用于射电天文学成像数据的高性能 W 堆叠：一种并行加速解决方案

Claudio Gheller，[1]⋆ Giuliano Taffoni[2] 和 David Goz，[2]1意

大利国家天体物理研究所射电天文研究所，*Via Gobetti 101*，*40121* 博洛尼亚，意大利[2]意大利国家天体物理研究所的里雅斯特天文台，*via GBTiepolo 11*，*34143* 的里雅斯特，意大利

接收日期 XXX。收到日期 YYY；原始形式为 ZZZ

arXiv:2301.06061v1 [astro-ph.IM] 15 Jan 2023

摘要

当前及即将投入使用的射电干涉仪预计将产生规模日益增大的数据，这些数据需要经过处理，才能通过成像生成相应的天空亮度分布。这构成了一项重大的计算挑战，尤其是在处理大视场和/或高分辨率观测数据时。我们研究了采用现代高性能计算系统来处理成像中的网格化、快速傅里叶变换和 w 校正，并结合并行与加速解决方案。我们证明，所开发的代码能够支持大小与可用硬件相匹配的任意数据集和图像，并可高效扩展至数千个核心或数百个图形处理器；即使生成的图像达到数十亿乃至数百亿像素，求解时间仍可控制在一小时以内。此外，我们将可移植性作为首要目标，既注重在不同计算平台上的易用性，也注重性能。本文展示的结果是在两种不同的先进高性能计算架构上取得的。

关键词：技术：图像　　处理——软件：数据分析——技术：干涉测量——高性能计算　　ng

1 引言

未来十年，现有及即将投入使用的射电干涉仪，例如低频阵列（LOFAR，van Haarlem et al. 2013）、MeerKAT（Jonas & MeerKAT Team 2016）、默奇森宽视场阵列（MWA，Mitchellet al. 2010）、澳大利亚平方公里阵列探路者（ASKAP，Johnston et al. 2007），以及平方公里阵列（SKA 中频阵列和低频阵列）1，将产生海量数据。这不仅为科学家带来宝贵机遇，也构成了重大的技术挑战。采用传统方法将难以管理预期的数据量。数据必须存储在专用设施中，以提供所需容量和最高性能。相应的数据处理必须在数据所在地进行，并充分利用可用的高性能计算（HPC）资源。数据约简和成像软件工具必须加以改造，甚至彻底重新设计，才能高效地大规模运行。

在现代高性能计算系统中，性能正通过众核和加速计算（例如基于图形处理器的计算）来实现，而当前趋势表明，某种形式的异构计算将继续在新兴架构中占据主导地位。因此，充分利用新的异构和众核解决方案，对于实现最佳性能至关重要。另一方面，随着观测数据的规模和复杂性不断增加，科学家能够够利用新兴高性能计算环境中的所有可用硬件，也变得至关重要。然而，由于混合硬件计算带来的挑战以及不断提高的架构并行性，利用这些新型混合架构并非易事。因此，需要新的算法以及数值和计算解决方案。

在射电干涉测量中，数据处理通常分多个步骤进行（且往往会迭代重复），其依据是 Hamaker 等人（1996）、Rau 等人（2009）和 Smirnov（2011a,b）的研究。首先，校准步骤旨在估计并校正随时间、频率、天线和方向变化的仪器误差（Bhatnagar 等人，2008；Cornwell 等人，2008；Tasse 等人，2013；Bhatnagar 等人，2013；Jagannathan 等人，2017；Bhatnagar 和 Cornwell，2017）。有多种软件工具可用于校准，例如 DPPP（van Diepen 等人，2018）、CUBICAL（Kenyon 等人，2018）或 KillMS[2]。接下来是成像，即将经过校准的可见度进行傅里叶变换以生成图像的过程（例如 DDFacet，它同时也能进行校准（Tasse 等，2018），以及 WSClean（Offringa 等人，2014））。随后进行反卷积（参见 Cornwell，1999），以校正因傅里叶平面采样不完整而导致的图像失真。反卷积通常采用成熟且广泛使用的 Clean算法及其不同变体（Högbom，1974；其中最常用的包括 Clark，1980；Schwab，1984），也可采用其他方法，例如最大熵模型（Cornwell 和 Evans，1985）、MORESANE模型（Dabbech 等人，2015），以及在优化理论背景下提出的稀疏性平均重加权分析（SARA）系列反卷积算法，

⋆ 电子邮箱：claudio.gheller@gmail.com[1]
https://www.skatelescope.org/

[2] https://github.com/saopicc/killMS

© 2015 作者

这些算法在 Puri-PSI 库中实现，已证明能够兼顾重建过程的精度和可扩展性（Carrillo 等人，2012年、2014年；Onose 等人，2016年；Pratley 等人，2018年；Thouvenin 等人，2020年、2022年），其中还包括一种将校准与反卷积结合起来的联合方法（Repetti 等人，2017年；Dabbech 等人，2021年）、SASIR 方法（Girard 等人，2015年）、RESOLVE 贝叶斯方法（Junklewitz 等人，2016年），以及 Clean 多尺度反卷积方法（Rau 和 Cornwell，2011年；Hoekstra 等人，2005年）。最后，去噪（全面综述参见 Roscani 等人，2020年）、源检测和表征，则使用源搜索软件包中通常提供的工具完成，例如 PyBDSF（Mostert 等人，2021年）、SoFiA（Serra 等人，2015年）和 AEGEAN（Hancock 等人，2012年），这些都是较新的工具。此外，还有一些功能全面且广泛采用的软件平台，能够执行上述大部分任务，例如 CASA（McMullin 等人，2007年）、AIPS（Greisen，2003年）、MIRIAD（Sault 等人，1995年）和 ASKAPsoft（Wieringa 等人，2020年）。最后，值得一提的是 Terris 等人（2022年）和 Dabbech 等人（2022年）展示的深度学习在成像领域的最初两项应用。

成像是处理流程中计算需求最高的步骤之一，无论是内存需求还是计算时间都如此；相关操作包括网格化，即将观测数据卷积重采样到计算网格上，或进行快速傅里叶变换，以在傅里叶空间与实空间之间相互转换。当考虑具有大型视场的观测（当前射电干涉仪的观测通常如此，尤其是在最低频率下）时，计算需求会进一步增加，因为此时不能忽略曲率效应，问题也会因引入w 项校正而完全变为三维问题（见第 2 节，Cornwell 等人 (2008年）；Offringa 等人 (2014年）。

为应对这类三维问题，人们实现了多种算法，其中一些算法经过启用和优化，以利用特定的计算资源（例如图形处理器或多核平台）。目前，应用最广泛的方法是 w-投影算法（Offringa 等人，2014年），该算法将 w-项表示为傅里叶空间中的卷积，并直接将其应用于可见度。这里，w 表示可见度空间坐标系（u、v、w）中的第三个方向（详见第 2 节）。Lao 等人（2019年）对 w-投影算法进行了并行化，并将其移植到图形处理器上，但他们将数据分布限制在可见度数据上。Muscat（2014年）也提出了一种基于 CUDA 图形处理器的实现方案。另一种方法是 w-堆叠技术（Cornwell 等人，2008年）：该方法将数据划分到 w中，并在快速傅里叶变换后，通过对每个 w 平面进行乘法运算，在图像空间中应用 w 校正。WSClean（Offringa 等人，2014年）采用了这一方案，随后 Arras 等人（2021年）和 Ye 等人（2022年）对原始实现进行了优化。然而，该方案仅支持多核共享内存架构。WSClean 软件还实现了 IDG 方案，同时支持多核和图形处理器实现（van der Tol 等人，2018年；Veenboer 和 Romein，2020年）。Merry（2016a）也使网格化步骤能够在图形处理器上运行。Pratley 等人（2020年）在分布式内存系统上将 w-堆叠和 w-投影方法结合并行化。另一种方案是分面法，即假设在天空的小区域内，w 校正为常数或线性变化（Cornwell 和 Perley，1992年）。图像通过拼接许多不同的面片构建而成。对于这种方法，DDFacet 软件包支持并行多核处理（Tasse 等，2018）。

上述解决方案只能部分应对利用现代混合式高性能计算解决方案处理大数据所带来的挑战。具体而言，并行计算未能得到有效利用，无法处理规模日益增大的数据集和图像；这些数据集和图像无法放入单一系统的内存中，而结合加速架构则能大幅缩短求解时间。

在这项工作中，我们针对成像流程中体现 w 堆叠算法特征的步骤，着重应对这一挑战，具体包括网格化、快速傅里叶变换和 w 校正。下文将这些步骤的组合称为 w 堆叠网格器。这些步骤适合采用分布式内存并行，可利用并行快速傅里叶变换方案，并依托笛卡尔三维计算网格；该网格易于分布，也能在不同处理单元之间得到高效管理，从而有望在大型高性能计算架构上实现良好的可扩展性。

此外，我们将可移植性作为首要目标之一。鉴于目前可用的高性能计算解决方案种类繁多，而且应用程序代码的开发周期通常长于高性能计算架构演进的典型周期，我们的目标是设计一套代码，使其能够轻松移植到不同计算平台，并兼顾代码易用性和性能。这样，科学家就能有效利用所有可用的超级计算资源，在新系统上构建和运行代码时不会遇到重大障碍，并能在各种平台上获得可接受的性能和可扩展性。

代码采用 C 语言标准开发（仅通过 CUDA 扩展 C++以支持图形处理器），且未针对任何特定计算架构进行微调，以避免引入会限制其可移植性的专用解决方案。为了实现连图形处理器实现也可移植的方案，除了 CUDA，我们还试用了 OpenMP 支持，以将操作卸载到加速器上。我们没有采用 OpenCL 或 Kokkos 等性能可移植解决方案，因为它们会引入对额外库和应用程序的不必要依赖。

我们展示了在两个最先进的超级计算平台上获得的结果：于利希超级计算中心（德国，以下简称*Juwels*）的 Juwels Booster 模块，以及 CINECA 超级计算中心（意大利，以下简称*M100*）的 Marconi 100。根据[4] 2022年6月的 TOP500 榜单，它们分别是欧洲排名第三和第六的高性能计算系统。两个不同平台上的编译，以及使用不同编译器进行的编译，均采用同一版本的源代码。唯一需要的定制与编译标志和环境变量有关。所有测试均使用 LOFAR 数据集进行，这些数据集能够代表当前的射电观测。

本文结构如下：第 2 节介绍用于执行 w 堆叠的方法；第 3 节介绍代码采用的不同高性能计算解决方案；第 4 节展示并讨论性能和可扩展性测试结果；第 5 节得出结论。

[3] https://basp-group.github.io/Puri-Psi/

[4] https://www.top500.org

2 THE叫堆叠网格化器

干涉仪测量与天空亮度分布相关的复可见度 V，其表达式如下：

$$V(u,v,w) = \int\int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} \times e^{-2\pi i\left(ul+vm+w\left(\sqrt{1-l^2-m^2}-1\right)\right)} dl dm, \quad (1)$$

其中 u，v，w 是天线坐标系中的基线坐标，I 是谱亮度，而 l，m 是天空坐标。对于较小的视场，项 $\sqrt{1-l^2-m^2}$ 接近于 1，公式 1就是普通的二维傅里叶变换。为了加快计算速度，可采用基于快速傅里叶变换（FFT）的方法求解。然而，这需要将作为点状数据的可见度映射到规则网格上，从而离散化（u，v）空间。具体做法是将可见度数据与有限尺寸核进行卷积，将其转换为连续函数，随后即可对其进行快速傅里叶变换。

