有迭代时重复使用。需要强调的是，只有涉及原始系统矩阵和 AMG 层次结构各层更粗层矩阵的稀疏矩阵-向量乘法才需要进行数据通信，而 V循环中涉及延拓算子和限制算子的所有稀疏矩阵-向量乘法均为纯本地操作，因为相关矩阵具有分块结构。

5 实验设置与结果

本节分析*BootCMatchG* 混合版本在知名基准测试用例上的并行性能。我们考虑求解三维泊松方程所需的代数方程组：

$$-\nabla \cdot (K\nabla u) = \mathbf{f}, \quad \text{in } [0,1]^3, \tag{1}$$

具有齐次狄利克雷边界条件， $K = 1$ 以及单位右端项。该问题通过经典的 7 点有限差分模板对左端算子（拉普拉斯算子）进行离散化，从而得到以下代数方程组：

$$\frac{6u_{i,j,k}}{h^2} - \frac{u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1}}{h^2} = f_{i,j,k} \equiv 1,$$

对于 i、 j、 $k = 1$、.....、 nd、以及 $h = 1/(nd+1)^2$,，并且对于 $u_{i,\ j,\ k} = u(ih$、 jh、 $kh) \equiv 0$ 在 $\partial[0,1]^3$ 上。上述线性方程组的系数矩阵为对称正定矩阵，是许多广泛使用的科学与工程应用的计算内核。与离散拉普拉斯算子相关的线性方程组在网络分析中也很常见，例如基于与网络相关联的拉普拉斯图的谱性质的社区检测算法 [D'Ambra(2019)]。我们关注分析代码的强可扩展性，即在线性方程组规模固定时，随着并行资源数量增加，求解该线性方程组的执行时间缩短的程度；同时也关注代码的弱可扩展性，即在线性方程组维数增加且并行资源同步增加时的性能。正如第4节所述，数据在 MPI任务及其相关右端项之间采用系统矩阵的行块分布，即根据 MPI 秩定义的顺序，将连续的行块分配给各个任务

。我们采用*BootCMatchG* 的混合版本来构建代数多重网格层次结构，并将最粗层矩阵的最大规模固定为 $40 \cdot nd$。同时将最大层数固定为40。我们将通过基本成对聚合计算得到的基本成对延拓算子两两组合，从而得到大小至多为8的聚合体，即每列至多包含8个条目的延拓算子。完成设置后，我们将 AMG预条件器应用为一次 V循环迭代，其中使用4次 ℓ^1–雅可比迭代作为前、后平滑器，而在最粗层使用同一平滑器进行20次迭代。

我们在瑞士国家超级计算中心运营的 Piz Daint[1], 上进行了实验。该系统基于 Cray XC40/XC50 架构，配备 5704 个混合计算节点（Intel Xeon E5-2690 v3，搭载 Nvidia Tesla P100 加速器）和 1813 个多核计算节点（Intel Xeon E5-2695 v4），使用采用 Dragonfly 网络拓扑的 Cray Aries 路由与通信 ASIC。测试时，我们在每个混合节点上分配 1 个 MPI 进程，并在节点中配置 1 个图形处理器。

我们通过考察所得 AMG 层次结构的算子复杂度 OPC 以及 $tsetup$（即层次结构设置的执行时间）来分析线性求解器的性能。此外，我们还讨论预条件共轭梯度法的迭代次数 it 、求解系统的执行时间 $tsolve$以及每次预条件共轭梯度法迭代的执行时间 $titer$ ，从算法和实现可扩展性两个角度刻画应用阶段。所有时间均以秒为单位。我们发现，AMG 层次结构的算子复杂度可以用于量化存储层次结构各级算子所需的内存需求，以及相关算子的应用成本。

PCG迭代过程中的V循环。它可以通过 OPC 的迭代测量 $\frac{\sum_{i=1}^{nl} \ell_i \ell_2^{nnz}}{\ell_1^{nnz}}$ 。其中nnz()是矩阵中非零元素的数量，请注意，当 ℓ_2 范数中的相对残差小于10^{-6}，或迭代次数达到设定的最大值1000时，我们将停止PCG迭代。

我们将BootCMatchG的混合版本（以下简称BCMG）与Nvidia AmgX的混合版本 [AmgX(2020)]进行了比较。AmgX提供了各种AMG预条件器，这些预条件器基于其他库（如hypre）中已经提供的不同知名粗化方法，并生成具有不同算子复杂度的AMG层次结构。为进行公平比较，我们选择了一个输入配置，该配置定义了基于未知量聚合的AMG层次结构，其算子复杂度与上述*BCMG* 预条件器相当。为此，我们修改了随该库分发的名为*PCG AGGREGATION JACOBI.json*的配置文件中的少数参数。所选的AmgX预条件器（以下简称*AMGX*）采用基于未知量局部匹配的解耦普通聚合方案，与我们的方法相同，但其匹配由一种简单的启发式方法驱动。这种启发式方法仅对M矩阵类 [Vassilevski(2008)], 有充分的理论认识，它通过刻画光滑误差分量来衡量未知量之间的连接强度；与我们对*BCMG*所采用的方法类似，我们将聚合体大小设为8。随后，在与*BCMG*相同的条件下应用所得预条件器：在PCG方法的每次迭代中，执行1次V循环，其中中间层执行4次雅可比前/后平滑迭代，最粗层执行20次 ℓ_1–雅可比迭代。在所有情况下，我们都使用针对*BCMG*的粗化停止准则所得到的相同层数，构建了AMG层次结构。

