为高性能数据工程增强分布式计算环境

Niranda Perera*、Kaiying Shan†、Supun Kamburugamuve‡、Thejaka Amila Kanewela‡、Chathura Widanage‡、Arup Kumar Sarker†、Mills Staylor †、Tianle Zhong†、Vibhatha Abeykoon‡、Geoffrey Fox†§
*Luddy 信息学、计算与工程学院，印第安纳大学，布卢明顿，IN 47408，
美国 dnperera@iu.edu

†弗吉尼亚大学，夏洛茨维尔，VA 22904，美国 {ks5qug, djy8hg, qad5gv, fad3ew}@virginia.edu
‡印第安纳大学校友，布卢明顿，IN 47405，美国 s
upun@apache.org, {thejaka.amila, chathurawidanage}@gmail.com§生
物复杂性研究所与倡议，弗吉尼亚大学，夏洛茨维尔，VA 22911，美国
vxj6mb@virginia.edu

arXiv:2301.07896v1 [cs.DC] 19 Jan 2023

摘要—**数据工程和数据科学社区已广泛接受在常规应用中使用 Python 和 R 数据框这一理念。在大数据革命和人工智能的推动下，如今这些应用对于处理太字节级数据至关重要。它们很容易超出单台机器的能力范围，同时也需要投入大量开发者时间和精力。因此，设计可扩展数据框解决方案至关重要。人们已经多次尝试解决这一问题，其中最引人注目的是使用 Dask 和 Ray 等分布式计算环境开发的 DataFrame 系统。尽管 Dask/Ray 的分布式计算功能看起来前景广阔，但我们认为 Dask 数据框/Ray Datasets 仍有优化空间。本文介绍 *CylonFlow*，这是一种替代性的分布式数据框执行方法，可在相同的 Dask/Ray 基础设施上实现最先进的性能和可扩展性（从而大幅提升其性能！）。为此，我们将高性能数据框系统 *Cylon* 集成到 Dask 和 Ray 中。Cylon 最初采用了完全不同的执行范式。实验结果表明，在由 DataFrame 运算符构成的流水线上，*CylonFlow* 的分布式性能比 Dask 数据框高出 30× 。有趣的是，由于 *Cylon* 采用原生 C++ 执行，它的顺序执行性能也更出色。我们相信 *Cylon* 和 *CylonFlow* 的成功不仅限于数据工程领域，还可用于整合高性能计算和分布式计算生态系统。**索引词—**数据工程，数据科学，high performance 计算、分布式计算、数据框**

一、引言

过去几十年间，数据工程领域以惊人的速度不断发展，这主要归功于大数据革命、机器学习（ML）和人工智能（AI）的兴起。在当今的信息时代，数据不再以兆字节、文件或电子表格来衡量，而是以千兆字节/太字节和对象存储来衡量。如此海量的数据占用了开发者大量时间进行数据预处理，而他们本可以将精力集中在构建数据工程模型上。因此，提高这些数据预处理阶段的性能，对于构建高效的数据工程流水线至关重要。

传统上，数据预处理是在数据库系统上使用结构化查询语言（SQL）完成的，但近年来，Python 和 R 编程语言已接管了这些 SQL 工作负载。这些语言的函数式接口、交互式编程环境和解释执行，为当代工程师提供了更加便于使用的开发生态系统。

Python 库 Pandas 一直处于这场变革的前沿，并在推动 Python 用于数据探索方面发挥了重要作用。本文主要关注 数据帧（*DF*）API，它是 Pandas 生态系统的核心。DF 的概念并非 Pandas 独有；事实上，它起源于 20 世纪 90 年代的 S 语言，随后由 R 语言推广开来。然而，根据 *PyPI* 软件包索引的统计数据 [1]，Pandas 始终占据主导地位，每月下载量超过 1 亿次。尽管如此受欢迎，Pandas 和 R 的 DF 即使面对中等规模的数据集，也会遇到性能限制[2]–[4]。例如，在一台配备 240GB 内存的 Intel® Xeon® Platinum 8160 高端工作站上，使用 Pandas 连接两个各有 10 亿行的 DF 约需 700 秒，而遍历每个数据框只需约 4 秒。另一方面，如今的计算机硬件拥有充足的算力和大量内存。按需弹性云计算服务只需轻触按钮，即可让工作在数千个此类节点上运行。因此，我们拥有丰富的资源，可用于开发更高效的分布式数据工程解决方案。

Hadoop YARN、Dask 和 Ray 只是众多分布式执行运行时中的几种，它们能够管理其管辖范围内数以千计的计算资源。这些引擎主要由分布式计算和云计算领域开发，并提供应用程序接口（API），方便用户将逻辑提交到多个节点上执行。它们采用多种执行模型，例如异步多任务（AMT）、参与者等。在数据工程领域，我们看到多个框架试图利用这些

分布式运行时来开发分布式数据框（DDF）解决方案。Spark SQL RDDs & Datasets 是这一领域的一项突破性框架，显著改进了传统的 MapReduce 范式 [5]。Dask 开发了自己的 DDF 方案 Dask DDF，紧随其后的是 Ray 的 Ray 数据集。Modin 是开发可扩展数据框系统[4], 的最新尝试，它同样构建于 Dask 和 Ray 之上。然而在实际应用中，我们发现这些系统存在若干性能限制 [2], [3], ，详见第 V 节。

传统上，高性能计算（HPC）社区一直基于整体同步并行（BSP）执行模型，并采用消息传递接口（MPI）规范开发解决方案。他们已在数千个 CPU 核心上（以及超级计算机上）实现了令人瞩目的可扩展性和性能。在此前的一篇出版物中，我们开发了一种现有 DDF 的替代方案，名为*Cylon* [2], ，从高性能计算视角审视这一问题。*Cylon* 采用 BSP 模型执行 DDF 算子，并运行在 MPI 运行时（OpenMPI、MPICH、IBM Spectrum MPI 等）之上。由于具备出色的可扩展性并源自高性能计算，我们将 *Cylon* 这一实现归类为高性能 *DDF*（*HP-DDF*）。除了基于 BSP 运行之外，HP-DDF 的另一项显著特征是使用优化通信库。