一次观测较大视场（FoV）时，来自非共面射电干涉望远镜的可见度数据无法通过二维傅里叶变换准确成像：成像算法需要考虑 w-项。该项描述了天线阵列偏离平面的程度。

一种可能的处理 w项的方法是 w层叠法（Cornwell 等人 2008年），该方法在计算网格中增加了 w 方向上的第三维，并将可见度映射到最近的 w平面。网格化完成后，分别对每个 w 平面进行傅里叶变换，并应用如下相位校正：

$$\frac{I(l,m)\,(w_{\max}-w_{\min})}{\sqrt{1-l^2-m^2}} = \int_{w_{\min}}^{w_{\max}} e^{2\pi i w\left(\sqrt{1-l^2-m^2}-1\right)} \times \iint V(u,v,w) e^{2\pi i(ul+vm)} du dv dw. \quad (2)$$

详情请参阅Offringa 等人 (2014年)。

该 w层叠方法采用 C 编程语言实现为一段代码，并使用了一些++ 扩展，这是 CUDA GPU 实现所需的；该实现采用过程式编程方法。图示形式的代码如图1所示，其中突出显示了采用分布式并行实现的部分，以及进一步经过加速的部分。

该算法以两种主要数据结构为特征。第一种是一个非结构化数据集，用于存储每个测量时间下天线阵列基线的（u，v，w）坐标。每条基线对应一定数量的可见度，其数量由频率带宽、频率分辨率和偏振数决定。此外，还为每个测量值和偏振分配一个权重。第二种是一个大小为 $N_u \times N_v \times N_w$的笛卡尔计算网格，其中 N_u，N_v 和 N_w 分别表示三个坐标方向上的网格单元数（见图4）。卷积后的可见度及其快速傅里叶变换结果均在网格上计算。这两种数据结构决定了算法的内存需求。它们按照第3.1节所述，均匀分配到不同的计算单元中。

代码由五个主要算法组件构成，每个组件支持不同类型的高性能计算实现。第一个组件负责从存储在磁盘上的二进制文件中读取观测数据。文件以并行方式读取，并将相同比例的数据分配给每个并行任务（MPI 任务，参见第 3.1 节）。

接下来的组件负责对可见度进行网格化。网格化沿 v 轴以连续的矩形分片进行。网格化过程分为三个子步骤。第一步为每个分片创建一个链表，将 u-v 坐标位于相应分片内的数据串联起来。第二步是网格化内核。遍历链表，选出属于指定分片的可见度，并通过高斯核进行卷积：

$$\tilde{V}(u_i,v_j,w_k) = \sum_{m\in\text{measures}} V_m G((u_m,v_m,w_m),(u_i,v_j,w_k)), \quad (3)$$

其中 m 是第 m 个测量值，$(u_m,\ v_m,\ w_m)$ 是其坐标，$(u_i,\ v_j,\ w_k)$ 是一个计算网格点，V_m 是测得的可见度，$\tilde{V}$ 是卷积到网格上的可见度。该 G 核函数是高斯卷积函数，核大小为 7 个网格单元，遵循 Offringa 等人 (2014年)。尽管高斯核并未用于任何生产成像器，且会导致图像空间中的混叠响应较高，但为了将不同计算架构支持方式各异的超越运算纳入计算，本研究选择采用这种方案。使用典型的生产用核函数（例如通常通过查表和插值实现的 Kaiser-Bessel 函数（Jackson 等，1991））无法测试此类运算。另一方面，像 Barnett 等人 (2019) 这样的解析近似，在计算开销上与高斯函数相当。无论如何，预计不同内核不会导致计算开销出现显著差异，Offringa 等人 (2014年) 也指出了这一点。最后一个子步骤是管理不同计算单元之间的数据交换。

算法的第三部分对网格化数据执行快速傅里叶变换（FFT），生成实空间图像。该过程使用 FFTW 库完成。

第四步是应用相位移，并将 w-平面归约为最终图像。

在第五步也是最后一步中，将生成的图像写入磁盘上的文件。

3 高性能计算实现

我们的 w 堆叠网格器的高性能计算实现包括：*i)*将数据分发到不同的计算单元，*ii)* 将工作负载分配到不同的计算单元，*iii)* 利用加速器加速计算。在全文中，我们将计算或处理单元 称为负责处理部分工作的计算实体。在基于 MPI 的并行计算中，计算单元是单个核心（对应一个*MPI* 任务）。在多线程 OpenMP 实现中，计算单元是多核 CPU。在加速计算中，计算单元是图形处理器。

3.1 数据分布

为了有效利用内存，所有大型数据集都必须进行分布处理；在我们的案例中，这些数据集涉及测量得到的可见度以及定义在网格上的数据。非结构化点状数据（观测可见度）与基于规则网格的数据（卷积可见度）同时存在，因此需要谨慎处理，才能实现高效的数据分布。

测量数据分布 – 点状可见度会在观测期间以时间序列的形式采集。在每个测量时间（快照），都会采集干涉仪中所有天线对（基线）产生的数据。各快照依次记录。由此形成的可见度分布具有空间数量

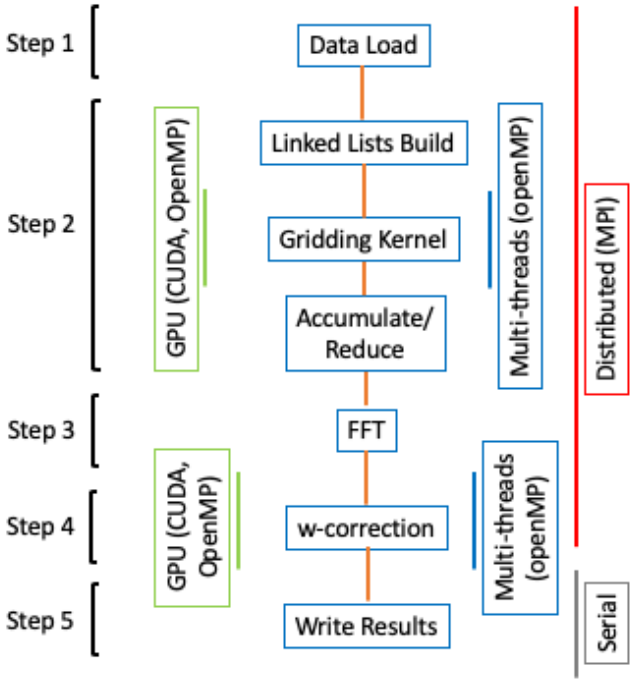


图 1. 示意性代码架构。图中突出显示了不同类型的高性能计算赋能技术。

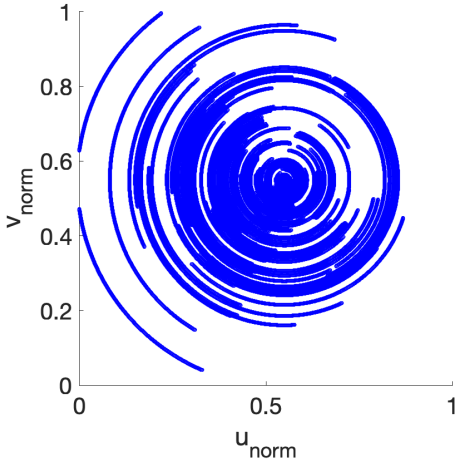


图 2. 射电干涉望远镜测得的测量值示例。图中展示了一次持续 8 小时的观测。角向覆盖范围和径向分布都可能极不均匀。 u 和 v 坐标均经过任意归一化处理。

密度，其变化幅度很大，取决于天线的排列方式和观测时间。例如，对于核心区域密集布局的天线阵列，$(u, v) = (0,0)$ 附近区域的密度高于外围区域。此外，径向数量密度并不均匀，会随朝向而变化。图2 展示了这类不均匀分布的示例。

在我们的方法中，可见度数据按时间片分布到各处理单元。如果 t_{obs} 是观测总时间，N_{pu} 是处理单元的数量，我们将数据划分为 N_{pu} 个时间片，每个时间片包含的时间间隔为 $t_{pu} = t_{obs}/N_{pu}$。处理单元 0 拥有时间 0 到 t_{pu} 的数据，处理单元 1 拥有 t_{pu} 到 $2t_{pu}$ 的数据，依此类推。图 3 展示了数据分布在 4 个处理单元之间的一个示例。

这种数据分布有几个优点。首先也是最重要的一点是，数据分布均衡，因为每个计算单元拥有数量相同的可见度，这些可见度分布在整个 $(u,\ v,\ w)$ 体积中。其次，数据在从磁盘读取时即可完成分布，无需后续通信。当然，要实现高效处理，还需要并行文件系统（我们假设高性能计算设施中具备该系统）。第三，位于笛卡尔网格同一区域、且可见度会在该区域进行卷积的数据，会分布到不同的计算单元，从而减轻该区域的计算负担，并降低潜在的竞态条件（即多个线程同时写入笛卡尔网格中同一个网格点）的发生概率。主要缺点是数据不具备局部性：数据分散在整个空间域，也就是计算网格中；此外，较短的观测还可能导致负载不均衡。这些问题可以通过按适当标准对数据排序等方式缓解。然而，这会增加额外的计算开销和通信开销，抵消可能带来的收益（例如参见 Hariri 等，2016）。因此，我们选择采用第 3.2 节所述的基于链表的方法。

网格数据分布– 定义在笛卡尔计算网格上的数据采用矩形板状分解（图4）分配给各个计算单元，为每个任务分配一个大小为 N_{mes} $h = N_u \times N_w \times (N_v/N_{pu})$ 个单元格的矩形区域，其中 N_u 和 N_w 分别是网格在 u 方向和 w方向上的大小。 N_v 是网格在 v 方向上的大小，该方向被划分为 N_{pu} 个部分，数量与计算任务数相等。