5.1 强可扩展性

我们首先讨论求解具有 $300^3 = 27 \times 10^6$ 个自由度（dofs）的线性系统时的性能表现：逐步增加并行节点数，每个节点配备 1 个图形处理器，最多扩展到 16 个节点。所有情况下，AMG 层次级数均为 5。图 2 展示了*BCMG* 和*AMGX*的算子复杂度和迭代次数。我们观察到，在所有情况下，*BCMG*生成的 AMG 层次结构复杂度都低于*AMGX*；*BCMG*的复杂度通常约为 1.14，而*AMGX* 的复杂度则处于区间 $[1.28, 1.34]$ 。这使得预条件子应用的内存占用和计算复杂度相较于*BCMG*增加了 12% 至 20%。另一方面，我们发现，尽管复杂度更高，*AMGX* 在求解阶段所需的线性迭代次数始终更多，虽然其迭代次数更加稳定。随着并行节点数增加，*BCMG* 的迭代次数出现预期中的波动，这是因为算法3中采用的解耦方法降低了粗矩阵的质量。并行节点数增加会导致非局部矩阵块增大，而这些矩阵块在局部 MWM 以及相应的块延拓算子/限制算子中均被忽略；这并非明显的缺陷，因为对于固定规模的问题，大幅增加并行资源并无益处，这是众所周知的阿姆达尔定律效应。*BCMG* 较低的算子复杂度和迭代次数带来了更短的求解时间：从单节点上的 5.14 秒降至 16 个节点上的 0.95 秒（见图 3），加速比约为 5.4，表现令人满意。*AMGX* 在 16 个节点上的加速比略高，约为 5.60，这主要是因为它在单节点上的求解时间非常长。这一表现凸显了*BCMG*在节点级别实现以及利用所有计算内核并行性方面的高效率

[1]https://www.cscs.ch/computers/piz-daint/

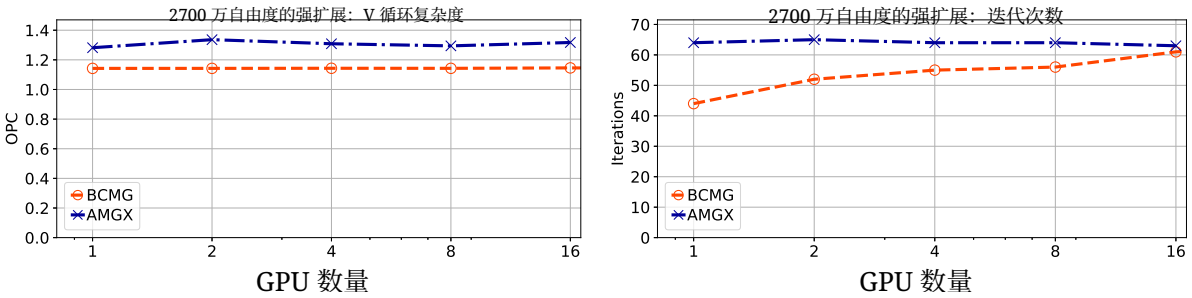

图 2：强可扩展性：算子复杂度和 PCG 迭代次数

求解阶段也证实了这一点：每次迭代时间表现出色，在 16 个节点上实现了约 7.5 的加速比，而*AMGX* 相应的加速比约为 5.53。为求完整起见，我们在图 4 中

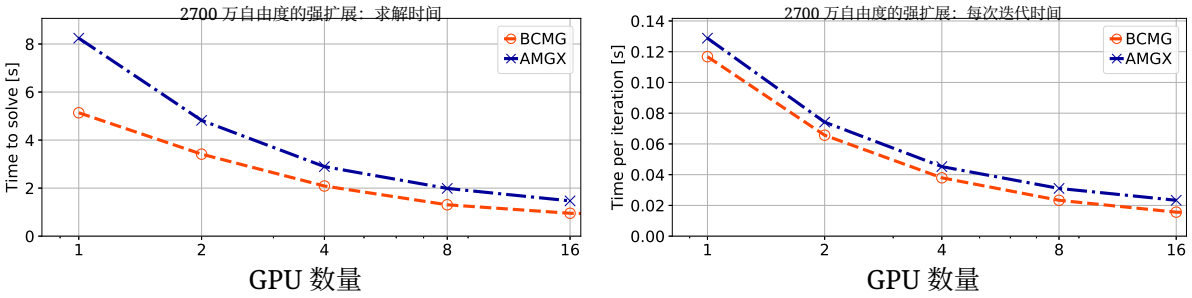

图 3：强可扩展性：求解时间和每次迭代求解时间

还展示了预条件器设置的执行时间，以及由求解时间与预条件器设置时间之和得到的总执行时间。正如预期，由于算法方法截然不同，*BCMG* 的设置需要更长时间。实际上，我们的算法依赖于用于粗化的 *MWM* 内核，并通过显式的三重稀疏矩阵乘积生成 AMG 层次结构。相比之下，*AMGX* 中实现的简单聚合由一种基于参数的启发式方法驱动，该方法根据未知量之间连接强度的度量来构建二元延拓算子，因此每一层更粗层矩阵的计算只需进行简单的局部求和。因而，整个过程具有令人尴尬的并行性 [Naumov(2015)]。尽管如此，在 16 个并行节点上，*BCMG* 仍可获得约 3.7 的加速比，并且*BCMG*的总执行时间始终更短。

