计算机科学与探索 1673-9418/2000/00(00)-0000-00
Journal of Frontiers of Computer Science and Technology doi: 10.3778/j.issn.1673-9418.0000000

容器云平台下的数据编排与分布式训练系统

摘　要: 随着机器学习和云计算的迅速发展，传统的分布式训练方法已经不能满足现有的需求，算法研究者与工程师们也都在尝试利用 Kubernetes 进行分布式训练。现有的方法需要用户编写资源对象文件，并构建训练镜像。用户和代码之间的交互性较差，训练结果无法直观地显示给用户。另外，由于不同存储系统拥有不同的调用接口，当发生数据集迁移时，就需要去修改大量的代码。同时，分布式训练读取磁盘数据时，存在 I/O 瓶颈问题，使得资源得不到充分利用。为了解决上述问题，设计并实现了一种利用自定义 Jupyter 内核的数据编排和分布式训练系统。一方面基于自定义 Jupyter 内核与 Operator 机制实现了用户与资源对象文件的完全隔离，用户无需编写任何资源对象文件。另一方面利用 Alluxio 进行数据编排，将数据编排技术应用到分布式训练中，提高了系统分布式训练过程中的资源利用率和数据集读写速率。与现有方法相比，系统保证了用户与云平台分布式训练的交互性，同时实验结果表明，系统完成分布式训练的平均时间比利用现有方法 Kubeflow 减少了 28%，平均成功率提高了 15%，具有更高的效率。

关键词: 深度学习云平台；云计算；分布式训练；Kubernetes；数据编排
文献标志码：A　　中图分类号：TP182

Data Orchestration and Distributed Training System under the Container Cloud Platform

Abstract: With the rapid development of machine learning and cloud computing, traditional distributed training methods can no longer meet the existing needs, and algorithm researchers and engineers are also trying to use Kubernetes for distributed training. Existing methods require users to write resource object documents and build training images. The interaction between the user and the code is poor, and the results of the training cannot be intuitively displayed to the user. In addition, because different storage systems have different call interfaces, when dataset migration occurs, a large amount of code needs to be modified. At the same time, when distributed training reads disk data, there is an I/O bottleneck, which makes the resources underutilized. In order to solve the above problems, this paper proposes a data arrangement and distributed training system using the custom Jupyter kernel. On the one hand, the user and resource object files are completely isolated based on the custom Jupyter kernel and Operator mechanism, and users do not need to write any resource object files. On the other hand, it is proposed to use Alluxio for data arrangement, apply data arrangement technology to distributed training, and improve the reading rate and resource utilization rate of data sets in the process of system distributed training. Compared with the existing methods, the system of this article ensures the interaction between users and the distributed training of the cloud platform. At the same time, the experimental results show that the average time for this system to complete distributed training is 28% less than that of using the existing method Kubeflow, and the average success rate is increased by 15%, which has higher efficiency.

Key words: Deep learning cloud platform; Cloud Computing; Distributed training; Kubernetes; Data organization

1 引言

随着云计算的发展，容器技术[1][2]和分布式训练[3]的结合也越来越受到许多互联网公司的青睐。它们都利用容器技术完成分布式训练，但是提供给用户的服务也是很有限的。一方面，现有的技术Kubeflow[4]需要用户构建目标镜像才能创建训练容器，编写大量资源对象文件才可以进行任务训练。当训练结束后，用户无法直接获取训练结果，而是需要通过命令行去读取训练容器的日志，这些对于用户并不友好，缺少了必要的交互性。一方面，现有基础训练镜像都存在文件库不完整的问题，当出现这种情况时，用户只能不断对镜像进行重新构建，无法安装自己缺失的库文件。另一方面，分布式训练读写磁盘数据存在 I/O 瓶颈问题[5]，可能导致训练任务的训练时间延长，资源得不到充分地利用，降低了资源的利用率。