这种域分解方式使每个内存中仅需存储整个网格的一部分，从而避免创建副本。因此，每个不同的计算单元负责存储和管理网格中的特定部分（下文称为驻留切片）。此外，切片

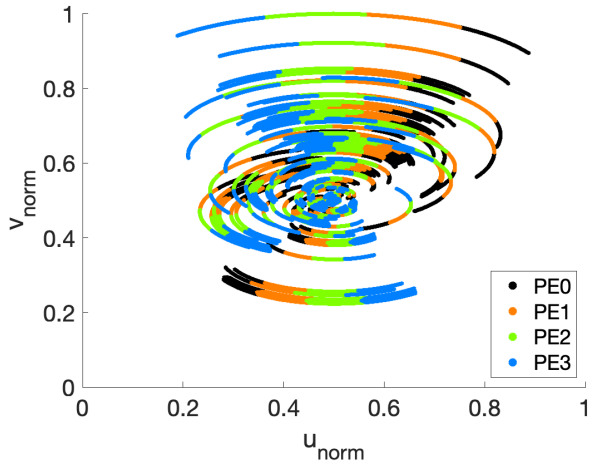


图 3. 观测数据在 4 个处理单元（标记为 PE#）之间的二维分布。后续时间片分配给各个处理器。每个处理器都拥有分布在整个 (u, v) 平面上的数据。

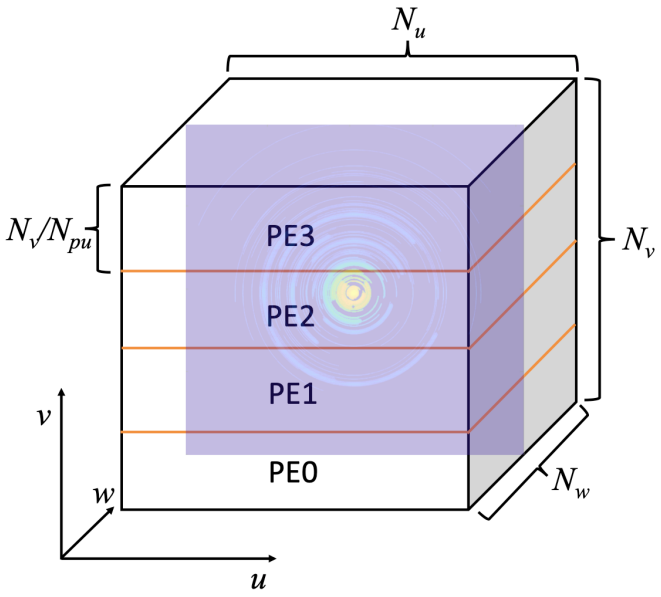


图 4. 笛卡尔计算网格的数据分布。每个计算单元（标记为 PE#）都拥有一个大小为 $N_u \times N_w \times (N_v/N_{pu})$ 个单元格的矩形驻留切片。本例中，数据分布在 $N_{pu} = 4$ 个计算单元之间。

分解为使用并行 FFTW 库提供了自然的布局，可将基于网格的可见度数据转换为图像。这种方法的主要缺点是：*i)* 可使用的计算单元的最大数量等于 v 方向上的单元格数量（网格沿该方向分解）：$N_{pu} \leq N_v$。$N_v > 1000$

不过，由于我们处理的网格相当大，因此这预计不会构成实质性的限制；*ii)* 必须实现适当的通信方案，以便重新组合图像。

3.2 并行处理

数据分布意味着需要额外管理多个计算单元所执行的工作，并通过通信在处理器之间传递数据。此外，还必须妥善实施同步，以确保在代码执行的特定步骤中，通信和工作都已正确完成（例如，在将结果写入磁盘之前）。通信和同步会限制算法并行执行的效率，即其可扩展性；可扩展性衡量的是随着计算机规模或数据量的增长，算法处理更多工作的能力。因此，为了充分利用日益强大的计算资源、支持更大的数据集并缩短求解时间，必须尽可能减少通信和同步。实现这一目标的方式是采用消息传递接口（MPI，Gropp 等，1994 年），这是分布式并行计算的事实标准。

数据输入/输出 – 从磁盘读取数据是通过在并行文件系统之上使用标准 POSIX 输入/输出实现的。每个计算单元同时打开输入二进制文件，并读取分配给它的数据部分（使用 fseek、fread 函数）。我们的工作并未针对输入/输出子系统的性能，因此没有进行专门的性能分析和优化。最终图像的写入操作也是如此。如有必要，可采用 MPI 输入/输出来提升这些操作的性能。

可见度网格合成 – 笛卡尔计算网格按照第 3.1 节所述划分为分片，并按以下流程进行处理。

每个处理单元同时计算其自身位于给定分片上的可见度的局部贡献。每个处理单元将此局部贡献存储在一个与驻留分片大小相同的辅助缓冲区中。给定分片上的可见度处理完毕后，存储在辅助缓冲区中的局部贡献便会累加到所属处理单元上的相应驻留分片中（驻留分片 i 属于 PEi）

并行求和采用两种不同策略实现：*i)* 使用阻塞式 *MPI_Reduce* 函数，将所有 MPI 任务的结果归约到拥有驻留切片的任务上，这意味着所有通信必须在下一次迭代开始前完成；*ii)* 使用非阻塞单边 *MPI_Accumulate*操作，在目标处理器的内存上执行并发归约操作。仅在遍历分片的循环结束时进行同步，此时调用集体 *MPI_Win_fence*函数。

在迭代过程中，对所有分片执行该操作，从分片 0 开始，直到分片 N_{pu-1}。处理完最后一个分片后，每个处理单元都拥有最新的驻留切片（即包含整个域中可见度贡献的驻留切片），可用于后续步骤（即快速傅里叶变换）。图 5 示意了四个计算单元同时对映射到分片 2 上的各自部分可见度执行网格化操作。完成网格化后，使用集体通信，将每个处理单元的贡献累积或归约到进程 2 的驻留切片上。

值得强调的是，位于同一分片（且属于同一处理单元）的可见度在内存中是分散存储的，因为测量值的内存位置与其物理空间位置之间没有关联；也就是说，内存中彼此接近的测量值在空间上可能相距很远（反之亦然）。这会导致内存使用效率极低。为加快内存访问，在遍历分片之前，每个处理单元都会创建 N_{pu} 链表。每个链表存放位于指定分片上的本地可见度。在每次迭代处理一个分片时，会遍历相应的

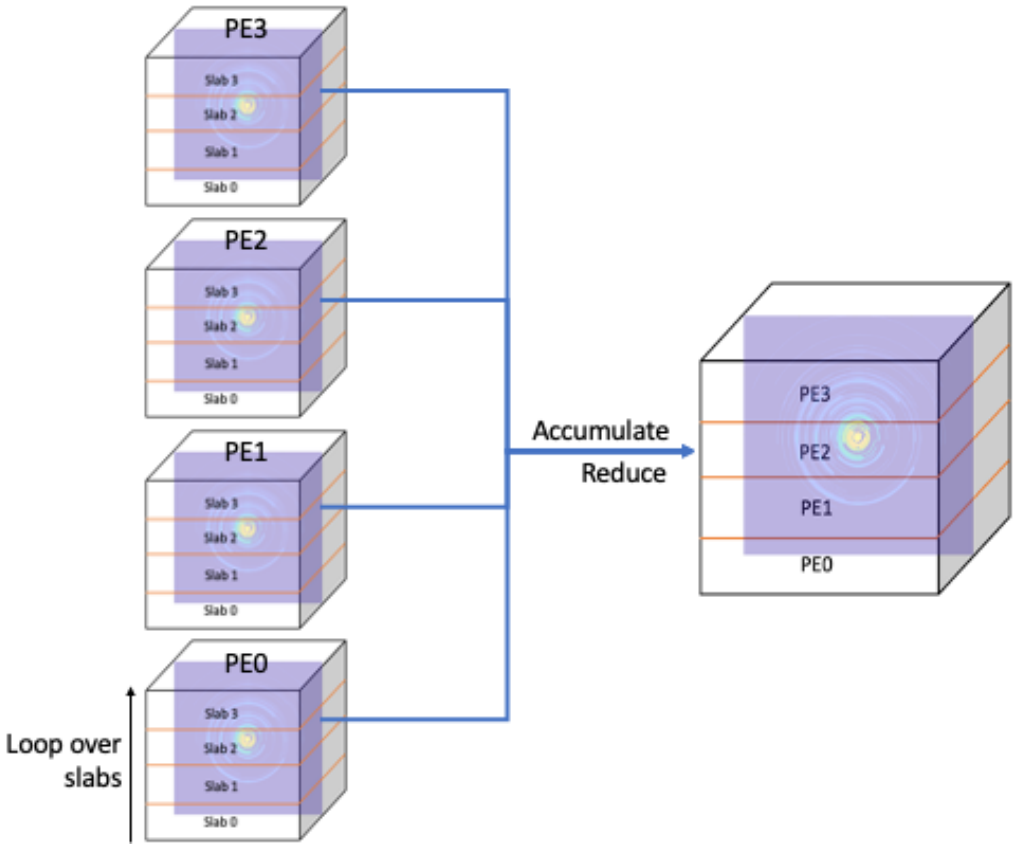


图 5。不同的计算单元（本例中为四个单元，以 PE# 表示）针对各自负责的可见度数据分片，逐片执行栅格化操作。该过程从分片 0 开始：所有处理单元利用链表加速内存中的可见度数据访问，计算分片 0 的局部贡献。最后，将各计算单元对分片 0 的贡献在驻留切片上累加/归约（即求和），该切片存储在 PE0 的内存中。随后，对分片 1 重复此过程（在 PE1 的驻留切片上累加），再处理分片 2（如本例所示，在 PE2 的驻留切片上累加），最后处理分片 3（在 PE3 的驻留切片上累加）。

链表，只提取所需的数据。提取的数据存入临时数组中；由于这些数组在内存中连续存储，因此网格化内核可以高效地处理这些数据。

快速傅里叶变换和相位校正– 计算得到的驻留切片通过 MPI 并行快速傅里叶变换进行处理，使用 FFTW 库。每个 w 平面单独计算。最后，应用 w 项相位偏移校正，将所有不同的 w 平面求和，生成最终的二维图像。相位校正是局部的，不需要 MPI 任务之间进行任何通信。

3.3 扩展定律

我们代码并行实现所遵循的扩展定律，可通过考虑关键模型参数来建模，即可见度数量 N_{vis}、计算单元数量 N_{pu} 和笛卡尔网格尺寸 $N_{\rm I} = N_u \times N_v \times N_w$。具体而言，我们关注代码中采用了分布式并行实现的部分，即网格化内核、快速傅里叶变换、w 校正和并行求和（称为“归约”）。每个函数都有其计算成本；在此处，计算成本定义为完成某项操作所需的时间 C_i，并且是某些模型参数的函数。

总体计算成本为：

$$\begin{aligned} C(N_{\rm vis}, N_{\rm I}, N_{pu}) = {} & \alpha_{\rm grid} \times C_{\rm grid}(N_{\rm vis}, N_{pu}) + \\ & \alpha_{\rm FFT} \times C_{\rm FFT}(N_{\rm I}, N_{pu}) + \\ & \alpha_w \times C_w(N_{\rm I}, N_{pu}) + \\ & \alpha_{\rm red} \times C_{\rm red}(N_{\rm I}, N_{pu}), \end{aligned} \tag{4}$$

其中 $\alpha_{\rm i}$ 是与模型参数无关的权重。

快速傅里叶变换的计算成本为 (Cooley 和 Tukey 1965)：

$$C_{\rm FFT}(N_{\rm I}, N_w, N_{pu}) \propto \frac{N_{\rm I}}{N_{pu}} \log\left(\frac{N_{\rm I}}{N_{pu} N_w}\right). \tag{5}$$

w 校正的计算成本为：

$$C_w(N_{\rm I}, N_{pu}) \propto \frac{N_{\rm I}}{N_{pu}}. \tag{6}$$

同样，网格化内核的计算成本为：

$$C_{\rm grid}(N_{\rm vis}, N_{pu}) \propto \frac{N_{\rm vis}}{N_{pu}}. \tag{7}$$

值得注意的是，参数 $\alpha_{\text{网格}}$ 取决于平滑核支撑域大小的平方（在我们的工作中为 7 个单元格），因此会对算法的计算成本产生显著影响。

最后，归约操作的开销为

$$C_{red}(N_{\rm I}, N_{pu}) \propto N_{\rm I} \times C_{\rm MPI}(N_{pu}). \tag{8}$$