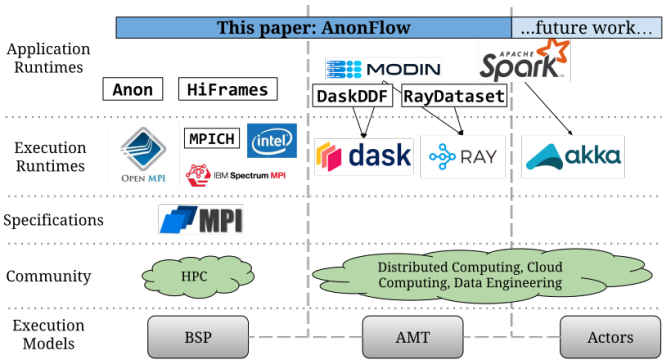

Fig. 1. 现状与*CylonFlow*的贡献

尽管 *Cylon* 的可扩展性已优于主流 DDF 系统，但它与 MPI 生态系统紧密耦合。正如我们在第四节中所讨论的，这限制了我们将 HP-DDF 概念扩展到 Dask 或 Ray 等分布式计算环境。本文提出了一种替代执行方法，以解决这一限制。我们的目标是在不牺牲其可扩展性和性能的前提下，将 *Cylon* 与其他执行运行时集成。这是一个由两部分组成的解决方案：1. 在执行运行时资源中创建有状态伪 BSP 环境；2. 使用模块化的 通信器，以便接入优化通信库。我们将其命名为 *CylonFlow* ，因为这一构想与工作流管理有相似之处。我们通过在 Dask（*Cylon* HP-DDF 运行时，称为 *Cylon-Flow*-on-Dask）和 Ray（*CylonFlow*-on-Ray）之上实现 HP-DDF 运行时，展示了这一构想的稳健性；这些运行时的表现优于各自原生的 DDF 实现。我们还确认，这一构想在相同硬件上的表现可媲美或优于基于 MPI 的 *Cylon*DDF。借助 *CylonFlow*，如今我们已能在从个人笔记本电脑到百亿亿次级超级计算机的各种平台上运行 HP-DDF。如图 1 所示，它将不同的执行模型和社群整合到单一应用运行时之下。据我们所知，这是首次尝试将高性能数据工程构件适配到分布式计算环境。我们认为，*CylonFlow* 背后的方法不止适用于数据工程领域，还可用于在分布式计算环境中执行许多高性能计算应用。

II. 分布式计算模型与库

要理解 *Cylon* 和 *CylonFlow* 的设计与实现，讨论现有的分布式计算模型及其常用实现库十分重要。分布式计算模型从机器的角度抽象地描述如何分解并执行特定问题，也说明分布式应用程序如何表达和管理并行度。数据并行使用多个计算单元，对数据的不同部分（分区）执行相同的计算。这种方式既出现在指令级别的单指令多数据（*SIMD*）中，也出现在程序级别的单程序多数据（*SPMD*）中。另一方面，任务并行是在多个计算单元上并行执行多个任务。这是程序级别多程序多数据（*MPMD*）的一种形式。

A. 整体同步并行（*BSP*）

整体同步并行或通信顺序处理器（CSP）模型[6], [7] 是最常见的模型，它在多个计算节点上采用 SPMD 和数据并行。消息传递接口（MPI）是 BSP 模型的正式规范，经过 30+年的发展已日趋成熟。OpenMPI、MPICH、MSMPI、IBM Spectrum MPI 等都是这一规范的知名实现。MPI 应用程序呈现静态并行特征，因为通常需要在程序启动时声明并行度。从数据的角度来看，这意味着数据分区与并行度紧密耦合。在应用程序启动时，数据分区会分配给执行器/工作节点。执行器随后会一直持有这些数据分区，直到应用程序结束，并对其执行计算。当工作节点在程序中遇到通信操作时，它们会通过传递消息彼此同步。许多高性能计算（HPC）应用程序在超级计算集群上使用 BSP 模型，并展现出出色的可扩展性。然而，只有少数数据工程框架采用了这一模型，其中包括*Twister 2* [8] 和*Cylon*。

B. 异步多任务（*AMT*）

AMT 模型通过将应用程序拆分为相互独立、可移植的子程序（许多任务）及其关联输入（数据依赖），放宽了整体同步并行的限制。AMT 运行时通常会管理一个用于接收这些

任务的分布式队列（管理器/调度器）。另设一组执行器/工作节点来执行队列中的任务，从而实现多程序多数据和任务并行。任务之间的依赖关系通过调度顺序来处理。这样一来，应用程序便可动态设置并行度，工作节点也可以扩容或缩容，从而实现动态并行。AMT 还支持在多租户/多应用环境中更有效地利用资源：空闲工作节点可以领取彼此独立的任务，从而提高系统的整体吞吐量。此外，任务并行支持任务级容错，失败的任务可以轻松重新运行。这些优势或许促使许多分布式数据框运行时（包括 Dask DDF 和 Ray Datasets）选择 AMT 作为首选执行模型。

C. 参与者

参与者 模型由 *Erlang* [9] 推广开来。参与者是一种基本计算单元，可以接收来自其他参与者的消息，并据此执行计算、创建更多参与者、发送更多消息，以及决定如何响应下一条收到的消息。与 AMT 中的执行器和任务相比，参与者管理并维护自身状态，而状态可能随计算或通信而改变。消息以异步方式发送，并存放在邮箱 中，直到指定的参与者取走。Akka 是一种流行的 actor 框架，曾作为 Apache Spark 项目的基础。有趣的是，Dask 和 Ray 项目也在各自的分布式执行运行时之上提供了参与者抽象，主要用于减少代价高昂的状态初始化。

III. 分布式数据框（DDF）

随着数据集规模呈指数级增长，可以公允地说，数据工程应用已超出单个工作站节点的能力范围。现代硬件提供了大量用于计算的 CPU 核心/线程，而最新的云基础设施使用户能够即时启动许多此类节点。因此，用户可支配的计算能力十分充足，数据工程软件必须加以利用。此外，每种 AI/ML 应用都需要预处理数据集，而数据预处理耗费大量开发者时间和精力，这已是众所周知的事实。多项 AI/ML 调查表明，这甚至可能占开发者总时间的 60% 以上 [10]。因此，采用可扩展的分布式数据框（*DDF*） 运行时，有望大幅提升数据工程流水线的效率。根据我们对一些广泛使用的 DDF 系统（第 V 节）所做的实验，我们认为，打造高性能可扩展 *DDF* 运行时 仍在持续探索之中。

A. 数据框（*DF*）

首先，让我们定义数据框。我们借鉴了 Abiteboul 等人提出的关系术语中的定义 [11]。与 SQL 表类似，数据框包含数据类型各异的数据。这些元素来自一组已知的域，$Dom = \{dom_1, dom_2, ...\}$。对于数据帧，这些域表示它所支持的所有数据类型。数据帧的**模式** S_M 是一个元组 (D_M，C_M)，其中 D_M是一个由 M个域组成的向量， C_M是一个由 M 个对应列标签组成的向量。列标签通常属于 字符串/对象 域。**数据帧（DF）**是一个元组 (S_M， A_{NM}， R_N)，其中 S_M是包含M 个域的模式， A_{NM} 是一个用于存储实际数据项的二维数组， R_N 是一个由 N 个行标签组成的向量，这些行标签属于某个域。数据帧的长度 为 N，即行数。

异构类型模式明显区别于多维数组或张量。不过，同一列中的数据仍是同质的，因此许多框架采用了列式数据格式，以支持对列进行向量化计算。由 NumPy 多维数组构成的集合是最简单的数据框表示形式。另一种常见做法是使用 Apache Arrow 列式格式 [12]，许多数据框运行时都采用了这种格式。Arrow 数组由多个缓冲区组成，例如数据、有效性信息，以及可变长度类型（如字符串）的偏移量。

正如先前文献所指出的，许多常用的数据框算子都是沿纵轴（按行）定义的 [3]，[4]。尽管列式表示支持对列进行连续访问，但对行进行索引或切片并非易事。此外，许多数据框算子都是基于一组键列定义的，而其余列（即值列）则随键列移动。因此，传统的基本线性代数子程序（BLAS）例程无法直接用于数据框算子。