为此，本文提出了一种基于自定义 Jupyter 内核与容器技术的数据编排与分布式训练系统的设计与实现方法。通过自定义 Jupyter 内核与 Kubernetes 集群[6][7]交互，用户无需编写资源对象文件、设置环境变量和构建训练镜像就能在 Kubernetes 集群中创建训练容器和进行数据编排，减少了训练所花费的总时间。训练结束后，自动将训练结果返回 Jupyter 前端，用户可以方便地得到训练结果。数据编排技术通过将数据缓存到内存，从而加速了数据读写速率，缓解了数据读写速率的瓶颈问题，缩短了分布式训练的时间。实验表明，利用本文构建的系统进行分布式训练的平均时间比现有方法 Kubeflow 缩短了 28%，平均成功率提高了 15%。本文的主要贡献包括以下几个方面：

(1) 设计实现了一种基于 Kubernetes 和自定义 Jupyter 内核的分布式训练方法。用户不需要构建任何其他镜像，利用自定义 Jupyter 内核传递用户的请求，对用户的请求进行解析。通过解析的结果创建训练容器，并在训练容器中安装相应的库文件。用户利用这种方法，解决了现有训练镜像存在的库文件不完整问题，能够更加简单高效地利用 Kubernetes 集群进行一次完整的分布式训练，缩短了分布式训练所花费的总时间。

(2) 利用数据编排技术，实现使用统一接口对不同的分布式存储系统进行管理，在训练过程中通过内存对数据进行读写，改善了 I/O 瓶颈问题，将计算与数据存储隔离，提高了资源的利用率。

(3) 利用 Kubernetes Operator 设计自定义 Kubernetes 资源对象，实现对所有分布式训练资源对象的统一创建，解决了需要编写大量资源对象文件的问题，同时提高了训练成功率。

2 相关工作

2.1 基于 Operator 的分布式训练技术

传统的利用 Kubernetes 集群进行分布式训练的方法是利用 Kubernetes 自带的资源对象 Job 完成的。但是，这种方法需要用户编写不同的配置文件从而启动多个不同的 Job 作为参数服务器和工作服务器，它们之间互相配合进行分布式训练。另外，每个 Job 被 Kubernetes 调度启动之前，无法固定训练容器的 IP 地址，因此用户还需另外创建资源对象文件 Service，显然增加了用户的工作量和出错率。

Kubernetes Operator 是用来扩展 Kubernetes API

的特定应用程序控制器[8]，它用来创建、配置和管理复杂的有状态应用，如数据库、缓存和监控系统。Operator 基于 Kubernetes 的资源和控制器概念之上构建，但同时又包含了应用程序特定的领域知识。创建 Operator 的关键是 CRD（自定义资源）的设计。Kubernetes Operator 结构图如图 1 所示。本文构建的系统基于 Kubernetes Operator 实现了容器分布式训练功能，简化了系统的复杂度。

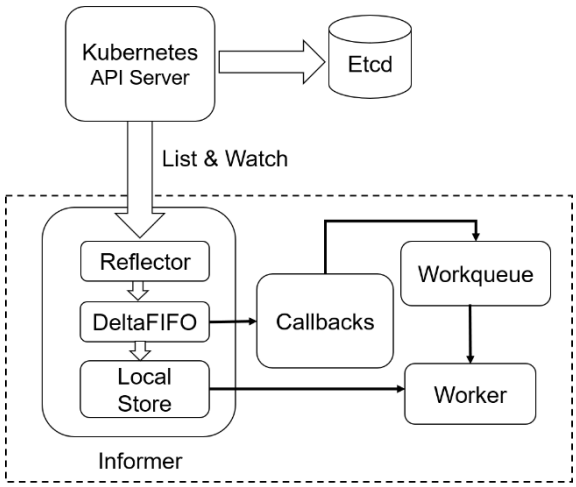

Fig.1 Architecture of Kubernetes Operator

图 1 Kubernetes Operator 结构图

2.2 自定义 Jupyter 内核

Jupyter Notebook 基于一组交互式计算的开放标准[9]，将 HTML 和 CSS 用于 Web 上的交互式计算。开发人员可以利用这些开放标准来构建具有嵌入式与交互式计算的定制应用程序。Jupyter 内核是使用特定编程语言运行交互式代码并将输出返回给用户的进程。Jupyter Notebook 结构图如图 2 所示。