与网格大小呈线性关系，是因为在我们的实现中，归约步骤所传输数据的总量不取决于处理单元的数量。因子 $C_{\mathrm{MPI}}(N_{pu})$取决于求和算法策略（归约还是累加）、带宽、延迟、网络负载以及求和操作的 MPI 实现。因此，其精确计算取决于本地配置和系统使用条件。无论如何，该因子都大于 1，随着 MPI 任务数量增加，并行求和的效率会逐渐降低。

总体而言，计算成本与每个处理单元上驻留的观测数据量以及切片大小均呈线性关系。对于给定的问题规模，相应的计算成本会随着处理单元数量的增加而降低。相反，归约项会随 N_{pu} 增长，因此其对计算成本的影响会逐渐增大；当启用大量 MPI 任务和 / 或网格及数据集较小时，它甚至可能成为主导因素。

3.4 多线程/众线程并行

通过利用多核解决方案加速代码，可以显著缩短每个 MPI 任务的求解时间。这些解决方案包括多核 CPU（多线程并行）或加速器，例如图形处理器（GPU，众线程并行）。加速器可作为通用协处理器与 CPU 配合使用，以加速可由独立任务高效执行的部分计算（内核）。

网格化和相位校正算法均针对多线程/众线程并行进行了优化。事实上，每个测量值的卷积和w校正都可以独立于其他测量值执行，且具有较高的算术强度，因此非常适合采用多线程并行执行。主要的潜在瓶颈在于将每个测量值的贡献累加到计算网格上，这会导致频繁出现竞态条件。

3.4.1 多线程 *CPU* 实现

多核 CPU 的多线程功能通过 OpenMP[5] 应用程序编程接口（API）实现。OpenMP 是一种成熟且稳定的解决方案，可充分利用共享内存设备，确保在多核心环境下具有良好的可扩展性和较高的可移植性。它受到所有主要硬件和软件提供商的支持，并已在 Fortran、C 和 C++ 等主流编程语言的常用编译器中实现。它基于一组指令，告知代码如何在可用线程之间分配工作，对源代码的影响极小；不过，为确保良好的性能和可扩展性，仍需进行特定的定制。

如果在编译时未明确启用 OpenMP，编译器会忽略 OpenMP 指令（将其视为注释）。（例如，对于 GCC 编译器，可通过 -fopenmp 编译标志启用 OpenMP。）

在我们的代码中，使用 OpenMP 对网格化和相位校正循环进行并行化：前者遍历测量值，后者遍历网格单元。两种情况下，循环迭代彼此独立，因此可以轻松地将其分配给 OpenMP 线程。

[5] https://www.openmp.org

累加不同线程的贡献时，必须注意避免竞态条件（即多个线程同时更新同一内存位置）。为此，我们使用 OpenMP 内置的 *atomic* 子句。

3.4.2 GPU 实现

图形处理器所采用的方法与多线程中描述的方法相似。

对于网格化算法，每个测量值分配给一个不同的线程，由该线程执行所有频率和极化的卷积。所需数据在内存中连续存储，因此能够有效地合并访问，从而提高内存使用效率。由于有数百万个测量值，可以实现较高的图形处理器占用率。

对于相位校正内核，嵌套循环中三个坐标方向的每次迭代都分配给一个不同的线程。

代码实现采用了两种不同的方法：第一种基于 CUDA[6]，第二种基于 OpenMP 的卸载构造。前者是 NVIDIA 开发的一种编程模型，用于在图形处理器上进行通用计算。目前，它是编写图形加速器程序时最高效、最全面的方法。不过，它仅限于 NVIDIA 图形处理器，因此限制了软件的可移植性。后者则是一组从 OpenMP 4.5 版本开始引入的指令，使开发者能够将数据和执行任务卸载到图形处理器等目标加速器上。基于指令的方法隐藏了 CUDA 等过程式方法固有的一些复杂性。编译器负责将带有指令的内核转换为面向加速器的构造，但这会牺牲一定的控制能力，进而限制所能达到的最高性能。此外，尽管原则上 OpenMP 可在不同平台和架构间移植，但目前支持它的编译器仍然有限。GCC 版本> 9目前支持完整标准，并可为 NVIDIA 和 AMD GPU 生成加速代码。NVIDIA 最近发布了 HPC 标准开发套件，其中包含 C 和 Fortran 编译器，并全面支持包括 OpenMP、OpenACC 和 CUDA 在内的一系列编程模型。

CUDA 实现 – 为保持单一源代码并确保其可移植性，代码中专用于 GPU 实现的部分会利用 __CUDACC__ 宏在编译时激活；该宏由 CUDA NVCC 编译器自动定义。

If s定义该宏后，以下两个函数将 GPU:

- 负责管理数据，将计算所需的数组（且仅限这些数组，以尽量减少数据传输）复制到图形处理器或从图形处理器复制回来；
- 将遍历可见度的循环和遍历三个坐标方向的循环替换为图形处理器内核调用，并将每次迭代分配给不同的线程；
- 使用 CUDA 的 atomicAdd 操作管理更新共享数组时的竞态条件。该操作获得硬件支持，可优化对内存位置的并发写入访问。

如果不使用 NVCC 编译，编译器会忽略 CUDA 部分，代码将在 CPU 上运行。

OpenMP 实现——基于 OpenMP 指令的编程模型允许用户插入非可执行的编译指示，

[6] https://developer.nvidia.com/cuda-zone

这些指示可指导编译器处理系统底层的复杂性。

目前，OpenMP 提供了一组指令，可将代码区域的执行卸载到加速器设备上。这些指令可以定义映射到目标设备的目标区域，也可以定义主机内存与目标设备内存之间的数据传输。我们正在实现的主要指令是 target data 和 target，它们分别用于创建数据环境，以及将代码区域的执行卸载到加速器设备上。这两种指令都配合 map 子句使用，以管理与之关联的数据；map 子句指定主机内存与目标设备内存之间的数据传输方向（to、from 或 tofrom）。

如果同一节点上有多个图形处理器可用，则使用一个标志将每个图形处理器映射到不同的 MPI 任务，从而优化可用资源的利用率。__ACCOMP__ 宏用于在编译时启用某些 OpenMP 特定函数（例如，检查图形处理器硬件和可用图形处理器的数量）。

竞态条件通过与 CPU 版本相同的*atomic* 指令处理。

所有 OpenMP 测试使用的编译器都是 NVIDIA 标准开发套件中的 NVCC 编译器，与 CUDA 测试使用的是同一套框架。

4 结果

为了分析我们代码的高性能计算实现的性能和可扩展性，我们在于利希和 CINECA 超级计算中心提供的两种不同的尖端高性能计算架构上进行了多项测试和基准测试。

于利希研究中心运营 Juwels Booster 模块，配备 936 2×AMD EPYC 7402 24C 2.8GHz 节点，每个节点配有 4 个 NVIDIA A100 图形处理器，每个图形处理器拥有 40 GB HBM2e 内存，并通过 4×InfiniBand HDR 互连网络连接。CINECA 是意大利国家级高性能计算设施，运行 Marconi100 系统，由 980 2×16- 核心 IBM Power9 AC922 CPU 节点组成，每个节点配有 4 个 NVIDIA Volta V100 图形处理器。每个 CPU 节点拥有 256 GB DDR4 内存，图形处理器内存为 16 GB HBM2。CPU 与图形处理器通过 PCIe Gen4 互连网络连接，而每个节点上的 4 个图形处理器则采用 NVLink 2.0 连接。系统互连网络采用 Mellanox IB EDR DragonFly++。两套系统均部署了 IBM Spectrum Scale 并行文件系统。

表 1 总结了各系统的特性，并列出了所采用编译器和 MPI 库的版本。

测试和基准测试使用了两个不同的输入数据集，分别是中型和大型数据集。这两个数据集来自一次持续 8 小时的 LOFAR HBA 内站观测，其主要特性见表 2。中型数据集是一个中心频率为 146 MHz、带宽为 2 MHz 的单个子频带（如此窄的带宽是这次特定观测的特点；典型的 HBA 观测带宽为 48 MHz），并划分为 20 个信道。该数据集用于对代码的单节点/GPU 实现进行基准测试，以及进行强可扩展性测试。大型数据集由完整观测数据构成，包括 25 个带宽各为 2 MHz 的子频带，频率范围为 120 至 170 MHz。该数据集用于弱可扩展性测试，并用于生成第 4.3 节所示的“大型图像”。两个数据集的主要特征，以及两种情况下采用的计算网格尺寸（后者逐渐增大），见表 3。

在所有系统中，均使用以下工具编译了相同的源代码：GCC 和 NVCC 编译器，以及可用的 MPI 库。代码唯一的依赖项是 FFTW3 库。编译代码时，只需对 Makefile 中的少数几个变量进行调整（例如 FFTW 或 CUDA 库的路径）。整个构建过程只需几秒钟。

4.1 可扩展性

我们首先分析代码的可扩展性，广义而言，即随着计算系统或应用规模的增大，处理更多工作负载的能力。我们测量了强可扩展性和弱可扩展性。

4.1.1 强可扩展性

第一组测试使用中型数据集考察代码的强可扩展性（见表 3）。强可扩展性衡量的是在数据和网格大小保持不变的情况下，逐步增加所采用的计算资源（即 MPI 任务数、OpenMP 线程数和图形处理器数量）时，代码的性能表现。理想情况下，代码执行时间应随计算单元数量呈线性缩放（即计算单元数量翻倍时，耗时应减半）。然而，多种因素会影响这一理想表现，导致计算单元数量增加时性能下降。因此，强可扩展性体现了用户在增加协同工作的计算单元数量时能够切实获得的性能提升。

完整代码及其主要算法组件的强可扩展性通过加速比 参数进行测量，计算方式如下：

$$S = \frac{T_{A,1}}{T_{A,n}} \tag{9}$$

其中$T_{A,\ n}$ 表示在系统 A 上使用 N个计算单元测得的墙钟时间，而$T_{A,\ 1}$ 表示在同一系统上使用基线配置（例如单个计算单元）测得的相应时间。OpenMP 测试仅在计算节点的单个处理器插槽上进行（由于非本地内存访问，较大的配置效率较低），而 MPI 和 GPU 测试最多在 4 个节点上进行。

CPU 上完整代码的强可扩展性 – 图 6 展示了完整代码的强可扩展性。对于所采用的两种计算系统，使用 CPU 核心（纯 *MPI* 运行，左图）时，最多 32 个核心都能保持近乎线性的可扩展性。超过这一数量后，加速比开始下降，这反映出通信对总计算时间的影响增大，原因是第 3.3 节讨论的若干问题。总而言之：*i)* 随着 MPI 任务数量增加，通信数据量保持不变，而每个任务的计算负载则不断降低（因为需要处理的子域越来越小）；*ii)* MPI 操作的数量随任务数量增加，通信流量开销也随之增大；此外：*iii)* 超过 32 个核心后，通信会经过互连网络，其速度无论如何都慢于本地内存访问；*iv)*随着任务数量增加，代码执行的不平衡程度也会加剧，这是因为各个处理单元上分配的数据量（位于给定分片上的数据量）可能各不相同。

