Online Container Scheduling for Data-intensive Applications in Serverless Edge Computing

摘要

边缘环境的计算与数据分散性和设备与网络异构性使得当前的无服务器计算平台无法适应网络边缘。本文介绍了一种在线容器调度算法，旨在解决边缘计算中的容器放置和数据流路由问题，以实现对数据密集型应用的高效调度。ODCS 算法通过将容器放置成本和数据传输延迟的组合作为目标函数，结合单源最短路径算法和贪心策略，实现了对容器请求的在线处理和优化调度。同时，本文还对该算法进行了性能评估，该算法在局部优化和全局优化方面具有有效性和优越性。通过本文的研究和实践，可以为边缘计算中的容器调度问题提供实际可行的解决方案，并为相关领域的研究和实践工作提供有益的参考。

关键词：边缘计算；容器放置

1 引言

物联网（IoT）的逐步普及和人工智能（AI）应用的快速普及使得边缘计算变得空前重要。通过在网络边缘部署计算和存储能力，许多时延敏感的物联网应用可以从远程云端卸载至边缘用户附近，从而显著减少业务延迟和骨干网压力。此外，边缘计算允许在本地处理物联网生成的大量数据，大大缓解了阻碍数据密集型服务发展的传输和隐私问题。

无服务器计算 [1] [2]（FaaS）正在边缘计算领域兴起。每个应用都以一个或多个函数的形式实现，并根据用户事件进行初始化和执行，使用户能够专注于应用开发，而将管理操作留给服务提供商。将无服务器范式从云端扩展到网络边缘，可为本地边缘应用带来新的效率、灵活性和可扩展性。在这种范例中，边缘应用是作为封装在轻量级容器中的功能来实现的，这些容器可以根据工作负载在边缘服务器上动态启动或停止，从而大大提高了开发人员的灵活性和透明度，促进了可移植性，缩短了产品上市时间。

工业界和学术界做出了各种努力，旨在将无服务器服务模型集成到边缘环境中。然而，无服务器边缘计算平台 [3]，尤其是针对边缘人工智能应用等数据密集型服务的无服务器边缘计算平台，仍处于早期阶段，这主要是由于边缘环境中计算和数据在地理上的分散性，以及设备和网络的异构性。将容器放置在数据源附近可能会导致严重的执行延迟，而将数据传输到具有足够计算资源的边缘设备或远程云可能会导致明显的延迟和隐私问题。

目前，一些创新性的工作已经提出了数据感知的容器编排策略和系统，旨在填补无服务器范式和网络边缘之间数据密集型应用程序的差距。但是，现有的解决方案并不足以解决上述挑战，它们尚未充分考虑地理分布和异构性质的无服务器边缘计算网络的多种独特特征。现有工作未充分考虑到网络边缘与云之间可能存在显著差异的异构网络类型。例如，边缘服务器和数据位置之间的连接可能不是通过直接的路由器实现的，而是通过复杂的路由路径连接的。除了容器调度之外，协同路由数据流以实现低传输延迟也是一个复杂而至关重要的问题。尽管无服务器计算平台致力于简化，但仍需要精力对容器进行维护和编排。考虑到网络边缘的能源和资源都非常有限，因此无服务器边缘计算的额外运营成本是不容忽视的。此外，由于存在无服务器运营成本，因此在远程云中初始化容器并向其传输相应的数据并不一定比在边缘数据位置附近启动容器更差。我们迫切需要一个模型，该模型可以在边缘和云之间平衡延迟和运营成本，以实现最佳的容器调度结果。

2 相关工作

许多研究也都集中在无服务器计算的当前挑战和开放性问题上。其中，一个重要的课题是如何使无服务器计算范式适应边缘环境，因为它在众多应用场景中具有巨大潜力。在学术界，许多学者还提出了算法、系统和架构研究。

2.1 容器编排平台

容器编排平台是一种用于管理和编排容器的软件平台，它可以自动化地部署、扩展和管理容器化应用程序。容器编排平台通常包括一个容器编排引擎，用于自动化地部署和管理容器，以及一个容器编排工具，用于管理和监控容器化应用程序。Xiong 等人在 Kubernetes 中扩展了一系列组件，使容器编排平台适应边缘环境，并将修改后的系统命名为 KubeEdge [4]。Nastic 等人在 [5] 中提出了一个无服务器实时数据分析平台，而 Baresi 等人在 [6] 中为无服务器边缘计算构建了一个实用的网络架构。根据这些开创性研究，无服务器边缘计算平台可以总结为多种前沿虚拟化技术的巧妙组合和扩展，例如 docker、OpenFaaS 和 Kubernetes，部署在互连的边缘设备上，以提供低延迟、高度灵活且具有成本效益的边缘原生应用程序，如图 1 所示。