Jupyter 魔术命令是针对一些操作而设计的一种简洁的命令。它特定于 Jupyter 内核并由内核提供。Jupyter 魔术命令在内核上是否可用是内核开发人员基于每个内核做出的决定。本文提出的系统实现自定义 Jupyter 内核，利用内核与 Kubernetes 集群交互，传递并解析相关魔术命令，实现了分布式训练与数据编排功能。

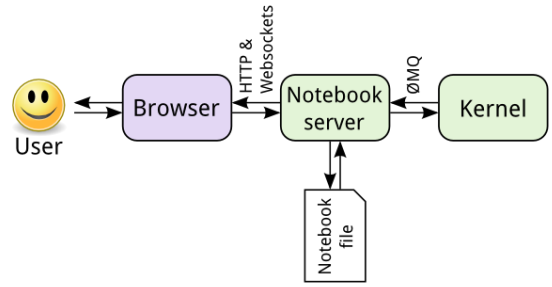

Fig.2 Architecture of Jupyter Notebook

图 2 Jupyter Notebook 架构图

2.3 数据编排技术

随着新型计算、存储框架以及云计算的迅速发展，整个数据生态系统的复杂度也在不断提高。当工程师或科学家想要编写应用程序来解决问题时，需要花费大量精力让应用程序高效地访问数据，而不是专注于算法和应用程序的逻辑。

数据编排技术从根本上实现了存储和计算的分离。对于不同存储系统的管理，数据编排技术实现了统一的接口进行数据管理，同时利用内存对数据进行读写，加速了对数据的读写速率，改善了数据读写的 I/O 瓶颈问题。Alluxio 是一个开源数据编排平台[10]，用于云中的数据分析和 AI/ML。数据编排平台 Alluxio 的结构图如图 3 所示。本文提出的系统结合 Alluxio 与 Jupyter 内核实现了交互式的数据编排功能，同时与分布式训练结合，实现了分布式计算与存储的隔离。

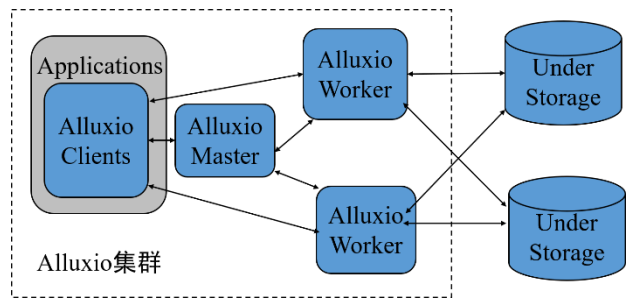

Fig.3 Architecture of Data Orchestration Platform Alluxio

图 3 数据编排平台 Alluxio 架构图

3 系统的设计与实现

在本节中，将主要对所提数据编排与分布式训练系统的结构进行详细的描述。系统的整体流程如图 4 所示，对应的前端交互界面设计如图 5 所示。系统接收用户输入的模型代码，分布式训练和数据编排请求，包括利用魔术命令指定的模型训练框架，例如 Tensorflow[11]，Pytorch[12]等，参数服务器和工作服务器的数量，以及每个服务器所需要分配的资源、数据集挂载路径等，请求将会通过 ZeroMQ[13] 传递到 Jupyter 内核，并且内核也会通过 ZeroMQ 将训练结果返回到 Jupyter 前端，最终直观地显示给用户。

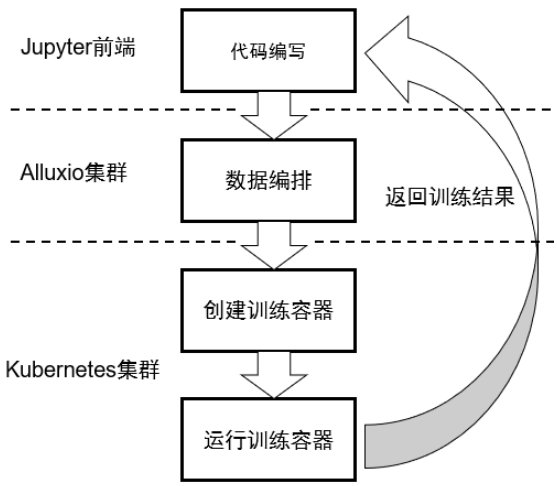

Fig.4 The flow chart of the system

图 4 系统流程图

Fig.5 System front-end interactive design

图 5 系统前端交互图

系统的整体框架如图 6 所示。系统主要包括两大模块：数据编排模块和训练容器管理模块。在接下来的小节中，将对所提数据编排模块和训练容器管理模块进行详细地描述。其中在第 3.1 节中将主要介绍数据编排模块。在 3.2 节中主要介绍训练容器管理模块。最后将在 3.3 节中介绍训练结果的读取。