为了减轻负载不均衡的影响，本文引入了基于 MPI 累积的通信方案。与基于 MPI_Reduce 的方法不同，该方法会在每次遍历分片的迭代时强制同步，因负载不均衡而累积开销；基于 MPI_Accumulate 的方法则是非阻塞的。它只需在以下时刻进行同步：

系统	计算节点	N_{nodes}	互连网络	编译器	MPI 发行版
M100	2×16-核心 IBM Power9 AC922 CPU + 4×NVIDIA Tesla V100 GPU	980	Mellanox IB EDR DragonFly++	GCC 8.4.0 NVCC 10.1	Spectrum MPI 10.4.0
JUWELS	2× 24 核心AMD EPYC 7402 24C CPU 940 + 4×NVIDIA Tesla A100 GPU		4×InfiniBand HDR（Connect-X6）	GCC 10.3.0 NVCC 11.3	ParaStationMPI 5.4.10

表 1. 本文测试所采用的两个高性能计算平台的技术特性、所采用的编译器及相应的编译选项。

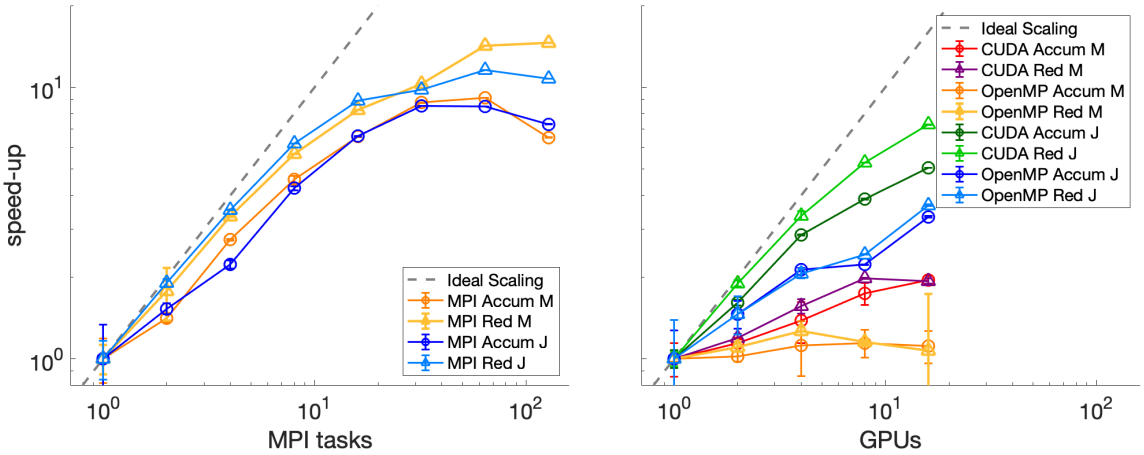


图 6. 完整代码在不同计算系统上的强可扩展性：M100 (M) 和 Juwels (J)。左图显示纯 MPI 运行，右图显示图形处理器运行。Accum 和 Red 分别表示 MPI_累加和 MPI_归约版本。CUDA 和 OpenMP 表示相应的图形处理器实现。强扩展性以加速比衡量（参见公式 9）。灰色虚线表示理想的线性扩展。

LOFAR HBA 内站	
RA	15:58:18.96
DEC	+27.29.19.2
观测时间	8 小时
积分时间	4 秒
天线数量	62
带宽	120 - 170 MHz
子频带数量	25
每个子频带的信道数	20
子频带带宽	2 MHz
最大紫外距离（波长）	58624.99
最大紫外距离（米）	119902.23
最小紫外距离（波长）	13.61
最小紫外距离（米）	27.83
视场	~12 度^2
角分辨率	~6 角秒

表 2. 用于中型和大型测试数据集的 LOFAR HBA 观测的主要特征。目标的坐标列于前两行。

分片循环结束时。由于每个计算单元上的可见度总数相同，预计该方法不会受到局限于各分片的负载不均衡影响。

然而，可扩展性曲线显示出相反的趋势：与 MPI_Reduce 相比，MPI_Accumulate 的可扩展性略差。

	中型	大型
可见度数量（约）	0.54×10^9	13.6×10^9
输入数据大小（GB）	4.4	102
网格大小最大值（$N_u \times N_v \times N_w$）	4096×4096×16	32768×32768×64
网格大小（GB）	4.0	1024.0

表 3. 中型和大型测试中输入数据集与计算网格的特征。网格大小考虑了实部和虚部所需的内存总量。

本节稍后将对此进行进一步讨论，并同时考虑弱可扩展性测试的结果。

完整代码在图形处理器上的强可扩展性——使用图形处理器时（图6右图），由于图形处理器的执行速度快于 CPU，通信的影响也相应增大，从而导致加速比下降（见第3.3节）。可扩展性因图形处理器实现方式而异。CUDA 版本的可扩展性优于 OpenMP 版本。事实上，OpenMP 版本的 GPU内核执行速度略快于 CUDA 版本，因此通信的影响更大，导致扩展性随之下降。FFTW 库的使用对图形处理器版本的可扩展性有不可忽视的影响。事实上，与 CPU 版本相比，图形处理器版本中用于 FFTW 库的计算时间占总计算时间的比例相对较高，因为网格化和 w 校正在图形处理器上执行得快得多

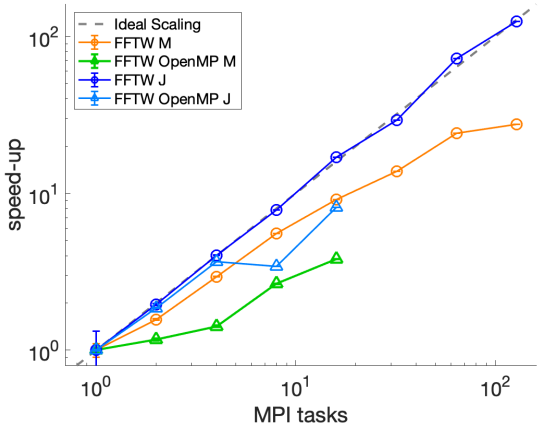


图7. 不同计算系统上 FFT算法的强可扩展性：M100 (M) 和 Juwels (J)。代码的 OpenMP CPU 版本标记为 OpenMP。其余曲线对应纯 MPI 并行。强扩展性以加速比衡量（见公式 9）。理想的线性扩展以灰色虚线表示。

加速器。在 Juwels 上，FFTW 库的纯 MPI 可扩展性呈线性，因此完整应用也能保持良好的扩展性（见图7）。而在 M100 上，加速比则低于线性。在标为*OpenMP*的两项测试中，FFTW 使用 NVCC 编译器重新编译，生成的 FFTW 库可扩展性较差。

网格化和 w-堆叠内核的强可扩展性——图8和 9 展示了代码中不涉及通信的部分的加速比，即网格化和相位校正内核。如第 3.3 节所述，这些内核的工作在各计算单元之间均匀分配。纯 MPI 多核测试（两幅图的左侧面板）显示，所有系统均实现了完美的线性扩展。中央面板展示了使用图形处理器时的加速比。由于采用了完美的数据并行算法，相位校正内核实现了近乎线性的扩展。对于网格化算法，随着待处理数据量减小，图形处理器的效率逐渐降低，因此 GPU 实现无法达到线性加速比。这种情况发生在将数据分配到多个加速器时。使用 8 个和 16 个图形处理器时，V100架构的性能仍持续提升（尽管低于线性增长），而数据量不足以充分发挥性能更强的 A100图形处理器的效能。

两幅图的右侧面板展示了 OpenMP 多核可扩展性。OpenMP 数据展示到单个处理器插槽上的核心数，并通过系统内存与进程绑定确保数据局部性。为利用内存局部性，循环调度策略采用静态方式。对于两个内核，最终的加速比都接近理想值，不过略逊于纯 MPI 的情况。

4.1.2 弱可扩展性。

我们使用大型数据集研究了弱可扩展性。在弱可扩展性测试中，处理单元数量和问题规模逐步增加，使每个处理单元的内存和工作负载保持不变。对于内存需求无法由单个节点满足的应用，弱扩展尤其有意义，例如处理大型观测数据和/或大型图像，以实现高分辨率和/或宽视场。

图 10 显示了总时间（左图）以及 FFTW 求解时间相对于单节点测得时间的归一化结果。计时采用的节点数分别为 1、2、4、8，对应的核心数（以及 MPI 任务数）为 32 至 256，图形处理器数（以及 MPI 任务数）为 4 至 32。

输入数据在一个节点上对应中型数据集，随后数据量与节点数量一同翻倍。分布在 8 个节点上的最大输入数据集约为 36 GB。同时，计算网格的大小从 4096×4096×16 配置开始翻倍。最大网格为 8192×8192×32，对应的总内存占用为 32 GB。

在理想的弱扩展情况下，求解时间应随节点数保持不变。图 10左侧面板显示，实际测得的总求解时间会随问题规模和节点数增加。这种增长主要源于 MPI 相关的通信开销，正如第 3.3 节所讨论的那样。右侧面板展示的 FFTW 曲线基本平坦，表明扩展性理想。网格化和相位校正内核也呈现相同情况（此处未展示，因为它们完全保持恒定）。因此，开销来自切片数据的汇集，并取决于所采用的具体通信方式。

使用 MPI_Reduce 集合通信函数时，与图形处理器情况相比，纯 MPI 情况显示出更大的开销，这是由于 MPI 任务数量更多（多八倍），这与公式 4 的预期一致。使用基于单边 MPI_Accumulate 的解决方案时，这种差异更加明显。如果使用的 MPI 任务较少，如图形处理器运行的情况，基于 accumulate 的解决方案凭借其异步特性可提供最佳扩展性。然而，当任务数超过 64（纯 MPI 情况下为 2 个节点）时，扩展性会显著恶化。尽管 accumulate 解决方案采用非阻塞方式，但仍需在遍历所有分片的循环结束时进行同步。这通过 MPI_Win_fence 调用实现，该调用由 MPI2 标准定义。这种同步在任务数超过 64 时会带来很大的开销，似乎限制了它在大型配置中的使用。这似乎并非由标准的某种特定实现造成，因为该问题同时影响 M100 上提供的 SpectrumMPI 库和 Juwels 上提供的 ParaStationMPI 库。

4.2 性能与内存使用

本节讨论代码在单个计算单元上的性能。表 4096×4096×16 中展示了处理中型数据集以生成4像素图像所需的墙钟时间。这是受内存限制、可在单个图形处理器上运行的最大配置。结合上文给出的可扩展性信息，这些数据反映了所讨论算法的计算性能，并有助于用户估算在充分利用各种异构高性能计算资源的数据处理流水线中，w 堆叠网格器所需的时间。

单核性能 – 单核测得的数据（表4第一行）展示了不采用并行处理时的求解时间。应用程序的总运行时间从 Juwels 上的约 87 秒到 M100 上的 130 秒不等。得益于 AMD 处理器架构所提供的有效向量化，Juwels 系统在网格化和相位校正内核上的性能优于 M100（快两倍以上）。而 M100 机器采用了效率极高的 FFTW 库（速度是 Juwels 安装版本的 3 倍）。