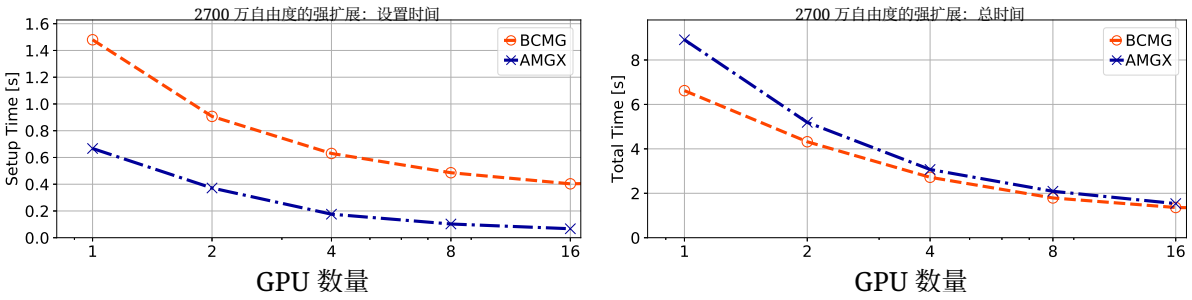

图 4：强可扩展性：预条件器设置时间和总时间

5.2 弱可扩展性

在本节中，我们分析求解器的弱可扩展性；每个节点上的固定矩阵规模为 $130^3 \approx 2.2 \times 10^6$，节点数从 1 增至 100。因此，我们求解的最大问题规模为 $\approx 2.2 \times 10^8$ 个自由度。在这种情况下，我们观察到

随着并行节点数增加，AMG层次结构的层数也随之增加：1 个节点时为 4 层，2 至 16 个节点时为 5 层，32 至 100 个节点时则为 6 层。这是由于我们为粗化算法设定的停止准则旨在实现层次结构层数与最粗层矩阵规模之间的良好权衡，从而在保持并行效率的同时，确保每次应用预条件器时都能获得最粗层系统的足够精确解。

在图 5 中，我们展示了*BCMG* 与*AMGX*的算子复杂度和迭代次数。可以看出，在所有情况下， OPC 对于*BCMG* 都小于*AMGX* ，并且对于所有节点数，其值均约为 1.14。这证实，我们的 AMG 层次结构通常比*AMGX* 的开销更低，无论是在内存占用还是在 V循环应用方面。另一方面，尽管复杂度更高，*AMGX* 在求解阶段所需的迭代次数始终多于*BCMG*。两种求解器的迭代次数最初均有所增加，之后表现趋于相似且更加稳定；但从 1 个节点增加到 100 个节点时，*AMGX* 的迭代次数增加了 $\sim 50\%$，而*BCMG* 仅增加约 36%，这表明*BCMG* 生成的层次结构具有更好的算法可扩展性。求解时间也证实了我们的预条件器质量更优且复杂度更低（见图 6）。*BCMG* 的求解时间始终显著短于*AMGX*。在许多情况下，*AMGX* 所需的求解时间是*BCMG*的两倍。随着资源数量和问题规模的增加，*BCMG* 求解时间表现得更加稳定，这是我们的求解器具有更好可扩展性特性的结果。最后，观察每次迭代时间，正如算子复杂度所反映的那样，我们发现*BCMG* 的耗时始终短于*AMGX*。另一方面，随着节点数量增加，*BCMG* 的增长比例也更小，这表明预条件器应用阶段的所有计算内核都得到了高效实现。我们观察到

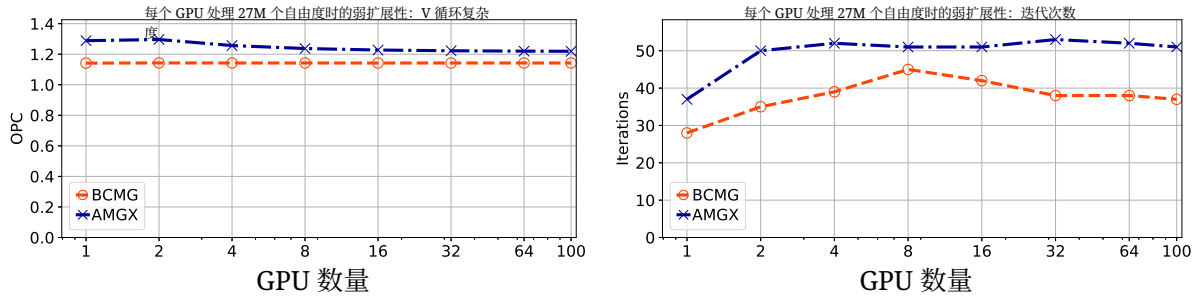

图5：弱可扩展性：算子复杂度和 PCG 迭代次数

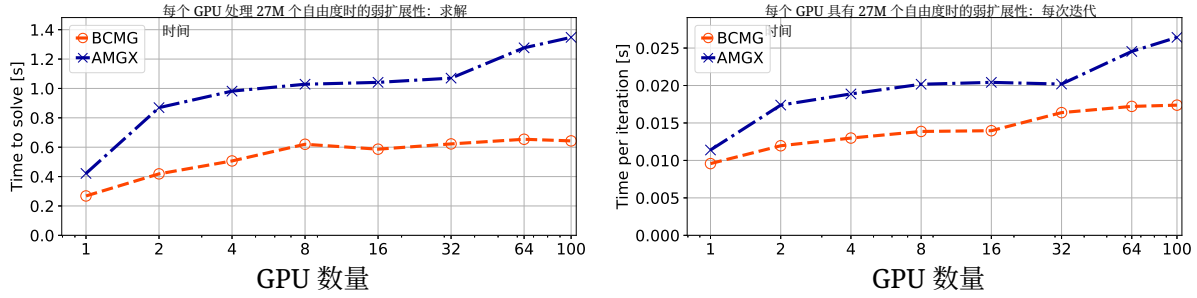

图6：弱可扩展性：求解时间和每次迭代求解时间

图7显示，正如预期，*AMGX* 的设置时间比*BCMG* 短，这是由于两者实现的算法大不相同。另一方面，我们较长的设置时间与显著较短的求解时间形成了很好的平衡。此外，我们指出，在稀疏线性求解器中，只要在求解阶段获得良好的收敛率和并行效率，较长的预条件器设置时间通常是可以接受的。事实上，在时变和/或非线性应用中，通常需要使用相同的系数矩阵求解多个线性系统，或者求解一系列矩阵缓慢变化、因而可以重复使用预条件器的线性系统。因此，较长的设置时间会分摊到多个求解步骤中。