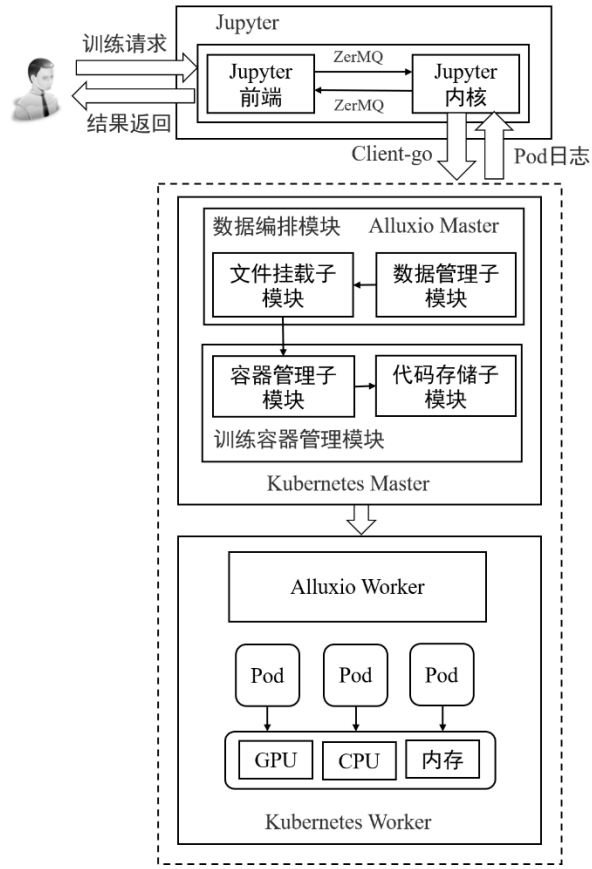

Fig.6 Framework of data orchestration and distributed training system

图 6 数据编排与分布式训练系统架构图

3.1 数据编排模块

数据编排模块能够通过 Jupyter 内核传递过来的信息与 Alluxio 集群交互。基于数据编排系统，分布式训练可以仅使用一种调用接口就可以读写不同存储系统上的数据集，同时读写文件的速度更快，缓解了分布式训练读写数据集存在的 I/O 瓶颈问题，

提高了资源利用率。该模块实现了通过魔术命令与数据编排系统交互，将数据和分布式计算框架分离。数据编排模块由文件挂载子模块和数据管理子模块构成。

3.1.1 文件挂载子模块

文件挂载子模块能够通过用户在魔术命令中定义的挂载规则将数据集所在的存储路径挂载到 Alluxio 文件系统中。当任务的数据集发生数据迁移时[14]，无需修改与读取数据相关的代码，也无需额外地配置环境。只需修改魔术命令重新挂载文件即可，避免了对代码地大量修改。该模块实现了利用 Jupyter 将分布式训练任务与数据编排技术结合，训练的时间不会由于 I/O 性能瓶颈而延长，提高了资源利用率。同时统一了数据访问接口，解决了由于数据迁移所造成的访问接口不一致的问题。

3.1.2 数据管理子模块

数据管理子模块实现在 Jupyter 上管理不同的数据存储系统。利用魔术命令对不同的存储系统上的数据进行相关操作。该模块将数据与计算分离，用户不需要配置不同的存储环境，不用调用不同存储系统的接口就可以与不同存储系统进行交互。

3.2 训练容器管理模块

训练容器管理模块通过 Jupyter 内核传递过来的信息去创建用于分布式训练的自定义 Kubernetes 资源对象。用户通过容器管理模块可以选择 Tensorflow 或 Pytorch 分布式训练框架，指定 Master，PS 和 Worker 的数量以及分配给容器的资源。用户在 Jupyter 中编写的代码通过 Jupyter 内核传递，存储在 Kubernetes 的资源对象 ConfigMap 中，实现了代码的持久化保存，同时无需重新构建镜像即可进行分布式训练。训练容器管理模块由容器管理子模块和用户代码存储子模块构成。