B. 分布式数据框系统设计

数据帧的组合在设计分布式数据框系统时带来了若干工程挑战。与任何分布式/并行系统设计类似，我们先从整体上考察计算和通信方面。

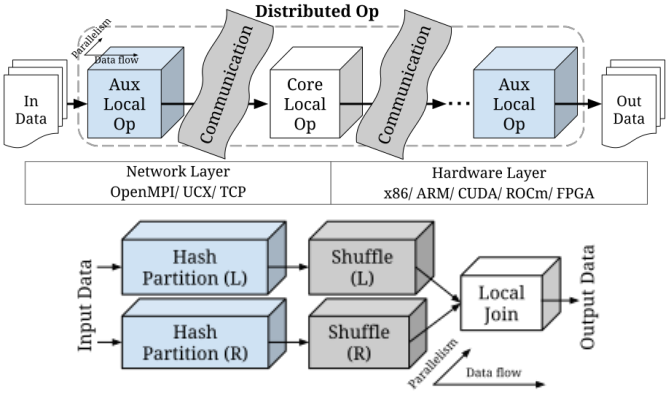

图 2. 分布式数据框子算子组合 [3]（下方：连接算子示例）

1）计算：Petersohn 等人 [4]认识到，许多 Pandas 算子都可以通过一组核心算子来实现，从而减轻实现庞大分布式数据框 API 的负担。相应地，在最近的一篇论文中，我们观察到数据框算子遵循若干通用分布

执行模式 [3]。模式决定这些子算子如何排列成有向无环图（有向无环图）。我们还发现，一个分布式数据帧算子由三个主要子算子组成：1. 核心本地算子；2. 辅助本地算子；3. 通信算子。图 2 展示了分布式连接操作的组合，图 3 展示了 *Cylon* 与 Modin 这些概念之间的关系。框架可以选择为每个子算子创建任务（即对一个工作单元的定义）。典型的应用会由多个分布式数据帧算子组成一条流水线。

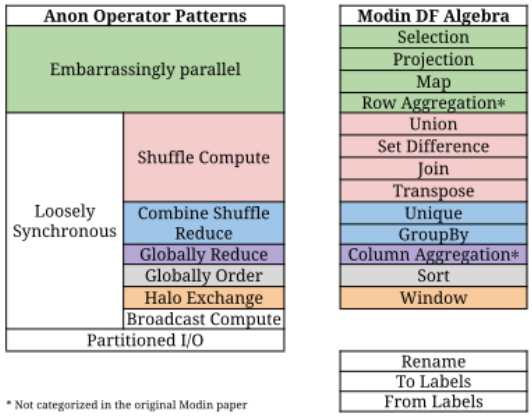

图 3. *Cylon*算子模式与 Modin 数据帧代数

使用 AMT 模型时，这些任务会针对每个数据分区（并行度）进一步展开。每个任务都会为后续任务生成输入数据。这种数据流决定了任务之间的依赖关系。当有向无环图中包含多个算子时，通常会将多个本地任务组合在一起。执行计划优化器 可以识别此类任务，并将它们合并为一个本地任务。我们可以在 Apache Spark SQL *Tungsten* 优化器中看到这类优化 [13]。如第 I 节所述，整体同步并行模型原生支持数据并行。由于执行器在应用程序结束前始终拥有数据分区，因此能够执行所有本地计算任务，直到到达通信边界。与 AMT 相比，该模型本身就支持合并后续的本地任务。

2）通信：实现分布式数据帧算子需要点对点（P2P）通信，以及工作进程之间复杂的消息传递。我们找到了几种此类集合通信例程，例如洗牌（全互换）、散播、（全）收集、广播、（全）归约等，它们对于分布式数据帧算子至关重要 [3]。通常，通信例程作用于数据缓冲区（例如 MPI、TCP），但数据帧的组合方式决定了这些例程还必须扩展到数据帧、数组和标量等数据结构上。这类数据结构可能由多个缓冲区组成（第 I 节），这会进一步增加实现的复杂性。例如，连接操作需要对数据帧进行洗牌，为此我们必须对所有列的缓冲区大小执行 AllToAll（计数）。随后，我们根据这些计数对列数据进行洗牌。在大多数数据帧应用程序中，通信操作可能占用大量实际运行时间，从而形成关键瓶颈。第 V-A 节展示了这一点：我们评估了多种数据帧算子模式下通信时间与计算时间的分布。此外，Spark SQL、Dask DDF、Ray Datasets 等的开发者文档都提供了专门的指南，以减少重分布例程的开销 [14], [15]。

虽然可以通过点对点消息传递巧妙地实现这些通信例程，但专用算法的实现已显著提升了性能 [16]–[18]。例如，OpenMPI 为其集体通信实现了多种此类算法，并可根据应用程序进行选择。通常，在 AMT 运行时中，任务之间的通信是在调度器的协助下发起的。另一种方法是使用分布式对象存储或网络文件系统来共享数据，而不是显式地发送/接收数据，不过这可能会导致严重的通信开销。

C. DDF 系统考察

让我们考察几种最常用的 DDF 系统，以了解它们的分布式执行模型和总体设计选择。然后，我们将把这些系统与第四节中描述的新方法进行比较。

1) Dask DDF：Dask DDF 是一种分布式数据帧，由许多沿纵轴分区的 Pandas 数据帧组成。Dask DDF 上的算子会被分解为任务，然后排列成有向无环图（图 4 展示了一个连接操作）。随后，Dask 分布式调度器会在 Dask 工作节点上执行这些任务。这种 DDF 执行是 AMT 模型的一个很好的示例。核心本地算子会交由 Pandas 执行。通信算子（主要是洗牌）通过由 *Partd* 磁盘支持的分布式对象存储，支持点对点 TCP 消息传递。

2) Ray Datasets：Ray Datasets 是一种类 DDF API，由存储在分布式对象存储中的 Apache Arrow 表或 Python 对象组成。与 Dask 类似，分布式算子（转换）遵循 AMT 模型。有趣的是，它们同时支持任务策略和执行器策略。对于开销较大的状态初始化（例如基于 GPU 的任务），建议采用后者，以便缓存初始化结果。在通信方面，系统采用映射归约式洗牌：先按值将任务映射到分区块，再通过归约任务合并共同分区的块。本质上，Ray 通信算子由对象存储提供支持。对于较大的数据，文档建议使用推送式洗牌。

3) Apache Spark 数据集：可以说，Apache Spark 是当今最受欢迎的基于 actor 的数据工程框架，自最初发布弹性分布式数据集 [5]以来，便吸引了庞大的开发者社区。*PySpark* 数据集是一种类分布式数据帧 API，近期还发布了名为*Spark* 上的 *Pandas*的类 Pandas 分布式数据帧。与 AMT 类似，Spark 会将算子拆分为一组 MapReduce 任务，随后由管理器进程将这些任务调度到分配的执行器中