单个图形处理器的性能 – CUDA 网格化和相位校正内核的实现，相较于单核基准测试，展现出预期的加速比。CUDA 的所有计时

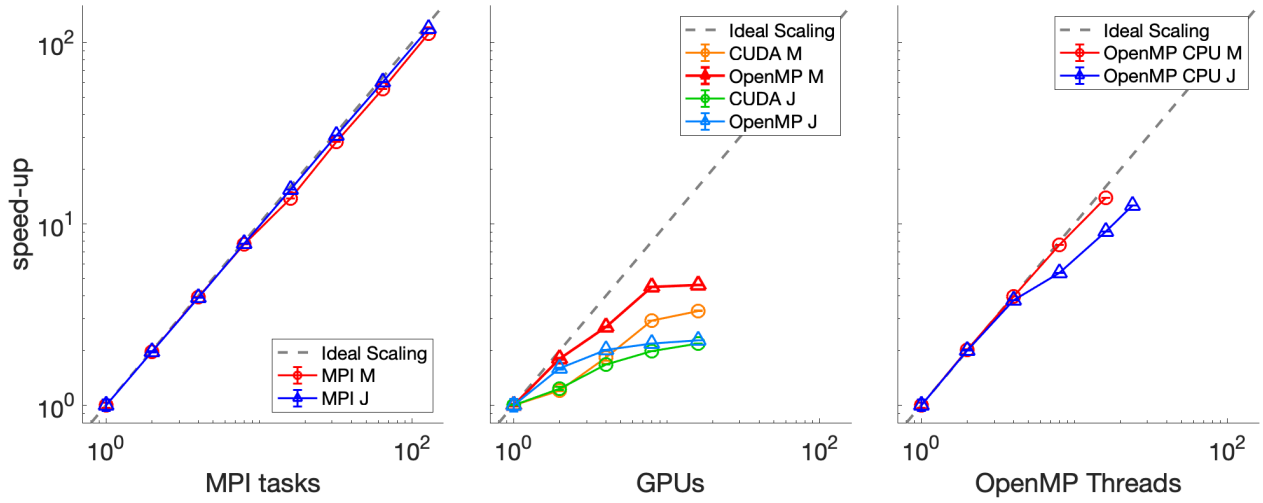


图 8. 不同计算系统上网格化算法的强可扩展性：M100 (M) 和 Juwels (J)。左图显示纯 MPI 运行结果，中图显示 GPU 运行结果（其中 CUDA 和 OpenMP 对应的 GPU 实现已标出），右图显示纯 CPU OpenMP 运行结果（限于单个处理器插槽，M100 为 16 个核心，Juwels 为 18 个核心）。强扩展性以加速比衡量（参见公式 9）。理想的线性扩展以灰色虚线表示。

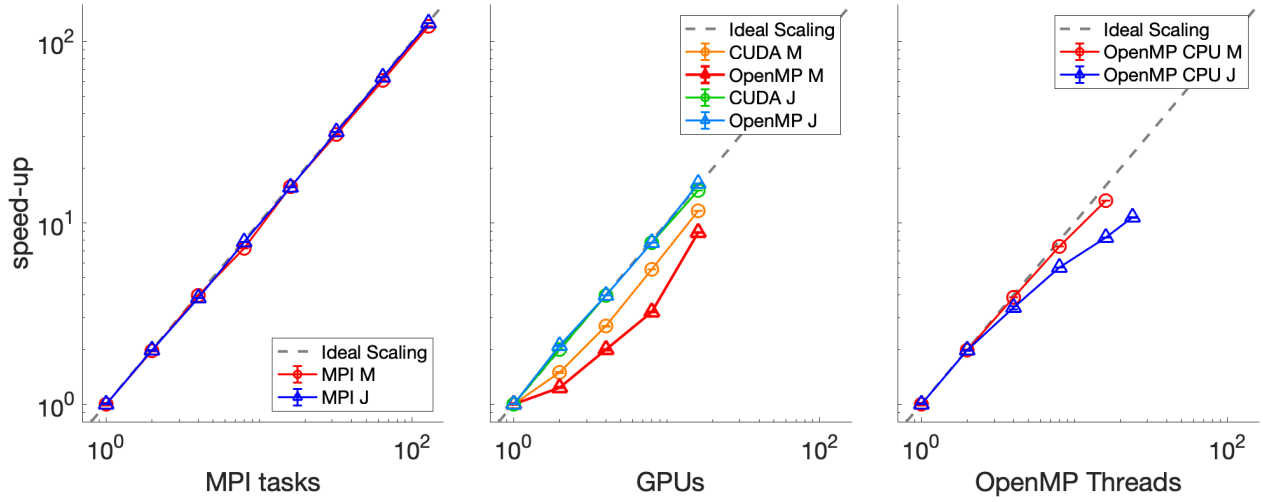


图 9. 不同计算系统上相位校正算法的强可扩展性：M100 (M) 和 Juwels (J)。左图显示纯 MPI 运行结果，中图显示 GPU 运行结果（其中 CUDA 和 OpenMP 对应的 GPU 实现已标出），右图显示纯 CPU OpenMP 运行结果（限于单个处理器插槽，M100 为 16 个核心，Juwels 为 18 个核心）。强扩展性以加速比衡量（参见公式 9）。理想的线性扩展以灰色虚线表示。

实现（见表格第二行）4，均包含与加速器之间的数据传输时间。网格化算法在 M100 和 Juwels 系统上的速度分别比单核快 25 倍和 30 倍。相位校正内核相较于单核的性能提升更为显著：在 V100 和 A100 架构上的加速比分别为 75 倍和 40 倍。

FFTW 库没有图形处理器实现。FFTW 的耗时与单核相同，在总计算时间中占比最高。在 Juwels 系统上，使用 GCC 编译器构建的 FFTW 耗时占总时间的 80% 以上。在 M100 上，这一比例为 55%。使用 NVCC 编译器和自定义选项重新编译 FFTW 库后，其速度在 M100 上提升了 1.3 倍，在 Juwels 上则提升了约 2 倍。

（但其可扩展性有所下降，详见第 4.1 节）。相应的耗时列于表 4 的第三行，其中还展示了采用 GPU OpenMP 实现的网格化和相位校正内核的耗时（同样包括与图形处理器之间的数据传输时间）。值得注意的是，对于我们的可移植实现，在这两套计算系统上，OpenMP 的运行速度都快于 CUDA。这表明，OpenMP 当前的卸载能力已能达到良好的性能水平。

总体而言，未加速的 FFTW 会使启用图形处理器的代码版本的性能提升幅度降至：与单核相比，M100 上为 8.5 倍，Juwels 上略低于 2.5 倍。这凸显了将

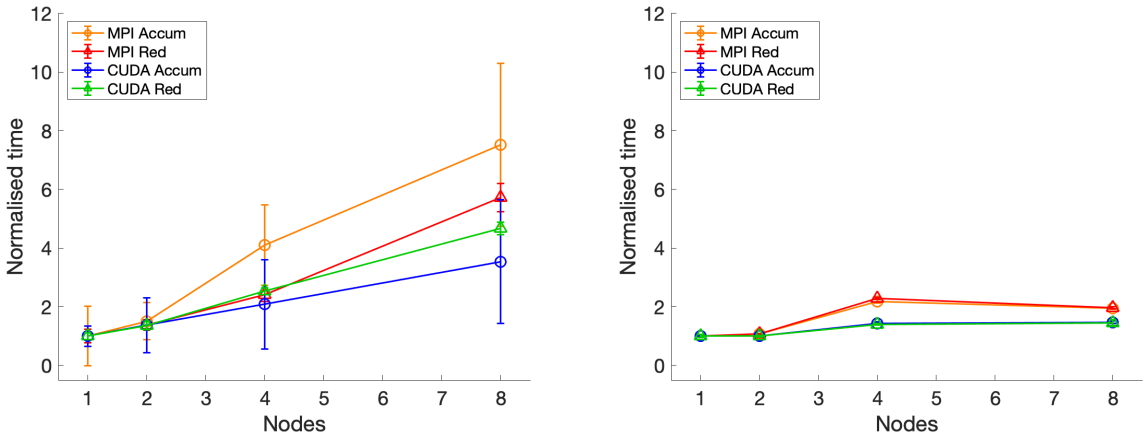


图10。 在 M100 系统上，完整代码（左图）和 FFT算法（右图）的弱可扩展性。Accum 和 Red 分别表示 MPI_Accumulate 和 MPI_Reduce 版本。MPI 和 CUDA 分别表示纯 MPI 运行和图形处理器运行。墙钟时间以单个节点上的求解时间为基准进行归一化。

完整代码移植到图形处理器上，以充分发挥加速系统性能的重要性。

单节点性能 – 节点级分析衡量了能够充分利用可用 CPU 的全部计算资源的软件所能达到的性能。我们展示了使用节点上的全部核心（纯 MPI，表 4 第四行）以及使用全部图形处理器（M100 和 Juwels 上每个节点均有 4 个图形处理器，分别列于表 4的第五行和第六行，对应 CUDA 和 OpenMP 实现）获得的计时结果。对于网格化内核，在 M100 上，CUDA 和 OpenMP 实现分别比纯 MPI 实现快约 1.5 倍和 3 倍；在 Juwels 上则分别快 1.5 倍和 2.2 倍。对于相位校正内核，在 M100 上的加速比分别约为 6.5 倍和 7 倍；在 Juwels 上，CUDA 和 OpenMP 的加速比均约为 5 倍。

总体而言，在 M100 上，代码的图形处理器实现比纯 MPI 实现快 1.3 倍（OpenMP）至 1.4 倍（CUDA）。在 Juwels 上，速度比分别约为 0.8 和 1。不过，节点上的图形处理器代码性能再次受到 FFTW 的显著影响，因为 FFTW 无法利用图形处理器。

我们通过 每秒十亿网格点加法次数 (GGPAS) 来衡量实现的有效性，这一指标沿用 Romein (2012年) 和 Merry (2016b) 发表的研究中的做法。值得注意的是，GGPAS 统计的是内核执行期间寄存器中的更新次数，而不是设备全局内存中网格点的更新次数（也就是说，GGPAS 不取决于网格的大小）。对于中型测试，估算出的每秒十亿网格点加法次数为 26.65。实测 GGPAS 的变化趋势自然与网格化内核的表现一致，其中性能最佳的是在 Juwels 系统上运行的 GPU OpenMP 实现，约为 49.60 GGPAS。我们得到的 GGPAS 低于上述研究中的结果，但那些研究采用了特定的优化。虽然我们的代码也可以进行这类优化，但这与我们在代码和性能两方面确保可移植性的目标不一致，因此我们没有进一步调整实现。

与 WSClean 的比较——图 11 展示了我们的代码与射电天文数据成像领域最先进的代码之一 WSClean 在求解时间方面的比较（Offringa 等人，2014年）。此比较仅用于检验我们实现的合理性，并非正式的基准测试，因为通过仔细调整参数，WSClean 的性能可能会得到提升。不过，这超出了本文的研究范围。运行时使用的是 WSClean 3.0 版，命令如下：

wsclean -j N -size 4096 4096 -scale 2asec -name test inputdata.ms，其中 -j 选项设置代码可使用的线程数 N，-size 选项设置以像素为单位的图像尺寸，-scale 选项设置以角秒为单位的分辨率，而 -name 选项设置输出文件的前缀。通过这种配置，WSClean 执行的操作与我们的代码相同。为了公平比较这两种代码，我们仅在一个节点上运行，且不使用图形处理器。WSClean 使用内置的多线程并行，我们的代码则利用基于 MPI 的并行。我们发现，在任务数不超过 4 时，两种代码的性能非常相近。对于更大的配置，多线程的扩展性略逊于 MPI，因此我们的代码性能略优于 WSClean。当并行任务数超过 16 时，两种代码的性能都不再提升。对于 MPI 实现，这是由于输入数据集规模较小，导致与并行相关的开销占据主导地位。而多线程则受到若干架构问题的影响（例如缓存一致性、内存争用等）。