3.2.1 容器管理子模块

容器管理子模块主要能够通过用户在 Jupyter 上定义的魔术命令去创建和运行自定义训练资源对象，同时将资源对象的运行结果直接返回并显示到前端。用户通过魔术命令定义需要的分布式训练框架，指定不同类型服务器数量和分配给容器的资源，可指定分配的资源有 GPU，CPU 和内存。该模块实现了在 Jupyter 上使用自定义分布式训练资源对象进行分布式训练的功能，解决了用户在利用现有方法 Kubeflow 进行分布式训练时需要用户构建训练镜像和编写大量资源对象文件的问题。

3.2.2 代码存储子模块

用户代码存储子模块能够将用户在 Jupyter 上编写的训练代码存储在资源对象 ConfigMap 中。资源对象 ConfigMap 存储在用户指定的 Kubernetes 的 Namespace 下，名称与创建的自定义训练资源对象名称一致。将文件存储在 ConfigMap 中后，自定义训练资源对象利用 Kubernetes 的文件挂载功能读取并运行代码，无需用户将训练代码和环境一起重新构建成镜像，上传至镜像仓库。该模块主要基于 Kubernetes 的资源对象 ConfigMap 和 Kubernetes 的容器文件挂载功能，实现了用户代码地持久化保存，解决了由于训练代码地修改而需要重新构建训练镜像的问题。

3.3 训练结果读取

自定义 Jupyter 内核在目标容器训练完成后，利用工具库 Client-Go 读取目标容器的运行日志，并将日志中的训练结果返回 Jupyter 前端。用户在 Jupyter 上提交代码后就可以在 Jupyter 上看到训练结果，无需通过命令行去手动读取训练容器的日志，解决了使用现有方法 Kubeflow 进行分布式训练缺少交互性的问题。

4 实验结果和分析

4.1 实验设计

本次实验利用不同的分布式训练模型对MNIST数据集进行分布式训练。实验环境如表1所示。为验证本系统可以在不影响训练准确率的前提下减少训练的时间成本，并提高训练成功率，本文与现有方法Kubeflow进行了对比实验。

第1组实验利用不同训练模型对MNIST数据集进行分布式训练，比较了分别利用现有方法Kubeflow和利用本文系统进行分布式训练的训练准确率以及训练总时间。第2组实验分别利用现有方法Kubeflow与本文系统对MNIST数据集进行训练，统计在不同影响因子 α 下的训练成功率。为了保证实验的准确性，在利用本文系统和现有方法Kubeflow进行训练时，设置的训练参数服务器数量以及工作服务器数量都是相同的。

Table 1 Cluster hardware configurations

表1 集群硬件配置

配置	型号或大小
CPU	Intel(R) Xeon(R) CPU E52690 v4
RAM	256GB
Memory	4GB
GPU	NVIDIA Tesla V100