为了更深入地分析设置阶段的性能，图7报告了两个主要计算内核（即*MWM* 和稀疏矩阵-矩阵乘法）所耗时间的细分情况。同时还展示了稀疏矩阵-矩阵乘法并行计算所需数据通信产生的测量开销（见图7中的SpMMComm柱）。我们注意到

，如第4.2节所述，这些数据通信由CPU负责，而MWM计算则在图形处理器上运行，以便实现MPI数据通信与计算的重叠。我们看到，SpMMComm柱所示的数据通信额外耗时仅在GPU数量大于16时才测得，而且正如预期，随着GPU数量增加，该耗时也会增加。值得注意的是，当问题规模和并行节点数量增加时，正如我们已经观察到的，AMG层次结构的层数会增加，因此*MWM* 和稀疏矩阵-矩阵乘法计算量也会增加，而更粗层矩阵的维数则会减小。这就是稀疏矩阵-矩阵乘法内核执行时间及相关通信时间增加的原因。

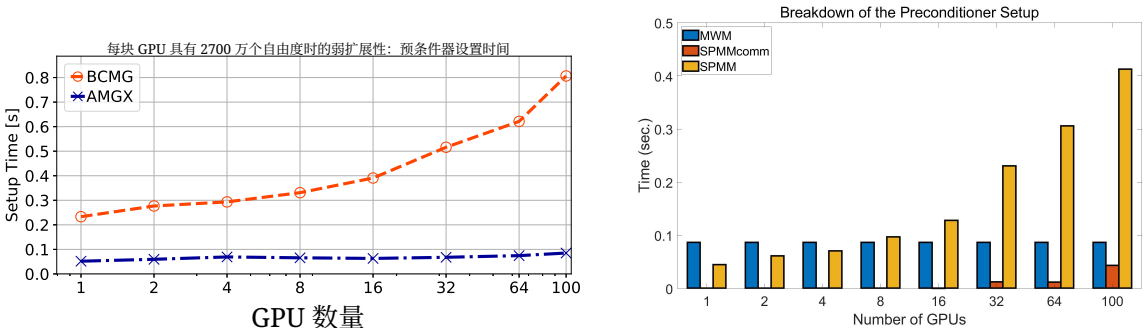

图7：弱可扩展性：预条件器设置时间及设置时间分解

6 结论

本文讨论了用于在多块 GPU 上求解大型稀疏线性方程组的软件库的混合MPI-CUDA实现的设计选择和结果。在算法和实现层面，该软件库的设计均以充分利用线性求解器每个计算内核中的并行性为驱动。我们在可能的情况下采用了通过重叠数据通信与计算来隐藏通信的最佳实践。弱可扩展性结果以及与最先进软件的比较表明，我们的软件包非常适合在配备多块 GPU 的异构并行计算机上求解超大规模方程组。未来的工作包括扩展该软件库，加入新的、稳健且高度并行的平滑器和最粗层求解器。该软件库的源代码可在 *https://github.com/bootcmatch/BootCMatchGX*获取。

7 致谢

本研究部分由欧盟委员会通过 EuroHPC项目资助，项目名称为：面向超大规模计算的欧洲高性能计算应用的极端规模技术与加速器（*TEXTAROSSA*），资助来自地平线2020研究与创新计划，编号：956831；另获 INdAM 通过 GNCS项目资助，项目名称为：*InSANE*：后冯·诺依曼时代稀疏算法研究CUP：E55F22000270001。

我们衷心　感谢 PRACE 授予我们使用瑞士苏黎世联邦理工学院/CSCS 的 Piz Daint 的权限。
作者还感谢 IAC-CNR 的博士后研究员 Dario Pasquini 协助实现了本研究中使用的一些图形处理器内核。

参考文献

[Shalf(2019)] Shalf, J. 超越摩尔定律的计算未来。《哲学学报：数学、物理与工程科学》**2019**，*378: 20190061*。[Top500(2022)] Top 500。榜单。在线查阅：https://www.top500.org/lists/top500/2022/11/

[Anzt(2019)] Anzt H 等。为百亿亿次计算做好准备的稀疏求解器。《哲学学报：数学、物理与工程科学》**2019**，*378:20190053*。

[Bernaschi(2020a)] Bernaschi M, D'Ambra P, Pasquini D. 面向图形处理器的相容加权匹配代数多重网格。并行计算 **2020**,*92: 102599*。