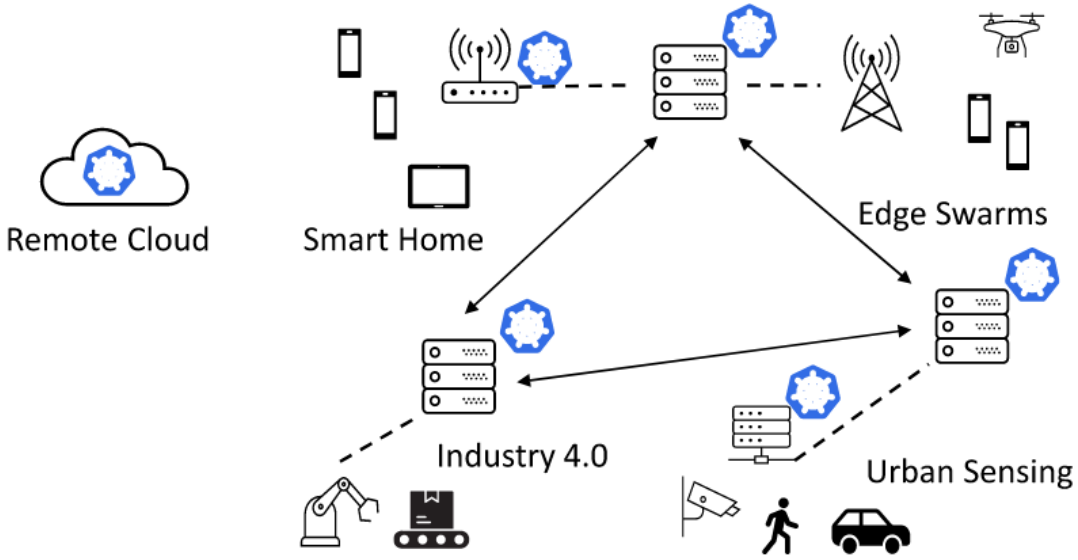

图 1. 无服务器边缘计算网络演示

2.2 容器放置问题

容器放置问题主要关注如何在边缘设备上高效地运行容器。现有的研究工作主要集中在如何选择合适的边缘设备来放置容器，以最大化性能和资源利用率。为了的计算和数据地理分布对无服务器边缘计算网络的负面影响，Pan 等人在 [7] 中提出了一种容器缓存与请求分发问题，并设计了在线算法来解决该问题。Rausch 等人在 [8] 中提出了一种数据感知容器编排系统，以适应数据密集型边缘功能与边缘环境。然而，鉴于各种异构边缘网络类型，现有工作忽略了数据流路由的必要性，并且没有考虑无服务器边缘平台的运营成本。

2.3 数据流路由问题

该问题主要关注如何在边缘网络中高效地传输数据。现有的研究工作主要集中在如何选择最佳路径来传输数据，以最小化传输延迟和能耗。例如，[9] [10] [11] 中的研究工作主要关注如何在网络拓扑结构中选择最佳路径。

3 本文方法

3.1 模型构建

为了克服无服务器边缘计算中计算、数据分散和设备、网络异构的问题，作者提出了联合容器放置和流路径问题，考虑了一个数据密集型应用的无服务器边缘计算网络。该模型反映了数据密集型应用在不同边缘设备上的不同执行和传输延迟。它还考虑了边缘处的各种网络类型，以进行详细的数据流路由，所有的重要符号如表 1 所示。

Notation	Definition
N	Set of nodes in the network, N = 1, ..., n, ..., |N |
I	Set of containers in the network, I = 1, ..., i, ..., |I|.
e_{n_1,n_2}	The communication link connecting node n_1 to n_2
E	Set of communication links.
$x_{i,n} \in \{0,1\}$	Decision variable whether container i is placed on node n.
$y_{i,n} \in \{0,1\}$	Decision variable whether data of container i is retrieved from node n.
$z_n \in \{0,1\}$	Decision variable whether node n is enabled for serverless computing.
$f_{i,n_1,n_2} \in \{0,1\}$	Decision variable whether flow between i and data passes through e_{n_1,n_2}
w_{i,n_1,n_2}	Transmission delay between container i and corresponding data pass through e_{n_1,n_2}
$\alpha_{i,n}$	Execution delay introduced by placing container i on node n.
$\beta_{i,n}$	Constant marking the availability of data for container i on node n.
γ_n	Operating cost of node n enabling serverless computing.
p	Cost of operating a container with one unit of resource consumption in the cloud.
c_i	Resource consumption of container i.
C_n	Resource capacity of node n.