4.2 结果分析

4.2.1 训练准确率分析

利用不同训练模型对MNIST数据集进行分布式训练，通过交叉熵误差对比准确率，其中交叉熵误差公式为：

$$L=-\frac{1}{N}\sum_i\sum_{c=1}^{M}y_{ic}\ln(p_{ic}) \tag{1}$$

其中，M是类别的数量，y_{ic} 是符号函数，p_{ic} 是观测样本 i 属于类别 c 的预测概率。实验结果如图7所示。

从实验结果可以看出，利用本文系统与现有方法Kubeflow分别对不同模型进行分布式训练，两者的训练准确度几乎相同。

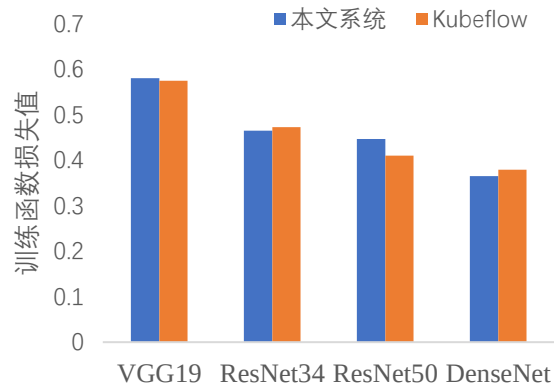

Fig.7 Comparison chart of loss value of cross entropy function

图7 交叉熵函数损失值对比图

4.2.2 训练时间分析

利用容器进行分布式训练所花费的时间主要包括构建镜像的时间与实际训练时间。实验分别利用现有方法Kubeflow与本文系统进行分布式训练，计算训练花费总时间 T 和影响因子 α。影响因子 α 代表构建镜像消耗时间占训练总花费时间的比例，训练总时间与影响因子 α 的计算公式分别是：

$$T=t_b+t_t \tag{2}$$

$$\alpha=\frac{t_b}{T} \tag{3}$$

其中t_b代表构建镜像所消耗的时间，t_t代表训练的实际训练时间。影响因子 α 的值越小，说明镜像构建时间对于训练总时间的占比越小。实验统计了利用不同训练模型对MNIST数据集进行分布式训练的训练总时间，实验结果如图8所示。

从实验结果可以看出，利用本文系统与现有方法Kubeflow分别对不同模型进行分布式训练，本

文所提出的系统所花费的训练总时间明显少于现有方法 Kubeflow，分布式训练的平均完成时间比利用现有方法 Kubeflow 减少了 28%。

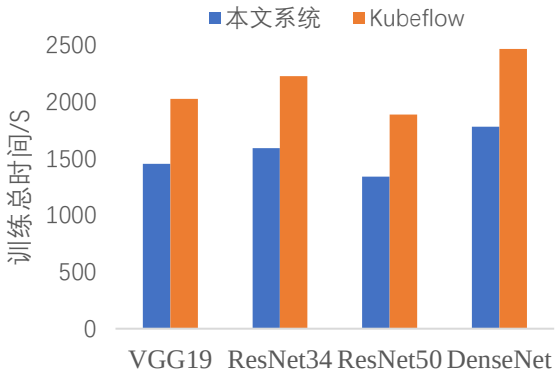

Fig.8 Comparison chart of average completion time of training

图 8 训练平均完成时间对比图

4.2.3 成功率分析

一次成功请求代表训练容器的成功创建与运行，并且最终可以将训练结果返回。训练成功率代表了系统的稳定性，太低的成功率会导致系统的各项功能和数据没有意义。本次实验在不同的影响因子 α 下对训练成功率进行实验，实验结果如图 9 所示。训练成功率的计算公式为：

$$Succ = \frac{T_{succ}}{T_{total}} \times 100\% \qquad (4)$$

其中T_{succ}代表成功请求的次数，T_{total}代表请求总次数。

从实验结果可以看出，随着影响因子 α 的增大，利用现有方法 Kubeflow 进行分布式训练的成功率一直是在下降的。当 α 增大时，意味着镜像构建时间也越长，增加了镜像构建失败的概率，从而也会增加分布式训练失败的概率。而本文构建的系统随着影响因子 α 的增大，表现的更加稳定。原因是本文系统中只需利用准备好的基础镜像创建容器，在训练中提前在容器中下载库文件即可，无需重新构建镜像，减少了镜像构建失败的概率，从而提高了分布式训练的成功率。