[Bernaschi(2020b)] Bernaschi M, D’Ambra P, Pasquini D. BootCMatchG：面向图形处理器的自适应代数多重网格线性求解器。Software Impacts 2020, 6.[Saad(2003)] Saad, Y. 稀疏线性方程组的迭代方法。第2版；SIAM：美国费城，2003。[AmgX(2020)]英伟达，代数多重网格求解器（AmgX）库，rel.2.1，2020。https://github.com/NVIDIA/AMGX.[Naumov(2015)] Naumov M, Arsaev M, Castonguay P, Cohen J, Demouth J, Eaton J, Layton S, Markovskiy N, Reguly I, Sakharnykh N, Sellappan V, Strzodka R. AmgX：用于GPU加速代数多重网格和预条件迭代方法的库，SIAM科学计算杂志，2015，37，S602—S626。[Falgout(2006)] Falgout RD, Jones JE, Yang UM. hypre的设计与实现：一个并行高性能预条件器库。载于《并行计算机上的偏微分方程数值求解》；Bruaset AM, Tveito A，编；2006，51 Lect. Notes Comput. Sci. Eng.，Springer，柏林，267—294。[Falgout(2021)] Falgout RD, Li R, Sj¨ ogreen B, Wang L, Yang UM. 将 hypre 移植到异构计算机架构：策略与经验。并行计算，2021，108，102840。[Brodtkorb(2010)] Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, Storaasli OO. 异构计算的最新进展。Scientific Programming 2010，18，1–33。[Filippone(2017)] Filippone S, Cardellini V, Barbieri D, Fanfarillo A. GPGPU上的稀疏矩阵-向量乘法。ACM Trans. Math. Softw. 2017，43:30。[D’Ambra(2016)] D’Ambra P, Filippone S. 一种用于图形处理器上高性能图像分割的并行广义松弛方法。J. Comput. Appl. Math. 2016，293，35–44。[D’Ambra(2002)] D’Ambra P, Danelutto M, di Serafino D, Lapegna M. 并行与分布式应用的先进环境：现状概述。载于《并行计算：并行与分布式计算先进环境特刊》；D’Ambra P, Danelutto M, di Serafino D，编；2002，28，1637–1662。[Heroux(2013)] Heroux MA, Dongarra JJ. 探索高性能计算系统的新排名指标。Sandia National Lab. Tech. Rep., SAND2013-4744, 2013。[Petsc(2020)] PETSc用户手册。http://www.mcs.anl.gov/petsc。技术报告 ANL-95/11 - Revision 731 3.111，2020，阿贡国家实验室。[Heroux(2005)] Heroux M, et al. Trilinos项目概述，ACM Trans. Math. Softw. 2005，13，397—423。[Notay(2015)] Notay Y, Napov A. 离散泊松型问题的大规模并行求解器，J. Comput. Physiscs 2015，281，237–250。[Xu(2017)] Xu J, Zikatanov L. 代数多重网格方法，Acta Numer. 2017，26，591—721。[Vassilevski(2008)] Vassilevski PS. 多层块分解预条件器：用于求解有限元方程的基于矩阵的分析与算法；Springer：美国纽约，2008。[Brannick (2013)] Brannick J, Chen Y, Hu X, Zikatanov LT. 图形处理器上的并行非平滑聚合代数多重网格算法。载于《偏微分方程的数值解：理论、算法及其应用》。2013，Iliev OP, Margenov SD, Minev PD, Vassilevski PS, Zikatanov LT 编，Springer，81–102。[Yamazaki (2020)] Yamazaki I, Rajamanickam S, Ellingwood N. 面向众核架构的高性能可移植超节点稀疏三角求解器。载于第49届国际并行处理会议 - ICPP。2020，加拿大埃德蒙顿。计算机协会，70。[D’Ambra(2013)] D’Ambra P, Vassilevski PS. 基于相容加权匹配粗化的自适应AMG，Computing and Visualization in Science 2013，16，59—76。[D’Ambra(2018)] D’Ambra P, Filippone S, Vassilevski PS. BootCMatch：一种基于图加权匹配的引导式 AMG 软件包，ACM Trans. Math. Softw. 2018，44，39。[D’Ambra(2019)] D’Ambra P, Vassilevski PS. 改进基于聚合的自适应AMG的求解时间，Numerical Linear Algebra with Applications 2019，26，e2269。[Goodfellow(2016)]Goodfellow I., Bengio Y., Courville A., 深度学习；麻省理工学院出版社，2016，http://www.deeplearningbook.org。[Zachariadis(2020] Zachariadis O., Satpute N., G´omez-Luna J., Olivares J. 利用GPU张量核心加速稀疏矩阵-矩阵乘法，Computers & Electrical Engineering 2020，88；106848。[Giffon(2021] Giffon L., Ayache S., Kadri H., Arti´eres T., Sicre R. PSM-nets：利用稀疏矩阵乘积压缩神经网络，HAL Id: hal-03151539，https://hal.archives-ouvertes.fr/hal-03151539。

[Nagasaka(2017)] Nagasaka Y., Nukada A., Matsuoka S. 面向 Nvidia Pascal GPU 的高性能、节省内存的稀疏通用矩阵-矩阵乘法，*IEEE*第*46*届并行处理国际会议**2017**，英国布里斯托尔，101–110。[Pothen(2019)] Pothen A，Ferdous SM，Manne F. 组合科学计算中的近似算法，《数值学报》**2019**，*28*，541–663。[Naim(2015] Naim M，Manne F，Halappanavar M，Tumeo A，Langguth J. 在 Nvidia Kepler K40 上优化近似加权匹配，*IEEE*第*22*届高性能计算年度国际会议**2015**，印度班加罗尔，105–114。[Acer(2021)] Acer S，Azad A，Boman EG，Buluc. A，Devine KD，Ferdous SM，Gawande N，Ghosh S，Halappanavar M，Kalyanaraman A，Khan A，Minutoli M，Pothen A，Rajamanickam S，Selvitopi O，Tallent NR，Tumeo A. EXAGRAPH：用于支持百亿亿次级应用的图与组合方法，《国际高性能计算应用杂志》**2021**，*35*，553—571。[Tuminaro(2000)] Tuminaro RS，Tong C. 并行平滑聚合多重网格：大规模并行计算机上的聚合策略。载于*ACM/IEEE*超级计算会议论文集。**2000**，CDROM，得克萨斯州达拉斯。[D'Ambra(2010)] D'Ambra P，di Serafino D，Filippone S. MLD2P4：一个采用 Fortran 95 编写的并行代数多层域分解预条件器库。*ACM* 数学软件汇刊。**2010**，*37*，7–23。[Kreutzer(2012)] Kreutzer M，Hager G，Wellein G，Fehske H，Basermann A，Bishop AR. GPGPU集群上的稀疏矩阵-向量乘法：一种新的存储格式和可扩展实现。*2012 IEEE*第*26*届国际并行与分布式处理研讨会&博士论坛**2012**，1696–1702。[D'Ambra(2019)] D'Ambra P，Cutillo L，Vassilevski PS. 用于谱聚类的引导式 AMG，《计算与数学方法》**2019**，*1, e1020*。