表 1. 定义符号

当数据从一个节点传输到另一个节点时，就会产生传输延迟。所有容器引入的数据传输总延迟可以总结为：

$$U_1 = \sum_{i \in I} \sum_{n_1 \in N} \sum_{n_2 \in N} w_{i,n_1,n_2} \cdot f_{i,n_1,n_2}.$$

边缘设备具有异构性，不同的边缘设备具有不同的资源类型和数量。我们可以将总的执行延迟定义为：

$$U_2 = \sum_{i \in I} \sum_{n \in N} \alpha_{i,n} \cdot x_{i,n}.$$

对于每个容器所需数据的可获得性，我们考虑该模型中最一般的情况。我们将目标函数中所有集装箱的数据可得性表示为：

$$U_3 = \sum_{n \in N} \sum_{i \in I} \beta_{i,n} \cdot y_{i,n}.$$

运行无服务器服务模型涉及从容器调度和守护进程等多个方面的运行成本。可以认为无服务器边缘计算网络的总运行成本为：

$$U_4 = \sum_{n \in N/1} \gamma_n \cdot z_n + \sum_{i \in I} p \cdot c_i \cdot x_{i,1}.$$

通过定义目标函数和约束条件，我们将数据感知的容器放置和流路由问题 P1 描述为如下形式：

$$\min_{x_{i,n},y_{i,n},z_n,f_{i,n_1,n_2}} U_1 + U_2 + U_3 + U_4$$

针对每个容器请求 i，首先计算其在每个节点 n 上的成本 $\alpha_{i,n}$，该成本反映了将容器 i 放置在节点 n 上的代价，包括数据传输延迟、资源利用率等因素。接下来，根据计算得到的成本 $\alpha_{i,n}$，选择对应的节点 n′，使得容器 i 的放置成本最小。这一步骤可以通过遍历所有节点并计算成本的方式来实现。在选择了节点 n′ 后，需要更新容器 i 的成本 $\alpha'_{i,n}$，并根据新的成本重新调整问题 P1。通过组合 $h_{i,n}$ 和 $\alpha_{i,n}$，消去 U1 和 U3，可将原问题 P1 转化为新的在线问题 P2：

$$\min_{x_{i,n},z_i} \sum_{i\in I}\sum_{n\in N} \alpha'_{i,n}\cdot x_{i,n} + \sum_{n\in N/1}\gamma_n \cdot z_n + \sum_{i\in I} p\cdot c_i \cdot x_{i,1}$$

同时，要满足以下约束条件：

$$\sum_{i\in I} c_i \cdot x_{i,n} \le z_n \cdot C_n, \forall n \in N.$$

$$\sum_{n\in N} x_{i,n} = 1, \forall i \in I.$$

$$x_{i,n}, y_{i,n}, z_n, f_{i,n_1,n_2} \in \{0,1\}, \forall i \in I, n, n_1, n_2 \in N.$$

4 复现

本篇文章没有开源代码，我参照了文章提供的伪代码实现 ODCS 算法。具体算法如 Algorithm 1 所示。

4.1 复现细节

我们需要分析和设计数据集。因为在实现算法之前，我们需要一些数据作为输入，这是最重要的部分。这个算法的输入部分较为复杂，有二维和三维矩阵。输入包括即将到来的容器信息，如 $\alpha_{i,n}$，$\beta_{i,n}, w_{i,n_1,n_2}, c_i$，还有边缘设备-云服务器的网络条件，如 C_n, γ_n, p。同时分析出算法的输出：每个容器决定在哪个结点执行 $x_{i,n}$，在哪个结点取数据 $y_{i,n}$，流量路由情况 f_{i,n_1,n_2} 及开启哪些结点进行无服务器计算 Z_n。在复现代码前，必须要考虑好用什么数据类型存储对应的数据。