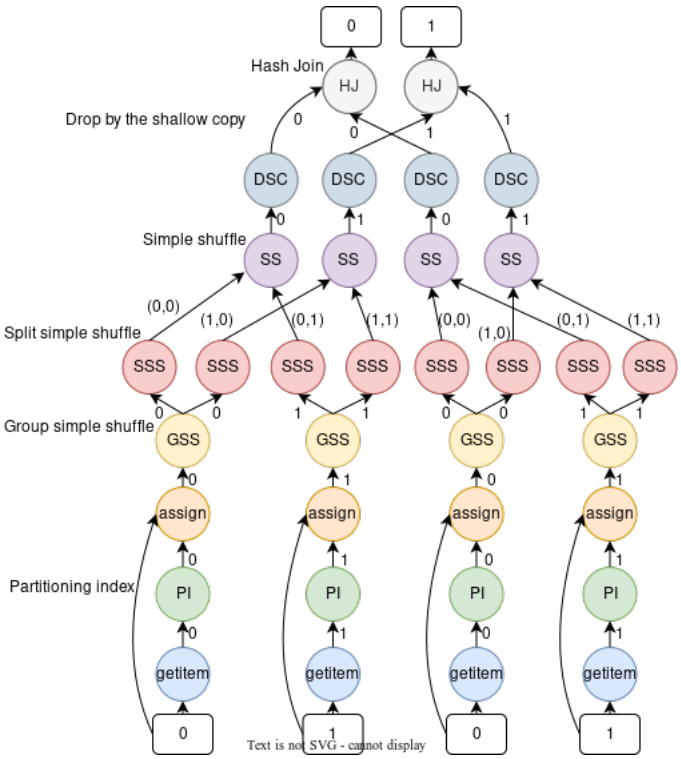

图 4。Dask DDF 连接（2 个分区）

应用程序。它使用*Akka-Actors* 来管理驱动进程（即提交应用程序的进程）、管理器和执行器。本质上，Spark 使用 Actor 模型为 MapReduce 任务实现了 AMT。所有这些进程都运行在 Java 虚拟机 (JVM) 上，在与 Python 传输数据时，可能会产生显著的序列化和反序列化开销。作为一种优化，最新版 PySpark 支持 Apache Arrow 列式数据格式。

4) Modin DDF： Modin [4] 是 DDF 领域的最新成员。它引入了*DF* 代数 的概念（图 3），其中一个 DDF 算子可以作为多个核心算子的组合来实现。它在 Dask 和 Ray 后端上执行，这些后端也为 DDF 提供通信手段。Modin 致力于仿照 Pandas API，并遵循急切执行模式，从而形成了自身特色。

IV. *Cylon* 和 *CylonFlow*：DASK 和 RAY 中的高性能 DDF

通过研究，我们发现，在使用上述 DDF 系统处理大型数据集时，存在若干性能限制。如第 V 节所述，其中许多 DDF 的可扩展性有限，我们认为 AMT 模型的局限性可能是造成这一问题的主要因素。集中式调度器可能会造成调度瓶颈。此外，缺乏专用且经过优化的通信机制进一步加剧了这些问题。可以合理地认为，通信例程的优化与设计 Dask/Ray 等分布式计算库是相互独立的，而将通用的分布式数据共享机制重新用于复杂的通信例程，可能会导致在 DDF 实现中使用时性能欠佳。

在最近的一篇出版物中，我们提出了一种适用于 DDF 的替代方法，该方法使用 BSP 执行模型，并将其命名为*Cylon* [2]。它构建于 MPI 之上，并使用 MPI 集合通信例程实现 DDF 算子。过去几十年来，MPI 库（OpenMPI、MPICH、IBM-Spectrum）不断成熟，采用了各种经过优化的分布式通信算法，而*Cylon* 从这些改进中获益匪浅。通过采用 BSP 模型，它还受益于数据并行，以及对本地任务的隐式合并。实验表明，*Cylon*具有出色的可扩展性，因此恰当地将其区分为高性能 *DDF*（*HP-DDF*）。尽管高性能 DDF 颇具发展前景，但必须依赖 MPI 环境也带来了若干限制。MPI 进程引导与底层 MPI 实现紧密耦合，例如 OpenMPI 使用 PMIx。因此，无法将 MPI 作为独立的通信库，置于 Dask/Ray 等分布式计算库之上使用。通常，这些库会自行引导其工作进程。MPI 运行时没有直接的方式绑定到这些工作进程。

我们坚信，将 HP-DDF 概念扩展到类 MPI 环境之外是值得的。当前的技术进步以及对高效数据工程解决方案的强烈需求，都支持这一主张。本文的主要动机是开发一种执行环境，使我们能够在整体同步并行的可扩展性与 AMT 的灵活性之间取得平衡。Dask 和 Ray 作为分布式计算库，已经证明了自身的实力。因此，我们没有从头开始构建新系统，而是专注于弥合整体同步并行与这些库之间的差距。我们针对这一问题提出了双管齐下的解决方案。首先，利用执行运行时的计算资源，创建一种有状态的伪 BSP 执行环境，为 HP-DDF 的执行奠定基础。第二步是采用模块化的 通信器 抽象（即定义通信例程的接口），以便接入优化通信库。我们将这个项目命名为*CylonFlow*，因为它秉持管理工作流这一理念。

A. 有状态伪 *BSP* 执行环境

在这个伪 BSP 环境中，执行器会初始化一个优化通信库，并将其附加到执行器的状态中。在*CylonFlow* 应用程序运行期间，该状态会使通信上下文保持活动状态。这样，*CylonFlow* 运行时便可重复使用通信上下文，无需重新初始化；对于并行度较高的情况，重新初始化可能会带来高昂的开销。环境设置完成后，执行器会隐式合并，并执行本地操作，直到遇到通信边界。正如第四节 IV-C 节所述，该状态还可用于在*CylonFlow* 应用程序之间共享数据。

创建有状态对象的这一方案与 Actor 模型完美契合。因此，我们利用 Dask 和 Ray 中提供的 actor API，实现了*CylonFlow*-on-Dask 和*CylonFlow*-on-Ray（图 5）。actor 是对驻留在远程工作节点上的指定对象（`Cylon`Actor 类）的引用。驱动进程/用户代码会调用这个远程对象的方法；在执行调用期间，*CylonFlow* 运行时会将通信上下文作为参数传入这些方法。这样，用户便可在这些方法中使用*Cylon* DDF 编写数据工程应用。

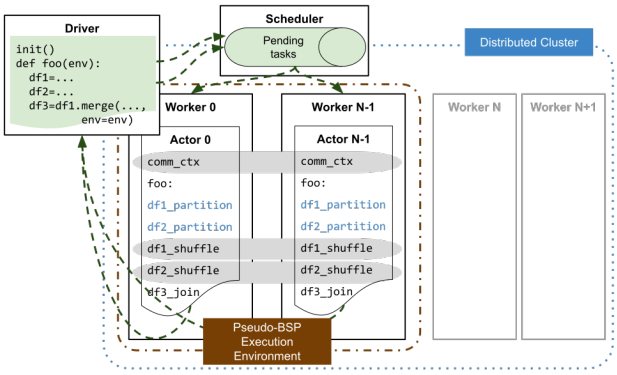

图 5. *CylonFlow*：使用 Actor 构建 HP-DFF

这种方法可以对集群资源进行分区，并调度相互独立的应用程序。这种工作划分的粒度会粗得多，但我们认为，现代处理器硬件中丰富的计算单元和存储资源，以及通过云计算获取这些资源的能力，仍足以支撑这种方式。据我们所知，这是首次将 actor 与专用通信库结合起来开发 HP-DDF 运行时。这种方法在语义上不同于 Apache Spark 中的 actor；后者仍以 AMT 方式执行相互独立的任务。此外，也不应将其与其他正交项目混淆，例如 *Dask-MPI*，后者可用于在现有 MPI 环境中轻松部署 Dask 集群。