8 附录

A 如何使用求解器

下面简要说明用户如何在其应用程序中使用我们的线性求解器。算法6展示了一个使用示例的伪代码。我们提醒感兴趣的读者，该库的源代码可在*https://github.com/bootcmatch/BootCMatchGX*获取。代码仓库包含编译和运行我们的求解器所需的全部信息。代码仓库中还提供了一个示例程序，并对命令行参数进行了说明。

算法6使用示例

```
    准备输入参数 A和 b
    输入矩阵 A 以 CSR 格式存储
 1: CSR *A ← 生成或读取局部矩阵
 2: vector⟨vtype⟩ *b ← 生成或读取局部右端项
    读取配置参数
 3: 参数 p ← 读取配置文件
    计算解
 4: 向量⟨vtype⟩ *x ← bcmgx(A, b, p, precflag)
```

算法 7预条件柔性共轭梯度

1: 给定 u_0 并设置
$r_0 = b - Au_0$ 2: $w_0 = d_0 = Br_0$
3: $v_0 = q_0 = Aw_0$ 4 $\alpha_0 = w_0^T r_0$:
5
: $\beta_0 = \rho_0 = w_0^T v_0$ 6:
$u_1 = u_0 + \alpha_0/\rho_0 d_0$ 7:
$r_1 = r_0 - \alpha_0/\rho_0 q_0$ 8: **对于** $i = 1,$
... **执行** 9: $w_i = Br_i$ 10: $v_i = Aw_i$
11: $\alpha_i = w_i^T r_i$ 12: $\beta_i = w_i^T v_i$ 13:
$\gamma_i = w_i^T q_{i-1}$ 14:
$\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$ 15:
$d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$ 16:
$u_{i+1} = u_i + \alpha_i/\rho_i d_i$ 17:
$q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$ 18:
$r_{i+1} = r_i - \alpha_i/\rho_i q_i$ 19: **结束循环**

使用求解器的第一步是准备输入参数， A 和 b。该库提供了一些函数，用于读取并分发输入矩阵 A ，或在每个图形处理器上生成该矩阵（第 1 行）。然后，从配置文件中读取求解器参数（第 3 行），其位置可以通过命令行指定。该库代码仓库中包含一个配置文件示例，并对每个参数作了简要说明。最后，函数*bcmgx* （第 4 行）会在需要时设置预条件器，并通过应用算法 7 计算解。未预条件的 CG 方法对应于 $B = I$取值为 $B = I$的算法 7，其中 I 是单位矩阵，因此，只需对实现算法 7 的代码稍作修改即可实现。在示例程序中，可以通过命令行参数 -p 设置未预条件的 CG；它对应*bcmgx*的输入参数*precflag* 取值为 0。本工作所讨论实验使用的配置文件如清单 1 所示。

```
1  NONE        % 右端项文件；不存在时为 NONE
```

```
2 NONE % 若不存在则为 sol 文件 NONE3 0 % 引导式_类型：
0 乘法型，4 % 1 对称多重型，2 加法型；5 % 注意：这是组合规则，
6 % 当应用引导式方法且设置了7 % 多于 1 个 AMG 层次结构时适用
8 1 % bootstrap AMG 中的最大_hrc，即最大9 % 层次结构数；注
意：此处设为 1 表示10 % 单一 AMG 组件11 0.8 % 组合 AMG 的目
标收敛率；12 % 注意：若达到最大_hrc标准，通常13 % 无法达到该
收敛率14 3 % 匹配类型：3 Suitor15 2 % aggrsweeps；成对聚合
16 % 步骤。0 表示成对；17 % 1 表示双对，等等.....18 0 % aggr_
类型；0 表示未平滑，19 % 1 表示平滑（尚不支持）20 39 % 最大_
层数；单一层次结构21 % 构建的最大层数22 0 % 循环_类型：0-V
循环，1-H 循环，23 % 2-W 循环24 4 % 粗网格_求解器：0 雅可比，
1 FGS/BGS，25 % 3 对称高斯-赛德尔，4 l1-雅可比26 4 % 松弛_类
型：0 雅可比，1 FGS/BGS，27 % 3 对称高斯-赛德尔，4 l1-雅可比
28 20 % 粗网格松弛次数_29 4 % 预松弛扫描次数_30 4 % 后松弛扫
描次数_31 1000 % 迭代次数上限32 1.e -6 % 相对容差33 2000000
% 内存分配大小__
```