接下来根据输入信息 w_{i,n_1,n_2} 构建多张图 $G_i = (N, E)$，为了节省空间，每张图都使用邻接表实现。接下来对具有数据的边缘设备做单源最短路径 Dijkstra 算法，对于每一个边缘设备和云服务器找到拥有数据的结点的距离 $l_{i,\hat{n}_k,n}$。将 $\alpha_{i,n}$ 用 $\alpha'_{i,n} = \alpha_{i,n} + h_{i,n}$ 替换即可消去 U_1 和 U_3 构成问题 $P2$。接下来根据容器信息分别更新 $\delta_{i,1}$ 和 $\delta_{i,n}$，第一种选择是将容器放置在已经初始化为无服务器计算且具有足够剩余资源容量的边缘节点上。第二种选择是初始化一个新的边缘节点，并在其上部署容器。第三种选择是在远程云端启动容器。对所有结点进行升序，在确保不超过

Algorithm 1 The ODCS Algorithm

Input: Related information of the upcoming containers, i.e., $\alpha_{i,n}, \beta_{i,n}, w_{i,n_1,n_2}, c_i$.
The conditions of the edge-cloud network, i.e., C_n, γ_n, p.

Output:*textContainerplacement*$x_{i,n}$, data location choice $y_{i,n}$, flow routing f_{i,n_1,n_2}, and enabled nodes for serverless computing z_n.

for all $i \in I$ do
  Construct a directed graph G_i with edge weight w_{i,n_1,n_2}.
  Suppose all nodes $\hat{n}_k$ with $\beta_{i,\hat{n}_k} = 0$ belong to Set N_i^{data}.
  for all $\hat{n}_k \in N_i^{\text{data}}$ do
    Conduct the single-sourced shortest path algorithm for node $\hat{n}_k$
    and get the shortest path and corresponding length $l_{i,\hat{n}_k,n}$ to each node n in G_i.
  end for
  for all $n \in N$ do
    Find the $\hat{n}_k \in N_i^{\text{data}}$ with the smallest $l_{i,\hat{n}_k,n}$ and assign $h_{i,n} = \min_{\hat{n}_k}\{l_{i,\hat{n}_k,n}\}$.
    Substitute $\alpha_{i,n}$ with $\alpha'_{i,n} = \alpha_{i,n} + h_{i,n}$ and eliminate U_1 and U_3 to formulate $P2$.
  end for
Suppose the increment to the objective function of $P2$ by placing container i on node n is $\delta_{i,n}$.
for all $n \in N$ do
  if $n = 1$ then
    $\delta_{i,1} = \alpha'_{i,1} + c_i \cdot p$
  else if $z_n = 0$ then
    $\delta_{i,n} = \alpha'_{i,n} + \gamma_n$
  else
    $\delta_{i,n} = \alpha'_{i,n}$
  end if
  Sort nodes in the increasing order of $\delta_{i,n}$ as Set N_i^{sort}.
  Denote the current workload of node n is L_n.
  for all $n \in N_i^{\text{sort}}$ do
    if $L_n + c_i \leq C_n$ then
      $x_{i,n} = 1, L_n = L_n + c_i$.
      $y_{i,\hat{n}_{k'}} = 1,$ where $l_{i,\hat{n}_{k'},n} = h_{i,n}$.
      if $z_n = 0$ then
        $z_n = 1$.
      end if
      Assign f_{i,n_1,n_2} to 1 along the shortest path from $\hat{n}_{k'}$ to n in G_i.
      Break.
      end if
    end for
  end for
end for

结点容量 C_n 的情况下选取花销 $\delta_{i,n}$ 最小的节点，同时更新 Z_n,f_{i,n_1,n_2}。最终输出结果 $x_{i,n}$，$y_{i,n}$，f_{i,n_1,n_2} 和 Z_n。

总的来说，每个容器运行 ODCS 算法的复杂度都是 $\mathcal{O}((|N|+|E|)|N|\log(|N|))$。则对于所有容器的总复杂度为 $\mathcal{O}((|N|+|E|)|N||I|\log(|N|))$。

4.2 复现结果分析

由于输入数据的结构复杂、具有大量变化的参数，以及对生成符合要求的测试数据的挑战性，我们采用了一系列简单而清晰的测试数据，如图 2 所示。这些测试数据的设计旨在使其易于理解和验证，以确保我们的实现在各种情境下能够正确运行。