应用程序初始化时，*CylonFlow* 会根据所需的并行度，将*Cylon* Actor 定义（一个类）发送给集群中的一部分工作节点。随后，工作节点将其初始化为 actor 实例（远程对象）。与此同时，这些 actor 实例会彼此建立通信通道，这是创建 *Cylon*DDF（即 `Cylon`_env）的入口。实例化 `Cylon`_env 可能是一项开销很大的操作，尤其是在并行度较高时，因为它需要在远程对象之间建立点对点通信通道。

Cylon *actor*类提供三个主要端点。

1) start_executable：允许用户提交一个可执行类，该类将在 actor 实例内部实例化。2) execute_`Cylon`：执行可执行类的函数，该函数接受一个`Cylon`_env 对象并生成一个 Future。

3) run_`Cylon`：执行一个 lambda 函数，该函数接受一个`Cylon`_env 对象并生成一个 Future。

以下示例代码使用 Parquet 文件创建两个 *Cylon*数据框，并对它们执行连接（合并）。

```
定义 foo(env:Cylon Env=None): 数据帧1 = 读取_parquet(...,env=env) 数据帧2 = 读取_parquet(...,env=env) 写入_parquet(df1.merge(df2, ..., env=env), ..., env=env)

init() wait(Cylon执行器(并行度=4).run_Cylon(foo))
```

1) 启动 *Dask Actors*：Dask 没有单独的 API 端点可为应用预留一组工作节点。因此，*CylonFlow* 使用 Distributed.Client API 收集所有可用工作节点的列表。随后，它使用 Client.map API 端点，并根据并行度选定工作节点列表，在这些节点上启动 actor 远程对象。Dask actor 远程对象会建立与驱动进程的直接通信通道，并通过该通道将结果传回。这样可以避免经过调度器产生额外的网络跳数，从而降低延迟。

2）生成 *Ray Actor*：Ray 提供了一种 *PlacementGroups*API，可在多个节点间预留成组资源（称为 gang 调度）。*CylonFlow* 会创建一个具有所需并行度的 placement group，并将*Cylon* Actor 定义提交给它。在 Ray 文档中，像这样相互通信的 actor 称为带外通信。

B. 模块化通信器

设置好伪 BSP 环境后，*Cylon*HP-DDF 通信例程便可在执行器之间传递消息。不过，由于前文讨论的限制，我们仍无法复用 MPI 通信。为此，我们必须寻找其他通信库，以便在不影响其可扩展性和性能的前提下，在 MPI 之外实现*Cylon* 通信例程。我们通过模块化*Cylon* 通信器接口，并按照第三节中的讨论，为 DDF 通信例程添加抽象实现来实现这一目标。这使我们能够方便地将 Gloo 和 UCX/UCC 库集成为 MPI 的替代方案。第五节 B 部分中的通信器性能实验表明，在相同硬件上，这些库的性能不逊于 MPI，甚至更好。

1）*Gloo [19]*：*Gloo* 是由 Meta 公司孵化器项目管理的集合通信库，主要面向机器学习应用。PyTorch 使用它执行分布式全归约操作。目前支持 TCP、UV 和 ibverbs 传输。Gloo 通信运行时可以通过 MPI 通信器或 NFS/Redis 键值存储进行初始化（点对点消息传递不受影响）。在 MPI 环境中，*Cylon* 使用前者；而在*CylonFlow* 中则使用后者。作为一个孵化器项目，Gloo 缺乏全面的算法实现，但我们的实验确认了它具有出色的扩展能力。我们扩展了 Gloo 项目，以适配 *Cylon* 通信接口。

2）统一通信 *X*（*UCX*） *[20]*：UCX 是一组库和接口，可在高速网络上高效、便捷地构建广泛使用的高性能计算协议，包括 MPI 标签匹配、远程内存访问（RMA）操作等。与 MPI 运行时不同，UCX 通信工作节点不受进程引导机制的约束。因此，许多框架都在使用它，包括 Apache Spark 和 RAPIDS（Dask-CuDF）。它提供基本的点对点通信操作。统一集合通信（UCC）是在 UCX 之上构建的集合通信操作 API，目前仍在开发中。与 MPI 类似，UCC 为集合通信实现了多种通信算法。根据我们的实验，UCX+UCC 的性能与 OpenMPI 持平或更优。*CylonFlow* 将使用 Redis 键值存储，在*Cylon* 参与者之间建立通信通道。

C. 与下游应用共享结果

如第四节-A所述，这种方法允许对集群资源进行分区，并为各个应用安排调度。这些应用可能包含数据依赖关系，例如，多个数据预处理应用向分布式深度学习应用提供数据。然而，这通常会生成DDF，因此将中间结果收集到驱动程序中并不切实际。我们提出一种*CylonFlow* 数据存储（即`Cylon_`存储）抽象，用于保留这些结果。在下面的示例中，data_df和aux_data_df将在两个资源分区上并行执行，而主函数将继续执行深度学习模型。

```
defprocess_aux_data(env:CylonEnv=None, store:CylonStore=
None): 辅助数据 数据帧 = ..._      _store.put("aux_data", aux_
data_数据帧, env=env) def main(env:CylonEnv=None,
store:CylonStore=None): 数据 数据帧 = ..._aux_data_数据帧  =
store.get("aux_data", timeout=..., env=env) 数据帧  =data_数据帧.
merge(aux_data_数据帧, ...) x_train = torch.from_
numpy(df.to_numpy()).to(device) model = Model(...) ... init()
Cylon执行器(parallelism=4).run_Cylon(process_aux_data)
wait(Cylon执行器(parallelism=4).run_Cylon(main))
```

`Cylon_`存储可以由 NFS 或分布式对象存储（例如：Ray 对象存储）提供支持。此功能目前正在 *CylonFlow* 中开发，此处提及仅为完整起见。当应用程序选择不同的并行度值时，可能需要存储对象执行重新分区例程。

D. CylonFlow 功能

所提出的基于 actor 的解决方案 *CylonFlow* 相比传统的类 MPI（整体同步并行，BSP）环境以及分布式计算环境，具有多项优势。

1) 可扩展性：实验表明，在相同硬件上，与采用 AMT 模型的 Dask DDF 和 Ray Datasets 相比，*CylonFlow*-on-Dask 和 *CylonFlow*-on-Ray 能够提供更好的算子可扩展性（第 V 节）。它还超越了采用传统 Actor 模型的 Spark Datasets。*CylonFlow*为数据工程用户提供了一种高性能且具备可扩展性的数据帧替代方案，只需对执行环境进行最少的改动即可用于现有应用。

2) 应用级并行：在分布式计算集群中划分资源，可实现多个 *CylonFlow* 任务的并行调度。与由 DDF 算子组成的典型任务相比，这些任务的并行粒度会大得多。我们计划在未来引入执行计划优化器，将 DF 应用程序的有向无环图拆分为独立的子应用程序（例如，合并整个分支）。随后便可在集群中分别调度这些子应用程序。输出（已完成分区）可以存储在分布式对象存储中，供后续子程序使用。我们可能会处理大型二进制输出，这些输出可以直接作为对象存储，而不必使用对象存储来执行内部通信例程。这种应用级并行也可支持多租户作业提交。

3) 交互式编程环境：Petersohn 等人[4]观察到，交互式编程环境是探索性数据分析的关键。R 和 Python 作为解释型语言，非常适合这种体验。*Cylon* 的一个主要缺点是无法在笔记本（例如 Jupyter）上运行分布式计算。*CylonFlow* 通过让用户获取本地/远程资源（由 Dask/Ray 管理），并以交互方式向其提交*Cylon*程序，轻松解决了这一问题。