清单 1：配置文件

B 与相关软件的比较

下面给出了在 Marconi-100 超级计算机的图形处理器上获得的对比结果。所有代码均使用 gnu/8.4.0 工具套件和用于 GPU 内核的 CUDA 11.0 进行编译，并链接到 spectrum mpi/10.4.0 和 openblas/0.3.9 库。我们将本文的 *BCMG* 求解器与 AmgX 的基于聚合的方法进行了比较，该方法已在论文中使用（第 5.1 节和第 5.2 节），下文称为 *AMGX-A*；同时还与 AmgX 对经典 AMG 方法的实现进行了比较，该实现详见*AMGX* 参考手册第 2.5 节，手册地址为 *https://github.com/NVIDIA/AMGX/blob/main/doc/*，下文称为 *AMGX-C*。我们使用了 Marconi 100 上已安装的 NVIDIA AmgX 库（版本 2.2.0）。此外，我们还在 Marconi 100 上安装了 hypre 库（版本 2.26.0），该库可从*https://github.com/hypre-space/hypre*获取。为进行公平比较，在所有可用的代数多重网格预条件器中，我们选用了基于并行极大独立集（PMIS）的经典 AMG 方法，因为目前只有该粗化方法的完整设置过程支持在图形处理器上运行。粗化过程采用库的默认配置，以建立 AMG 层次结构。下文将基于 hypre 的求解器称为 *HYPRE*。在所有测试中，论文中所述的 V循环配置均应用于预条件共轭梯度法的每次迭代。因此，我们在中间层次执行 4 次 ℓ_1–雅可比预/后平滑迭代，并在最粗层次执行 20 次 ℓ_1–雅可比迭代。此外，当 ℓ_2 范数中的相对残差小于 10^{-6}，或迭代次数达到设定的最大值 1000 时，我们停止 PCG 迭代。由于许可限制，我们最多只能使用 Marconi 100 的 32 个图形处理器，并对求解器进行了弱可扩展性分析；每个图形处理器对应的固定矩阵规模为 $130^3 \approx 2.2 \times 10^6$。需要指出的是，*HYPRE* 在使用 8 个图形处理器时无法运行，并会因 MPI 错误退出。图中使用黑色方块表示该曲线是由前一个数值进行常数插值得到的。在图 8 中，我们展示了不同求解器的算子复杂度和迭代次数。可以看到，在所有情况下，OPC 对于 *BCMG*都更小，这表明我们的 AMG 层次结构通常比其他预条件器更经济，无论从内存占用还是从应用 V循环所需的浮点运算次数来看都是如此。与论文中已经观察到的结果一致，*AMGX-A*具有相近的算子复杂度；而*AMGX-C* 和*HYPRE*的算子复杂度都非常高

这是由于它们在 AMG 粗化过程中建立了更稠密的传递算子。*AMGX-C* 和*HYPRE* 较高的算子复杂度一方面带来了很好的算法可扩展性，这体现在随着图形处理器数量增加，迭代次数较少且非常稳定；另一方面也导致更长的预条件器设置时间，如图 10（上）所示。迭代次数的减少使*AMGX-C* 和*HYPRE* 这两种经典 AMG 方法相较于基于聚合的 AMG 方法具有更短的求解时间，但代价是极大的内存占用。然而，我们观察到*BCMG* 的求解时间与*AMGX-C* 相当，尽管其复杂度更低。这证明了我们的预条件器以及整体实现流程的有效性。最后，从图 10（下）可以看出，*BCMG* 始终能够获得最佳的总时间。