Test Case	scaleI	scaleN	Ain	Bin	Win1n2	Ci	Cn	Yn	p
1	3	4	$\begin{bmatrix}0&0&0&0\\1&1&1&0\\1&1&0&1\end{bmatrix}$	$\begin{bmatrix}0&0&0&0\\0&0&1&1\\0&1&0&3\\0&1&3&0\end{bmatrix}$	$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}$, $\begin{bmatrix}0&0&0&0\\0&0&1&1\\0&1&0&3\\0&1&3&0\end{bmatrix}$, $\begin{bmatrix}0&0&0&0\\0&0&10&3\\0&10&0&3\\0&3&3&0\end{bmatrix}$	$[0\ \ 0.2\ \ 0.3]$	$[0\ \ 6\ \ 10\ \ 8]$	$[0\ \ 2\ \ 4\ \ 7]$	0.3
2	10	4	$\begin{bmatrix}0&0&0&0\\1&0&1&0\\1&1&0&1\\1&0&0&1\\1&1&1&0\\1&1&1&0\\1&0&0&1\\1&0&1&0\\1&0&0&0\\1&1&0&1\end{bmatrix}$	$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}$	$\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}$, $\begin{bmatrix}0&0&0&0\\0&0&1&1\\0&1&0&3\\0&1&3&0\end{bmatrix}$, ...	$[0\ \ 1.3\ \ 0.5\ \ 1.1\ \ 0.7\ \ 1.4\ \ 0.4\ \ 1.0\ \ 2.2\ \ 0.9]$	$[0\ \ 6\ \ 10\ \ 8]$	$[0\ \ 2\ \ 4\ \ 7]$	0.3

图 2. 测试数据

在测试数据集中，我们考虑了多样化的情况，涵盖了输入数据中的各种变化。这包括容器信息、网络条件、以及其他决策参数的不同取值。尽管这些测试数据相对简化，但我们通过手动计算的方式验证了它们的期望输出结果。在确保手动计算的结果准确无误的基础上，我们进一步运行程序进行验证。经过验证，程序生成的结果与手动计算的结果一致。程序在面对不同情况时，输出结果均符合预期，这证明了算法的正确性和稳健性。图 3 展示了程序的部分结果，进一步说明了在不同条件下程序的输出质量。

```
/usr/bin/python3.10 /home/yangxiaofan/PycharmProjects/pythonProject/Online Container Scheduling/online_container.py
container 1 :
在哪个结点执行: [0. 0. 0. 1.]
在哪个结点取数: [0. 1. 0. 0.]
container 2 :
在哪个结点执行: [0. 0. 1. 0.]
在哪个结点取数: [0. 0. 1. 0.]
container 3 :
在哪个结点执行: [0. 0. 1. 0.]
在哪个结点取数: [0. 1. 0. 0.]
```

图 3. 程序部分结果

5 总结与展望

本文展示了一种数据感知的容器调度算法，该算法可以保证每个到达的容器请求都达到了局部最优。通过对论文所展示的 ODCS 伪代码，完成对论文的复现。

虽然论文复现成功了，仍有机会进一步拓展和深化研究，可以考虑以下几个方向的工作。例如，扩展测试数据集生成更多的大规模测试数据集，尤其是包含各种边缘情况的数据集，有助于更全面地评估算法的性能；进行多因素影响分析，进一步研究容器数量、网络状况、边缘设备数量等因素对 ODCS 算法性能的影响；将 ODCS 算法与其他相关算法（如 DAP、ND、FPTR 等）进行比较，分析它们在不同条件下的效率差异；进一步细化性能度量和评估指标，确保对算法效果的客观评估。可以考虑加入更多的指标，如资源利用率、执行时间、能效等，以提供更全面的性能评价。

参考文献

[1] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Michel Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: function composition for serverless computing. *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017.

[2] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph Gonzalez, Raluca A. Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *ArXiv*, abs/1902.03383, 2019.

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2658–2659, Los Alamitos, CA, USA, jun 2017. IEEE Computer Society.

[4] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[5] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21:64–71, 2017.

[6] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10, 2019.

[7] Li Pan, Lin Wang, Shutong Chen, and F. Liu. Retention-aware container caching for serverless edge computing. *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1069–1078, 2022.

[8] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Gener. Comput. Syst.*, 114:259–271, 2021.

[9] Yu Liu, Xiaojun Shang, and Yuanyuan Yang. Joint sfc deployment and resource management in heterogeneous edge for latency minimization. *IEEE Transactions on Parallel and Distributed Systems*, 32:2131–2143, 2021.

[10] Yingling Mao, Xiaojun Shang, and Yuanyuan Yang. Provably efficient algorithms for traffic-sensitive sfc placement and flow routing. *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 950–959, 2022.

[11] Xiaojun Shang, Yaodong Huang, Zhenhua Liu, and Yuanyuan Yang. Reducing the service function chain backup cost over the edge and cloud by a self-adapting scheme. *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 2096–2105, 2020.