内存需求 – w 分层代码每个计算单元的内存占用（字节）可计算如下：

$$M \simeq 8\times\frac{1}{N_{\rm pu}}\left(3.5\times N_b+(2\times N_{\rm vis}+4\times(N_u\times N_v\times N_w)+L_k\right)\text{B}, \quad (10)$$

其中， N_b 是基线数量， $N_{\text{可见度}}$是要映射到网格的可见度总数， $N_u\times N_v\times N_w$ 是网格单元数量， L_k 是用于将基线分配到不同分片的链表大小。系数 8 反映了大多数变量使用双精度浮点数；系数 3.5 则源于基线的三个相空间坐标以及单精度权重变量。系数 2 考虑了可见度的实部和虚部；系数 4 是可见度在网格上计算所得实部和虚部对应的系数 2，再乘以 2 得出的，后一个系数 2 是因为分片被复制，以便在内存中同时保留驻留切片和用于对可见度进行卷积的分片。 L_k 的大小可估算如下：

$$L_k \sim 2\times\alpha\times\frac{N_b}{N_{\rm pu}}, \quad (11)$$

其中，系数 2 反映了用于表征.....的变量数量

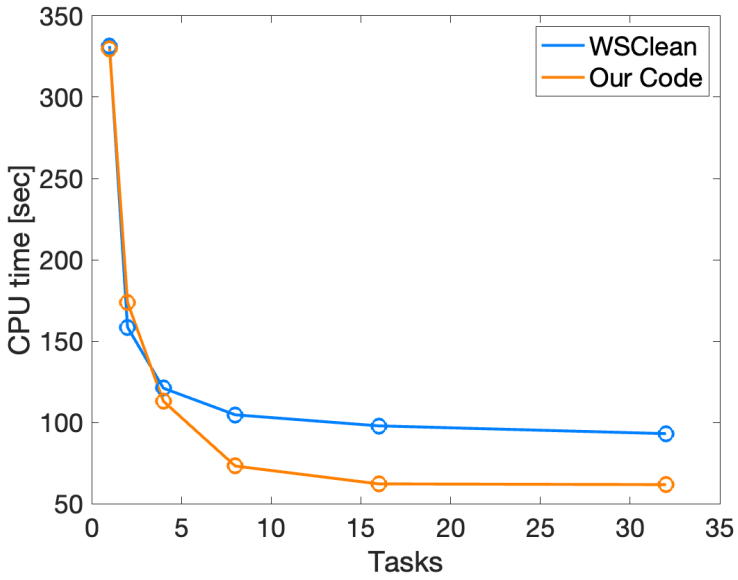


图 11. 在同一计算节点上，使用相同设置处理同一输入数据集时，随着并行任务数量增加，WSClean 与本文所述代码（“我们的代码”）的 CPU 时间对比。WSClean 使用多线程，而我们的代码采用 MPI 并行。由于 WSClean 3.0 版不支持图形处理器，因此未使用图形处理器。多节点并行的情况也相同。

链表中一个节点的变量数量（基线的 ID 和指向链表下一个节点的指针），α 是一个大于 1 的参数，用于考虑每个分片的边界条件（落在相邻分片中但对当前处理分片有贡献的基线）。该参数取决于卷积核的支撑范围（在我们的应用中为 7 个网格单元），但预计不会是 > 2。对于 $N_{\rm pu} > 2$，L_k 对 M 的贡献可视为可以忽略不计，其大小取决于处理单元数的平方的倒数。

方程 10 表明，计算中涉及的所有主要数据结构的大小都随计算单元数量线性变化；并行任务数量翻倍时，内存需求减半。与纯顺序代码相比，唯一显著的内存开销是额外的驻留切片和链表；不过，从计算单元数量达到几个起，这些开销便不再产生影响。

4.3 大型图像

我们的最后一项测试处理的是表 3中所列的大型配置，网格为 32768×32768×64，输出图像尺寸为 32768×32768 像素。为获得尽可能大的输入数据集，我们合并了可用观测的全部 25 个频段。

该测试需要约 2.5 TB 的内存，相当于使用 M100 系统的 128 个节点。纯 MPI 和 GPU 配置都已运行：前者使用 4096 个 MPI 任务（每个节点 32 个核心），后者使用 512 个（每个节点 4 个图形处理器）。两种情况下，求解时间均在 2750 至 3150 秒之间。在图 12中，32768×32768 像素图像与一幅 4096×4096 像素图像进行了比较。后者是确保以至少 2 个像素采样标称观测分辨率（~6 角秒）的最小图像尺寸。前者计算网格分辨率的提高，使其视场大得多；4096×4096 该视场对应于 32768×32768 图像中心的一小块方形区域。最外侧的源会因不同频段的人为合并而变得模糊；这些频段通常会分别处理，再进行适当合并，但为满足本次测试的计算需求，必须采用这种做法。

值得注意的是，处理一个 262144×262144×1 网格需要与大型测试相同的内存需求。这意味着使用上述配置可以生成一幅 262144×262144 像素图像。

5 结论

本文探讨了利用先进的高性能计算解决方案，支持大型射电干涉数据集的有效策略，以满足平方公里阵列射电望远镜的预期需求。我们重点研究了成像算法，尤其是网格化、快速傅里叶变换和 w 校正步骤，分析了其内存需求和计算时间。

我们开发了 w 堆叠网格器，以展示其在现代加速型多处理器设备上的高性能和可扩展性，同时兼顾可移植性和可维护性，从而便于在不同计算系统上使用同一软件，使其适用时间远超典型高性能计算技术的预期生命周期，最大限度地提升其可用性和影响力。

我们的主要成果可概括如下：

- 我们已使原型 w 堆叠网格器能够应用于异构高性能计算解决方案，将并行计算与加速器结合起来。
- 生成的代码支持处理大小仅受可用计算单元物理内存限制的数据集，并将数据均匀分布到分布式内存中。概念验证运行表明，该代码可在不到一小时内对 32768×32768×64 个网格单元组成的网格执行 102 GB 输入数据集的 W 堆叠（在内存中）（总内存需求约为 2.5 TB）。

	系统	总计（秒）	网格化（秒）	相位校正（秒）	FFTW（秒）	每秒十亿网格点加法次数（GGPAS）
CORE	M100	130.80±0.12	81.71±0.03	36.732±0.014	9.27±0.08	0.328± 0.002
	Juwels	87.56±0.16	36.47±0.04	15.839± 0.004	31.62±0.11	0.733±0.006
GPU CUDA	M100	15.92±0.03	3.25±0.01	0.493±0.005	9.27±0.02	8.18±0.02
	Juwels	36.79±0.07	1.239±0.004	0.397±0.001	31.55±0.06	20.84±0.02
GPU OMP	M100	12.11±0.01	2.64±0.02	0.309±0.004	6.92±0.03	10.09±0.04
	Juwels	18.56±0.39	1.08±0.08	0.400±0.002	15.04±0.01	26.15±0.41
NODE (核心)	M100	12.74±0.33	2.88±0.01	1.193±0.010	0.71±0.03	8.79±0.02
	Juwels	8.95±0.16	1.19±0.01	0.499±0.001	1.22±0.02	22.55±0.04
NODE (GPU CUDA)	M100	8.99±0.23	1.79±0.06	0.181±0.01	3.17±0.02	14.88±0.18
	Juwels	10.92±1.36	0.738±0.005	0.098±0.001	7.43±1.38	36.11±0.04
NODE （GPU 和 OpenMP）	M100	9.58±0.07	0.98±0.02	0.158±0.005	4.87±0.03	27.06±0.10
	Juwels	9.00±0.36	0.53±0.03	0.100±0.002	4.10±0.02	49.60±0.29

表 4. 在不同设备（核心、图形处理器、节点）和系统（Marconi 100 和 Juwels）上，使用不同代码内核处理中型数据集并生成 4096×4096×16 像素图像的耗时。误差通过十次测量值的标准差计算。最后一列显示了以每秒十亿网格点加法次数（GGPAS）表示的效率，即计算网格中各单元格的计算速率。

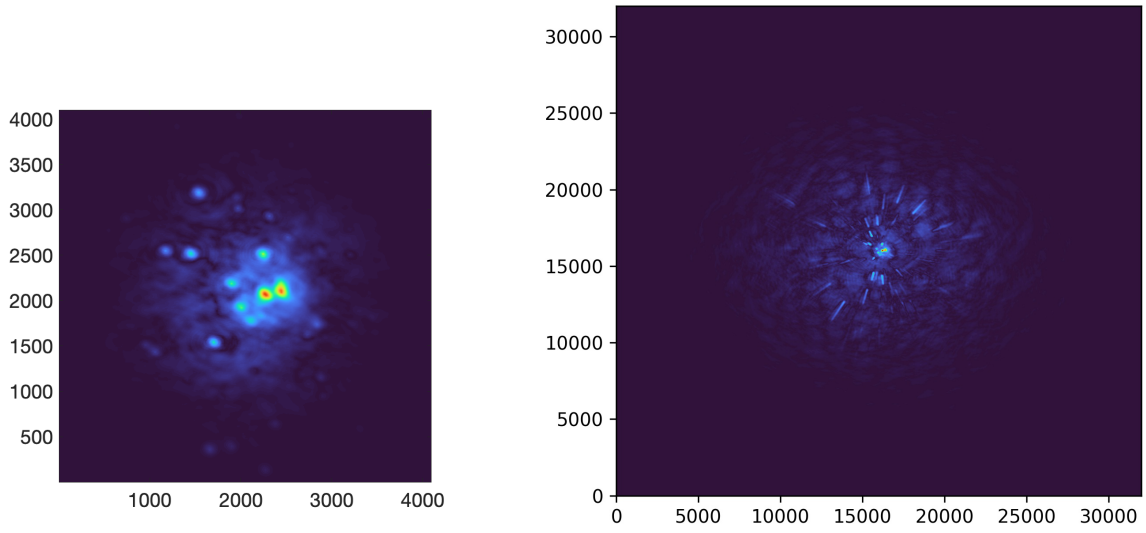


图 12. 对大型数据集应用 w 层叠过程后得到的结果，生成了 4096×4096 像素（左图）和 32768×32768 像素（右图）图像。不同的网格大小反映了两幅图像视场的差异。最外围源的模糊是由不同带宽的人为合并造成的。

• 代码的可扩展性（强扩展性和弱扩展性）确保其能够在大型高性能计算配置上高效运行，涉及数百甚至数千个协同工作的计算单元。

• 使用图形处理器可凭借其出色的性能缩短求解时间，因此，与纯 MPI 设置相比，可以使用更少的 MPI 任务，从而缓解与通信相关的问题。

• 当使用数百或数千个计算单元生成大型图像时，通信开销会对代码性能产生重大影响。不过，在内存中执行完整计算这一优势，仍能将求解时间控制在合理范围内（对于我们最大的案例，少于 1 小时）。