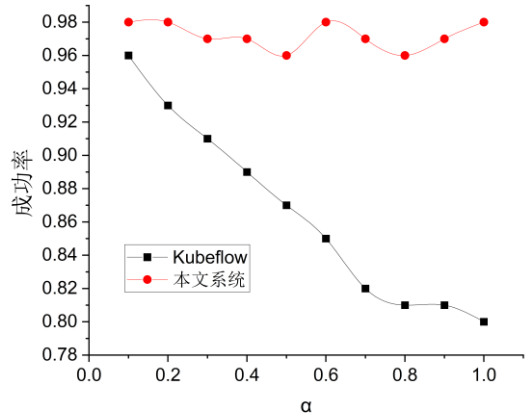

Fig.9 Distributed training success rate comparison chart

图 9 分布式训练成功率对比图

对比发现，本文构建的系统比现有方法 Kubeflow 在不同影响因子下的平均训练成功率提高了 15%。

5 结论与展望

本文设计实现了一种交互式的容器分布式训练与数据编排系统，基于 Kubernetes Operator 机制设计实现了一种针对分布式训练的 Kubernetes 资源对象，利用自定义 Jupyter 内核与 Kubernetes 集群交互，结合数据编排技术，利用内存提高了分布式训练过程中读写数据的速度，进一步缩短了分布式训练所需要的总时间，提高了资源利用率和利用容器进行分布式训练的效率。同时，本文构建的系统设计并实现了自定义 Jupyter 内核和一种针对分布式训练的 Kubernetes 资源对象，提高了系统整体的交互性，为用户提

供更加开放合理的分布式训练服务。

未来研究方向：(1) 设计容器资源调度算法，基于 Scheduling Framework[15]实现容器资源调度器，提高集群资源利用率；(2) 设计实现 Kubernetes 资源对象，集成更多的常用分布式训练框架，例如 Keras[16]和 MXNet[17]等；(3) 实现数据处理、特征工程、模型架构选择、超参数优化[18]等深度学习功能，实现一站式深度学习容器云平台，提高模型构建效率。

参考文献：

[1] M. Deng, R.Yang and Y. Li. Analysis and prospect of automated container terminal technology development[C]//2021 3rd International Academic Exchange Conference on Science and Technology Innovation, 2021: 965-969.

[2] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux journal, 2014, 2014(239): 2.

[3] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, et al. Optimizing distributed training deployment in heterogeneous GPU clusters[C]//Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies. 2020: 93–107.

[4] Kubeflow 2021. The Machine Learning Toolkit for Kubernetes. https://www. kubeflow.org/.

[5] Manu Shantharam, Mahidhar Tatineni, Dongju Choi, et al. Understanding I/O Bottlenecks and Tuning for High Performance I/O on Large HPC Systems: A Case Study[C]//In Proceedings of the Practice and Experience on Advanced Research Computing. 2018: 1–6.

[6] Kubernetes: Production-Grade Container Orchestration 2021. https://kubernetes.io/.

[7] Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at Google with Borg[C]//Proceedings of the Tenth European Conference on Computer Systems. 2015: 1-17.

[8] R. Duan, F. Zhang and S. U. Khan. A Case Study on Five Maturity Levels of A Kubernetes Operator[C]//2021 IEEE Cloud Summit. 2021: 1-6.

[9] Kluyver T, Ragan-Kelley B, Pérez F, et al. Jupyter Notebooks-a publishing format for reproducible computational workflows[M]. 2016: 87-90.

[10] X. Chang and L. Zha. The Performance Analysis of Cache Architecture Based on Alluxio over Virtualized Infrastructure[C]//2018 IEEE International Parallel and Distributed Processing Symposium Workshops. 2018: 515-519.

[11] Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning[C]//12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). 2016: 265-283.

[12] Paszke A, Gross S, Massa F, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library[C]//Advances in Neural Information Processing Systems. 2019: 8024-8035.

[13] PU F P, CHEN J Z, Distributed system based on ZeroMQ[J]. Electronic Test, 2012(7): 24-29.
蒲凤平, 陈建政. 基于 ZeroMQ 的分布式系统[J]. 电子测试, 2012(7): 24-29.

[14] Laila Bouhouch, Mostapha Zbakh, and Claude Tadonki. 2019. Data Migration: Cloudsim Extension[C]//In Proceedings of the 2019 3rd International Conference on Big Data Research. 2019: 177–181.

[15] Li D, Wei Y, Zeng B. A Dynamic I/O Sensing Scheduling Scheme in Kubernetes[C]//Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications. 2020: 14–19.

[16] Mathias Lux and Marco Bertini. Open source column: deep learning with Keras[J]. SIGMultimedia Rec. 2018, 10(4): 1.

[17] MXNet: A Scalable Deep Learning Framework 2021. https://mxnet.apache.org/

[18] Yang L, Shami A. On hyperparameter optimization of machine learning algorithms: Theory and practice[J]. Neurocomputing, 2020, 415: 295–316.