4) 无处不在的高性能：*Cylon-Flow*的理念并不局限于分布式计算库，还延伸到超级计算机等更大型的计算环境。我们目前正在为顶级超级计算机开发*CylonFlow* 扩展。我们的最终目标是让高性能可扩展数据工程无处不在，从个人笔记本电脑到百亿亿次级超级计算机均能实现。

V. 实验

以下实验是在一套由 15 个节点组成的 Intel® Xeon® Platinum 8160 集群上进行的。每个节点包含 2 个处理器插槽上的 48 个硬件核心、255GB 随机存取存储器和 SSD 存储，并通过带宽为 40Gbps 的 InfiniBand 互连。使用的软件包括 Python v3.8；Pandas v1.4；*Cylon*（GCC v9.4、OpenMPI v4.1 和 Apache Arrow v5.0）；Dask v2022.8；Ray v1.12；Modin v0.13；Apache Spark v3.3。使用了包含两列 int64 列的均匀随机分布数据，10^9 行（∼16GB），采用列主序格式（Fortran 顺序）。数据的基数（即数据中唯一键所占的比例）为 90%，这构成了基于键的算子最坏的情况。运行这些实验的脚本可在 GitHub [21] 获取。在之前的工作 [3], 中讨论的算子模式中，我们只选取了连接、分组和排序算子。这些涵盖了一些最复杂的

从 DDF 算子设计的角度来看，这些是最复杂的例程。我们只考虑算子耗时（不包括数据加载时间）。输入数据要么从驱动进程加载到工作节点，要么由工作节点自行以 Parquet 文件的形式加载（Dask 和 Apache Spark 不建议采用前一种方式）。随后，数据会根据并行度重新分区并缓存。

我们承认，在实际应用中，仅凭算子的性能未必能全面反映数据工程框架的整体性能。不过，我们认为，这足以用于提出一种替代执行方案。此处仅将 Dask DDF、Ray Datasets、Spark Datasets 和 Modin DDF 作为基准。进行这些测试时，我们尽最大努力参考了公开可用的文档、用户指南和论坛，以获得最优配置。

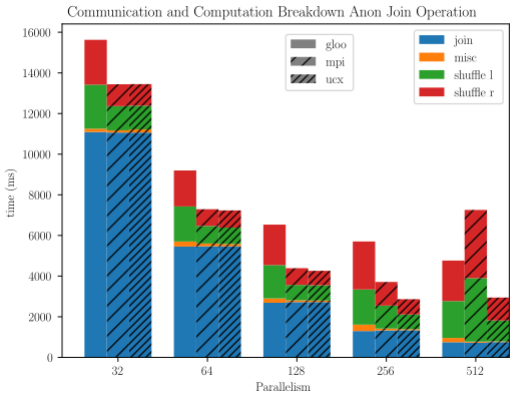

图 6. *Cylon*连接操作的通信与计算分解（10 亿行）

A. 通信与计算

在考察的 3 种算子中，连接操作的通信开销最大，因为它是一个二元算子（有 2 个输入数据帧）。我们研究了通信和计算时间如何随并行度变化。即使在最低并行度（32）下，通信开销也相当显著（Gloo 为 27%，MPI 为 17%，UCX 为 17% ）；随着并行度增加，通信时间占据了实际运行时间的大部分（Gloo 为 76%，MPI 为 86%，UCX 为 69%）。遗憾的是，我们对 Spark、Dask 或 Ray DDF 代码库的了解不足，无法运行类似的微基准测试。但即使使用专为消息传递设计的库，*Cylon* 仍会遇到显著的通信开销。

B. OpenMPI 与 *Gloo* 与 *UCX/UCC* 的比较

在本实验中，我们测试了 *Cylon* 通信器实现（用于连接操作）的可扩展性。如第四节所述，我们无法在分布式计算库中使用 MPI 实现。图 7 证实，我们选择的替代方案 Gloo 和 UCX/UCC 在性能和可扩展性方面表现相当。事实上，在较高并行度下，UCX/UCC 的表现优于 OpenMPI。我们在其他算子基准测试中也观察到了这一趋势。

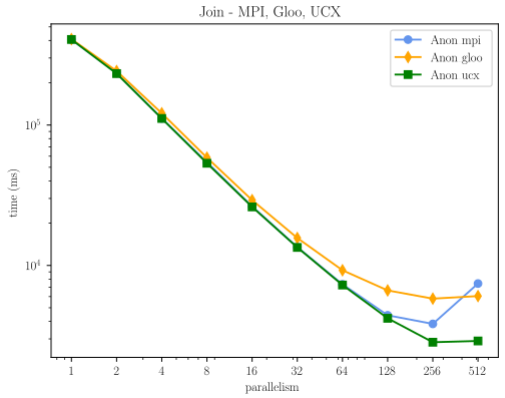

图 7. OpenMPI、Gloo 与 UCX/UCC（10 亿行，对数-对数坐标）——由 mpirun 启动的进程

C. CylonFlow-on-Dask 与 *CylonFlow-on-Ray*

在本实验中，我们展示了所提出的 HP-DDF 方法在分布式计算库（Dask 和 Ray）上的性能，并将其与各自的 DDF 实现（Dask DDF 和 Ray Datasets）进行了比较。遗憾的是，我们在使用 Ray Datasets 时遇到了若干挑战。目前它仅支持一元算子，因此我们无法测试连接操作。此外，Ray 的分组操作在 3 小时内仍未完成，而排序则能给出尚可的结果。我们还纳入了 Apache Spark，因为所提出的方法利用了 Actor 模型。我们启用了 PySpark 的 Apache Arrow 功能，以便进行更具可比性的测试。我们也将 Modin DDF 纳入了比较。不幸的是，它仅支持广播连接，而在两个规模相近的数据框上表现不佳。对于我们的数据集，我们只能让 Modin 在 Ray 后端上运行；排序则会默认使用 Pandas。我们还将 Pandas 的串行性能作为基准进行比较。

观察图 8 中 10 亿行的强扩展运行时间，可以发现 *Cylon*、*Cylon*-on-Dask 和 *Cylon*-on- Ray（使用 Gloo 通信）的表现几乎没有区别。因此，显然，在 Dask/Ray 之上采用所提出的 *CylonFlow* 参与者方法，并不会给原生 *Cylon* HP-DDF 性能带来任何意料之外的开销。Dask 和 Spark Datasets 在连接和排序方面展现出令人称赞的可扩展性，但前者在分组操作中的可扩展性非常有限。我们通过执行 1 亿行测试用例（图 8 底行）进一步研究了 Dask 和 Spark，该用例属于通信受限操作。在这种情况下，随着并行度提高，这两个系统的表现都出现了显著分化，表明其通信实现存在局限性。我们还注意到，Spark 在并行度为 8–32 时的运行时间始终存在异常。我们希望在 Spark 社区的帮助下对此作进一步研究。*CylonFlow* 的可扩展性也有所下降，但梯度平滑得多，且通信性能更好。这些发现进一步支持了我们的建议：采用配备模块化通信器的伪 BSP 环境。事实上，我们的初步测试表明，在相同配置下使用 UCX/UCC 通信器可能会进一步提升性能（第