• 这段代码采用标准 C 编写，仅使用少量 C++ 扩展，因此很容易移植到不同系统上，并且无需针对特定架构进行调优。它可以使用 GCC GNU 编译器进行编译；如果使用 NVIDIA 图形处理器，也可以使用 NVCC 编译器进行编译。除 MPI 外，唯一的依赖项是 FFTW3 库，不过该库可以轻松安装在任何计算系统上。

• 借助 OpenMP 的卸载支持，只需使用简单指令即可将程序移植到图形处理器；原则上，这些指令可移植到不同的加速架构，而且在本例中，与 CUDA 相比没有任何性能损失。不过，卸载指令只有 GCC ＞ 9 版或 NVCC ＞ 10 版才提供完整支持。

• 使用高性能计算系统能够处理规模前所未有的数据和图像，这对于超大型数据集以及／或低频观测和超长基线天线阵列尤为重要。

这项工作仅仅是朝着充分利用射电天文学中极致高性能计算解决方案迈出的第一步。后续步骤包括

已经在优化图形处理器的使用方面展开（例如利用 CUDA 快速傅里叶变换库 cuFFT，其多节点版本已于 2022 年初发布），并且已在性能方面取得了令人鼓舞的结果；此外，还将代码扩展到其他类型加速器的利用上，尤其是 FPGA 架构。关键的一步还包括提升通信相关的性能：一方面探索新的模式以尽量减少数据交换，另一方面深入理解给定互连网络上的 MPI 数据传输与同步机制，从而充分发挥其效率。最后，我们将扩展研究范围，加入支持完整成像流程的其他功能。更具体地说，我们将引入射电天文学中常用的卷积核，例如长旋转椭球波函数或凯泽-贝塞尔窗函数（Jackson 等，1991），并将并行方法扩展到去网格化过程（从图像转换到可见度空间）。预计相同的并行化策略对这一逆过程同样有效，不过，由于需要将可见度分布到网格单元并还原为单个测量值，因此仍需进行专门实现，具体是在(u，v，w）空间中。此外，我们还将支持多波长观测以及不同偏振的完整处理。

数据可用性

本研究所使用的数据由 LOFAR 项目 LC14_018 生成，并将根据 LOFAR 的标准政策提供。

致谢

作者感谢 Francesco De Gasperin、Luca Bruno、Luca Tornatore 和 Gianfranco Brunetti，感谢他们提出宝贵的讨论、建议、意见和提示，帮助改进了本文内容。本研究所依据的高性能计算测试和基准测试在 CINECA（意大利博洛尼亚）的 Marconi100 超级计算机上，于 ISCRA 计划的 IscrC_TRACRE 项目框架内完成；同时也在于利希超级计算中心（德国）的 Juwels Booster 模块上完成，这得益于欧盟“地平线 2020”研究与创新计划 EuroEXA 项目（资助协议编号 754337）的支持。用于执行所有测试的数据由 LOFAR 项目 LC14_018 慷慨提供，项目负责人为 Franco Vazza。

参考文献

Arras P., ReineckeM., Westermann R., Enßlin T. A., 2021, 《天文学与天体物理学》, 646, A58Barnett A. H., Magland J. F., afKlinteberg L., 2019, SIAM J. Sci. Comput.,41, C479BhatnagarS., Cornwell T. J., 2017, 《天文学杂志》, 154, 197BhatnagarS., Cornwell T. J., Golap K., Uson J. M., 2008, 《天文学与天体物理学》, 487, 419Bhatnagar S., Rau U., Golap K., 2013, 《天体物理学杂志》, 770, 91 Carrillo R. E., McEwen J. D., Wiaux Y., 2012年, 《皇家天文学会月刊》, 426, 1223 Carrillo R., McEwen J., Wiaux Y., 2014年, 《皇家天文学会月刊》, 439Clark B., 1980, A&A, 89, 377Cooley J. W., TukeyJ. W., 1965, Mathematics of Computation, 19, pp. 297 Cornwell T., 1999, 收录于 Taylor G., Carilli C., Perley R. 编，《太平洋天文学会会议系列》第 180 卷《射电天文学综合成像 II》，第 151–170 页。Cornwell T. J., Evans K. F., 1985, 《天文学与天体物理学》, 143, 77 Cornwell T. J., Perley R. A., 1992, 《天文学与天体物理学》, 261, 353 Cornwell T. J., Golap K., Bhatnagar S., 2008, IEEE《信号处理精选主题期刊》, 2, 647 Dabbech A., Ferrari C., Mary D., Slezak E., Smirnov O., Kenyon J. S., 2015, 《天文学与天体物理学》, 576, A7 Dabbech A., Repetti A., Perley R. A., Smirnov O. M., Wiaux Y., 2021, 皇家天文学会月刊, 506, 4855 Dabbech A., Terris M., Jackson A., Ramatsoku M., Smirnov O. M., Wiaux Y., 2022, 《天体物理学杂志》, 939, L4 Girard J. N., Garsden H., Starck J. L., Corbel S., Woiselle A., Tasse C., McKean J. P., Bobin J., 2015, 《仪器学报》, 10, C08013 Greisen E. W., 2003, AIPS、甚大天线阵和甚长基线阵。施普林格·荷兰，多德雷赫特，第 109–125 页, doi:10.1007/0-306-48080-8_7, https://doi.org/10.1007/0-306-48080-8_7 Gropp W., Lusk E., Skjellum A., 1994, 《使用 MPI：基于消息传递接口的可移植并行编程》。麻省理工学院出版社，马萨诸塞州剑桥 Hamaker J. P., Bregman J. D., Sault R. J., 1996, 《天文学与天体物理学增刊》, 117, 137 Hancock P. J., Murphy T., Gaensler B. M., Hopkins A., Curran J. R., 2012, 皇家天文学会月刊, 422, 1812 Hariri F., et al., 2016, 《计算物理通讯》, 207, 69 Hoekstra H., Hsieh B. C., Yee H. K. C., Lin H., Gladders M. D., 2005, 《天体物理学杂志》, 635, 73 Högbom J. A., 1974, 《天文学与天体物理学增刊》, 15, 417 Jackson J., Meyer C., Nishimura D., Macovski A., 1991, 医学成像，《IEEE 医学成像汇刊》„ 10, 473 Jagannathan P., Bhatnagar S., Rau U., Taylor A. R., 2017, 《天文学杂志》, 154, 56 Johnston S., et al., 2007, 《澳大利亚天文学会刊》, 24, 174 Jonas J., MeerKAT 团队 2016, 收录于《MeerKAT 科学：通往 SKA 之路》，第 1 页 Junklewitz H., Bell M. R., Selig M., Enßlin T. A., 2016, 《天文学与天体物理学》, 586, A76 Kenyon J., Smirnov O., Grobler T., Perkins S., 2018, 《皇家天文学会月刊》, 478, 2399 Lao B., An T., Yu A., Zhang W., Wang J., Guo Q., Guo S., Wu X., 2019, arXiv 电子预印本，第 arXiv:1905.03213 页 McMullin J. P., Waters B., Schiebel D., Young W., Golap K., 2007, 收录于 Shaw R. A., Hill F., Bell D. J. 编，《太平洋天文学会会议系列》第 376 卷《天文数据分析软件与系统 XVI》，第 127 页 Merry B., 2016a, 《天文学与计算》, 16, 140 Merry B., 2016b, 《天文学与计算》, 16, 140 Mitchell D., et al., 2010, 收录于《射频干扰缓解研讨会》，第 16 页（arXiv:1008.2551） Mostert R. I. J., et al., 2021, 《天文学与天体物理学》, 645, A89 Muscat D., 2014, arXiv 电子预印本，第 arXiv:1403.4209 页 Offringa A. R., McKinley B., Hurley-Walker et al., 2014, 皇家天文学会月刊, 444, 606 Onose A., Carrillo R. E., Repetti A., McEwen J. D., Thiran J.-P., Pesquet J.- C., Wiaux Y., 2016, 《皇家天文学会月刊》, 462, 4314 Pratley L., McEwen J. D., d’Avezac M., Carrillo R. E., Onose A., Wiaux Y., 2018, 皇家天文学会月刊, 473, 1038 Pratley L., Johnston-Hollitt M., McEwen J. D., 2020, 《澳大利亚天文学会刊》, 37, e041 Rau U., Cornwell T. J., 2011, 《天文学与天体物理学》, 532, A71 Rau U., Bhatnagar S., Voronkov M. A., Cornwell T. J., 2009, IEEE《学报》, 97, 1472 Repetti A., Birdi J., Dabbech A., Wiaux Y., 2017, 皇家天文学会月刊, 470, 3981 Romein J. W., 2012, 收录于《第 26 届 ACM 超级计算国际会议论文集》。ICS ’12。计算机协会，纽约州纽约市，第 321–330 页, doi:10.1145/2304576.2304620, https://doi.org/10.1145/2304576.2304620 Roscani V., Tozza S., Castellano M., Merlin E., Ottaviani D., Falcone M., Fontana A., 2020, 《天文学与天体物理学》, 643, A43 Sault R. J., Teuben P. J., Wright M. C. H., 1995, 收录于 Shaw R. A., Payne H. E.,

Hayes J. J. E. 编，《太平洋天文学会会议系列》第 77 卷《天文数据分析软件与系统 IV》，第 433 页（arXiv:astro-ph/0612759）Schwab F., 1984,《天文学杂志》, 89, 1076 Serra P., et al., 2015, 皇家天文学会月刊, 448, 1922 Smirnov O. M., 2011a,《天文学与天体物理学》, 527, A106 Smirnov O. M., 2011b,《天文学与天体物理学》, 527, A107 Tasse C., van der Tol S., van Zwieten J., van Diepen G., Bhatnagar S., 2013,《天文学与天体物理学》, 553, A105 Tasse C., et al., 2018,《天文学与天体物理学》, 611, A87 Terris M., Dabbech A., Tang C., Wiaux Y., 2022, 皇家天文学会月刊，Thouvenin P.-A., Abdulaziz A., Dabbech A., Repetti A., Wiaux Y., 2020, arXiv 电子预印本，第 arXiv:2003.07358 页 Thouvenin P.-A., Dabbech A., Jiang M., Abdulaziz A., Thiran J.-P., Jackson A., Wiaux Y., 2022, arXiv 电子预印本，第 arXiv:2209.07604 页 Veenboer B., Romein J. W., 2020,《天文学与计算》, 32, 100386 Wieringa M., Raja W., Ord S., 2020, 收录于 Pizzo R., Deul E. R., Mol J. D., de Plaa J., Verkouter H. 编，《太平洋天文学会会议系列》第 527 卷《太平洋天文学会会议系列》，第 591 页 Ye H., Gull S. F., Tan S. M., Nikolic B., 2022, 皇家天文学会月刊, 510, 4110 van Diepen G., Dijkema T. J., Offringa A., 2018, DPPP：默认预处理流程（ascl:1804.003）van Haarlem M. P., et al., 2013,《天文学与天体物理学》, 556, A2 van der Tol S., Veenboer B., Offringa A. R., 2018,《天文学与天体物理学》, 616, A27

本文由作者使用 TEX/LATEX 文件排版。