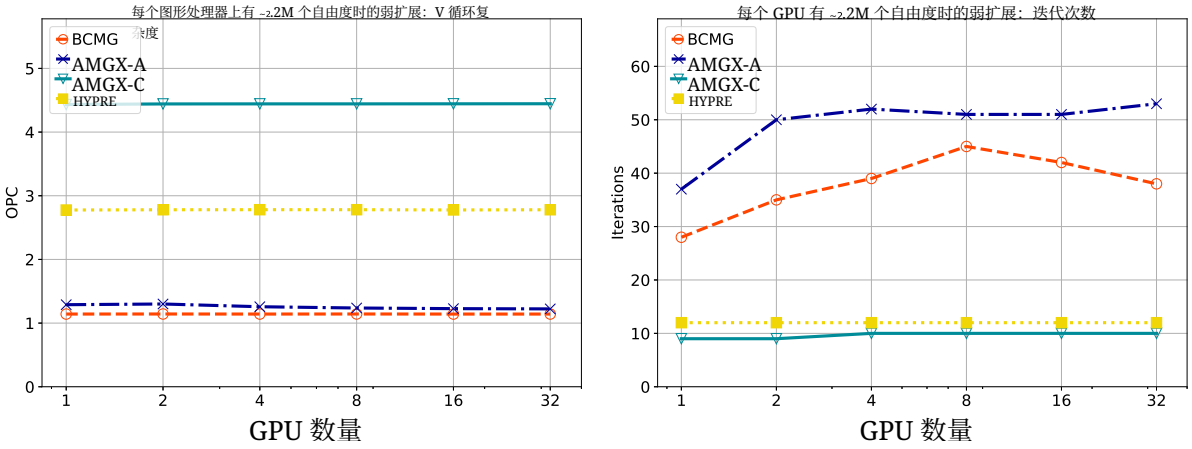

图 8：弱可扩展性：算子复杂度（上）和 PCG 迭代次数（下）

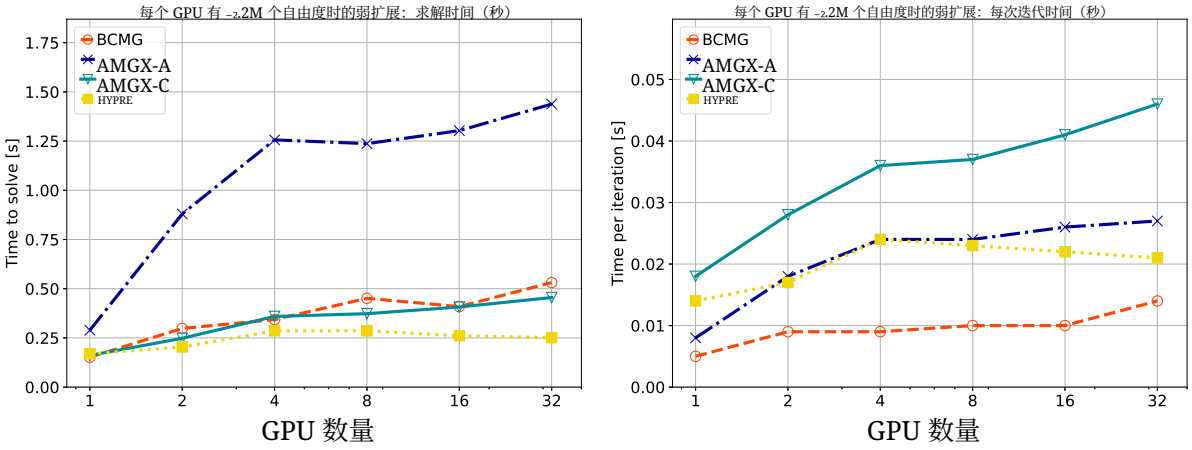

图 9：弱可扩展性：求解时间（上）和每次迭代求解时间（下）

C 示例测试和配置文件：AmgX 方法

本研究中的实验使用 AMGX 发行版示例文件夹中的示例程序 *amgx mpi poisson7* 运行。该示例程序会生成论文第 5 节所述的数据集，并求解相关方程组。运行 *AMGXA* 方法和 *AMGXC* 方法所需的配置文件，分别是对随库分发的 *PCG AGGREGATIONJACOBI.json* 文件（清单 2）和*PCG CLASSICAL V JACOBI.json* 文件（清单 3）的修改版本。为便于复现，我们在此列出这两个文件。

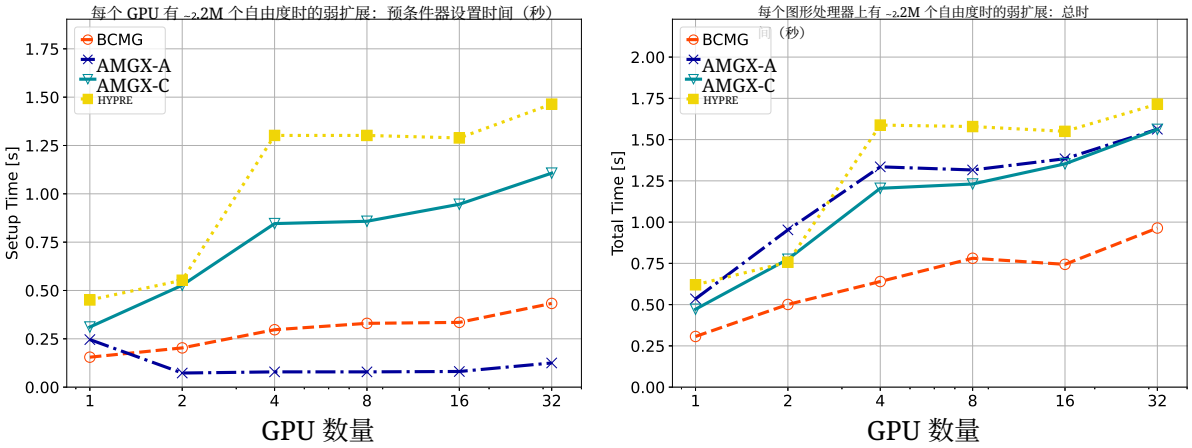

图10：弱可扩展性：预条件器设置时间（上）和总时间（下）

```
{
  "配置_版本": 2,
  "确定性_标志": 1,
  "求解器": {
      "预条件器": {
          "打印_网格_统计信息": 1,
          "算法": "聚合",
          "打印_可视_数据": 0,
          "选择器": "大小_8",
          "求解器": "代数多重网格",
          "平滑器": {
              "作用域": "雅可比",
              "求解器": "雅可比_L1",
              "监视器_残差": 0,
              "打印_求解_统计信息": 0,
              "松弛_因子": 1
          },
          "打印_求解_统计信息": 0,
          "预扫描次数": 4,
          "粗层_求解器": "雅可比_L1",
          "最粗层_次平滑": 20,
          "最大_迭代次数": 1,
          "监控_残差": 0,
          "存储_残差_历史记录": 0,
          "作用域": "代数多重网格",
          "后平滑次数": 4,
          "循环": "V"
      },
      "求解器": "预条件共轭梯度法",
      "打印_求解_统计信息": 1,
      "获取_计时信息": 1,
      "最大_迭代次数": 1000,
      "监控器_残差": 1,
      "收敛": "相对_初始",
      "作用域": "主作用域",
      "容差": 1e-06,
      "范数": "L2"
  }
}
```

清单 2：修改后的 PCG AGGREGATION JACOBI.json

```
{
```

2 " config_version ": 2 , 3 " 求解器 ": {4 " 预条件器 ": {5 " print_grid_stats ": 1 , 6 " print_vis_data ": 0 , 7 " 求解器 ": " 代数多重网格 " , 8 " 平滑器 ": {9 " 作用域 ": " 雅可比 ", 10 " 求解器 ": " JACOBI_L1 " , 11 " monitor_residual ": 0 , 12 " print_solve_stats ": 0 13 }, 14 " print_solve_stats ": 0 , 15 " 预扫描次数 ": 4 , 16 " 插值算子 ": " D2 ", 17 " coarse_solver ": " JACOBI_L1 " , 18 " coarsest_sweeps " : 20 , 19 " max_iters ": 1 , 20 " monitor_residual ": 0 , 21 " store_res_history ": 0 , 22 " 作用域 ": " amg " , 23 " 循环 ": "V" , 24 " 后平滑次数 ": 425 }, 26 " 求解器 ": " 预条件共轭梯度法 ", 27 " print_solve_stats ": 1 , 28 " obtain_timings ": 1, 29 " max_iters ": 1000 , 30 " monitor_residual ": 1, 31 " 收敛 ": " ABSOLUTE ", 32 " 作用域 ": " 主作用域 " , 33 " 容差 ": 1e -06 , 34 " norm ": " L2 " 35 }36 }

清单 3：修改后的 PCG CLASSICAL V JACOBI.json

D hypre 的配置行

使用随库分发的 *ij* 测试程序对 hypre 进行了实验。在这种情况下，我们不需要任何配置文件，运行时使用了以下选项：*-solver1 -rlx down18-rlx up18 -ns 4-pmis*。