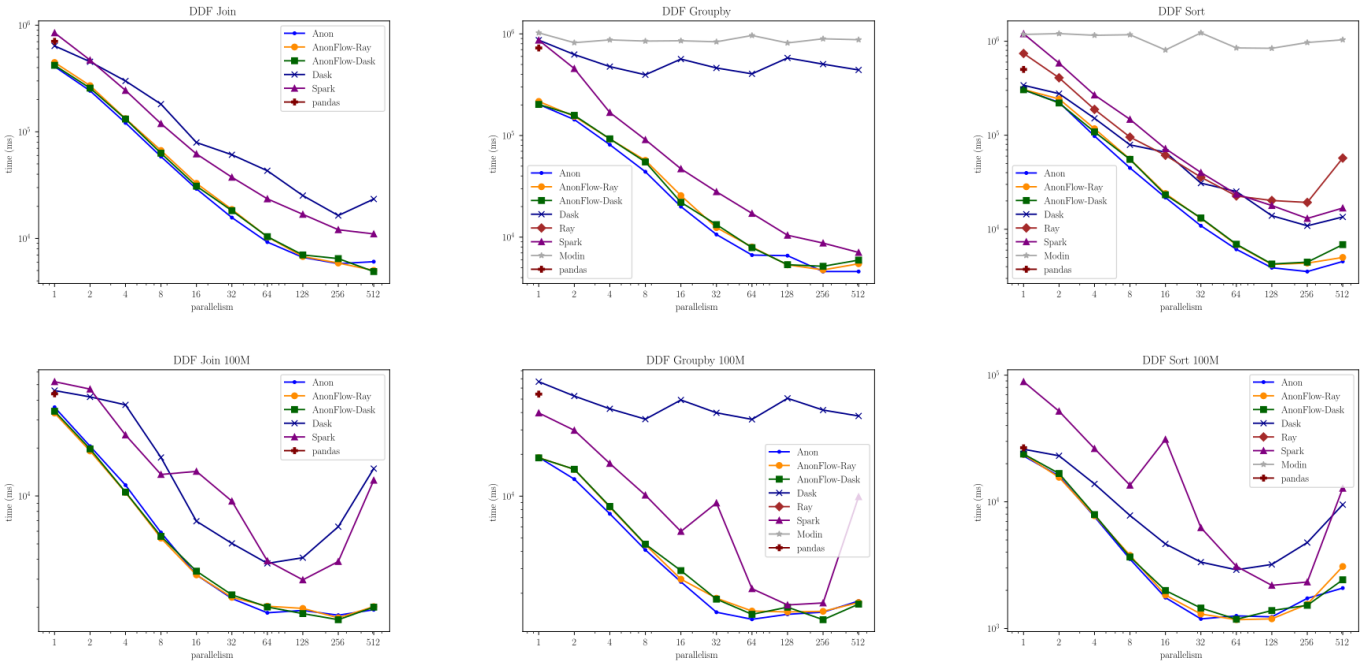

图 8. 强　　DDF算子的扩展性（双对数坐标），上：10亿行，　下：1亿行

有人建议，在相同设置下使用 UCX/UCC 通信器可能会进一步提升性能（第五节 B 部分）。

在并行度为 512 时，对于连接、分组和排序，*CylonFlow* 的平均性能分别比 Pandas 的串行性能$142\times$，$123\times$和 $118\times$ 更好。我们还观察到，*CylonFlow* 的串行性能始终优于其他方案，这可能与*Cylon*的 C++ 实现以及 Apache Arrow格式的使用直接相关。在每种并行度下，*CylonFlow* 的分布式性能始终$2-4\times$ 高于 Dask/Spark。这些结果证实了所提方法的有效性。

D. 算子流水线

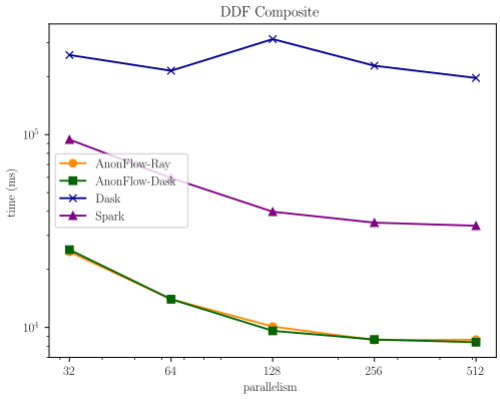

图 9. 算子流水线（10亿行，双对数坐标）

我们还在*CylonFlow*、Dask DDF 和 Spark Datasets 上测试了以下流水线：连接 → 分组 →排序 → 添加_[redacted]如[redacted]示，[redacted]*CylonFlow* 的优势更加明显。相较于 Dask DDF，平均加速比范围为 $10-24\times$；相较于 Spark Datasets，则为 $3-5\times$。如第四节所述，*Cylon* 的执行会合并流水线中位于通信例程之间的所有本地算子，我们认为这是取得这一优势的一个主要原因。

VI. 局限性与未来工作

根据我们在第四节中的发现，在高性能计算和超级计算集群中，使用BSP执行环境是非常常见的用例，而*CylonFlow* 的概念很适合这些环境。我们目前正与Radical-Cybertools和Parsl团队合作，基于工作流管理软件栈，将*CylonFlow* 扩展到顶级超级计算机。此外，我们计划在Akka等纯Actor库之上扩展*CylonFlow*。这将使*Cylon*能够通过Java本地接口（JNI）在JVM上实现原生性能。我们目前正在为*Cylon*和*CylonFlow*添加这些JNI绑定。

在第 V 节中，我们看到通信耗费了大量时间。在现代 CPU 硬件上，我们可以在等待通信结果的同时进行计算。由于一个算子由按有向无环图排列的子算子组成，我们可以通过重叠通信和计算来利用流水线并行 。此外，我们还可以改变计算的粒度，使其适合 CPU 缓存。我们已对这些想法进行了初步研究，

并且发现*Cylon*的性能有了显著提升。第四节提出了一种 *CylonFlow* 数据存储，可与下游应用共享数据。这项工作仍在积极开发中。

在类 MPI 环境中提供容错能力颇具挑战，因为它假设整个应用运行期间通信通道始终正常。这意味着要在通信层面提供容错能力会相当复杂。不过，我们计划加入检查点机制，以实现粒度更粗的容错能力。负载不均衡（尤其是在数据集存在偏斜的情况下）可能导致某些进程得不到足够的工作，从而降低整体吞吐量。为避免此类情况，我们正在开发一种基于采样的重新分区机制。

七、相关工作

在此前的一篇论文中，我们提出了一个用于设计和开发高性能数据工程框架的形式化框架，其中涵盖数据结构、架构和程序模型 [22]。Kamburugamuve 等人提出了一个类似的大数据工具包，名为 *Twister2* [8], ，该工具包基于 Java。作者指出，在数据处理中采用类 BSP 环境可以提升可扩展性，并且还在 Java 中引入了一个名为 *TSets* 的类数据帧 API。然而，*Cylon* 使用 C++ 开发，能够发挥硬件的原生性能，并与 Python 和 R 实现更稳健的集成。*CylonFlow* 作为一个以 Python 构建的扩展，仍能达到与 *Cylon* 相同的性能。

与*Cylon*并行，Totoni 等人也提出了一种类似的 HP-DDF 运行时，名为*HiFrames* [23]。他们主要尝试使用 Numba 为 DDF 算子编译原生 MPI 代码。尽管 *HiFrames* 与*Cylon*在架构上有若干相似之处，但目前后者是唯一可用的开源 HP-DDF。前者仍受限于 MPI，因此在 Dask/Ray 等分布式计算库中使用并不实际。

Horovod 在其*Horovod-on-Ray* 项目[24]中，利用使用 Gloo 通信的 Ray 执行器进行数据并行深度学习。乍看之下，这与*Cylon-Flow*-on-Ray 有许多相似之处，但其 API 仅支持张量通信。*Cylon*/*CylonFlow* 则是一种更通用的方法，可同时支持数据帧和张量。事实上，这些框架可以互为补充，将数据预处理与深度学习整合到同一条流水线中。

除了本文讨论的分布式数据流运行时，我们还想介绍一些令人振奋的新项目。Velox 是一个由 Meta 公司孵化器 [25] 管理的 C++ 向量化数据库加速库。目前，它不提供数据帧抽象，但仍提供图 3 中展示的大多数算子。Photon 是另一个基于 C++ 开发的向量化查询引擎，由 Databricks [26] 开发，旨在为 Apache Spark 生态系统提供原生性能。不幸的是，它尚未向开源社区发布。Substrait 是另一个有趣的模型，试图对数据计算操作进行独立描述 [27]。

VIII. 结论

可扩展的 DataFrame 系统对于现代数据工程应用至关重要，但尽管如此，如今许多系统仍无法达到预期的可扩展性。本文中，作者提出了一种可扩展数据框的替代方案 *CylonFlow*，尝试将高性能计算引入分布式计算运行时。他们提出的有状态伪 BSP 环境和模块化通信器，使其在 Dask 和 Ray 环境中实现了业界领先的可扩展性和性能，从而大幅提升其性能。*CylonFlow* 与 Dask 和 Ray 自带的 DataFrame 系统，以及 Apache Spark 、 Modin 和 Pandas 进行了比较。使用*Cylon* HP-DDFC++后端和 Apache Arrow 格式，使*CylonFlow* 的顺序执行性能优于其他方案。*CylonFlow* 中的模块化通信器支持在 Gloo 和 UCX/UCC 之间切换，以实现 DDF 通信，从而在 Dask/Ray 环境中实现可扩展的分布式性能。本质上，*CylonFlow* 构建了一个无处不在的数据工程生态系统，统一了高性能计算和分布式计算社区。

参考文献

[1] “Pypi 下载统计：Pandas”，https://pypistats.org/packages/pandas。[2] A. authors，“匿名论文1。”[3] ——，“匿名论文 2。”[4] D. Petersohn、S. Macke、D. Xin、W. Ma、D. Lee、X. Mo、J. E. Gonzalez、J. M. Hellerstein、A. D. Joseph 和 A. Parameswaran，“面向可扩展的 DataFrame 系统”，*arXiv* 预印本 *arXiv:2001.00888*，2020年。[5] M. Zaharia、M. Chowdhury、T. Das、A. Dave、J. Ma、M. McCauly、M. J. Franklin、S. Shenker 和 I. Stoica，“弹性分布式数据集：用于集群计算的{容错} 内存中} 抽象”，载于第九届 *USENIX* 网络系统设计与实现研讨会（*NSDI 12*），2012年，第15–28页。[6] L. G. Valiant，“并行计算的桥接模型”，*ACM*通讯，第33卷，第8期，第103–111页，1990年。[7] G. Fox、M. Johnson、G. Lyzenga、S. Otto、J. Salmon、D. Walker 和 R. L. White，“在并发处理器上求解问题，第1卷：通用技术与规则问题”，《计算物理》，第3卷，第1期，第83–84页，1989年。[8] S. Kamburugamuve、K. Govindarajan、P. Wickramasinghe、V. Abeykoon 和 G. Fox，“Twister2：大数据工具包的设计”，《并发与计算：实践与经验》，第32卷，第3期，第e5189页，2020年。[9] J. Armstrong，“Erlang”，*ACM*通讯，第53卷，第9期，第68–75页，2010年。[10] Anaconda，“《2020年数据科学现状》——anaconda.com”，2021年。 [在线] 2020 。可访问：[11] https://www.anaconda.com/state-of-data-science- S. Abiteboul、R. Hull 和 V. Vianu，数据库基础。Addison-Wesley Reading，1995年，第8卷。[12] “Arrow列式格式——Apache Arrow v9.0.0”，https://arrow.apache.org/docs/format/Columnar.html。[13] “Catalyst 和 Tungsten：Apache Spark 的加速引擎”，https://www.linkedin.com/pulse/catalyst-tungsten-apache-sparks-speeding-engine-deepak-rajak。[14] “用于分组和连接的 Shuffle——Dask文档”，https://docs.dask.org/en/stable/dataframe-groupby.html。[15] “欢迎使用 Ray 文档——Ray 2.0.0”，https://docs.ray.io/en/latest/index.html。[16] J. Bruck、C.-T. Ho、S. Kipnis、E. Upfal 和 D. Weathersby，“多端口消息传递系统中全互通通信的高效算法”，《*IEEE*并行与分布式系统汇刊》，第8卷，第11期，第1143–1156页，1997年。[17] R. Thakur、R. Rabenseifner 和 W. Gropp，“MPICH 中集合通信操作的优化”，《国际高性能计算应用期刊》，第19卷，第1期，第49–66页，2005年。

[18] J. L. Tr¨aff、A. Rougier 和 S. Hunold，“经典实现：使用 MPI 数据类型进行零拷贝全互通通信”，载于第*28*届 *ACM* 超级计算国际会议论文集，2014年，第135–144页。[19] “facebookincubator/gloo：用于多机训练、包含各种原语的集合通信库。”https://github.com/facebookincubator/gloo。[20] P. Shamis、M. G. Venkata、M. G. Lopez、M. B. Baker、O. Hernandez、Y. Itigin、M. Dubman、G. Shainer、R. L. Graham、L. Liss 等，“UCX：面向 HPC 网络 API 及更多领域的开源框架”，载于*2015* 年 *IEEE* 第 *23* 届高性能互连年度研讨会。IEEE，2015年，第40–43页。[21] A. authors，“匿名实验仓库”，待办。[22] ——，“匿名论文 3。”[23] E. Totoni、W. U. Hassan、T. A. Anderson 和 T. Shpeisman，“HiFrames：脚本语言中的高性能数据框”，*arXiv* 预印本 *arXiv:1704.02341*，2017年。[24] “Ray 上的 Horovod——Horovod 文档”，https://horovod.readthedocs.io/en/stable/ray include.html。[25] P. Pedreira、O. Erling、M. Basmanova、K. Wilfong、L. Sakka、K. Pai、W. He 和 B. Chattopadhyay，“Velox：Meta 的统一执行引擎。”[26] A. B̈ehm、S. Palkar、U. Agarwal、T. Armstrong、D. Cashman、A. Dave、T. Greenstein、S. Hovsepian、R. Johnson、A. Sai Krishnan等，“Photon：面向湖仓系统的快速查询引擎”，载于*2022* 年国际数据管理会议论文集，2022年，第2326–2339页。[27] “substrait-io/substrait：一种跨平台方式，用于表达数据转换、关系代数、标准化记录表达式和计划。”https://github.com/substrait-io/substrait。