计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220729
60(5): 1002−1020, 2023

Mort: 面向实时数据分发和传输优化的依赖性任务卸载框架

殷昱煜　苟红深　李尤慧子　黄彬彬　万　健
（杭州电子科技大学计算机学院　杭州　310018）
（yinyuyu@hdu.edu.cn）

Mort: A Dependent Task Offloading Framework Towards Real-Time Data Distribution and Transmission Optimization

Yin Yuyu, Gou Hongshen, Li Youhuizi, Huang Binbin, and Wan Jian
（*College of Computer Science & Technology*, *Hangzhou Dianzi University*, *Hangzhou* 310018）

Abstract　In edge collaborative computing, a single device can no longer support more and more complicated applications and services. Their tasks are offloaded to the adjacent edge server with rich computing and storage resources to match the various high calculation capabilities and low latency requirements. At the same time, the publish-subscribe system is commonly applied from the communication perspective to build a unified transmission protocol to protect data privacy. In the publish-subscribe system, tasks often execute periodicity, depend on each other and the data is distributed to different clients in high-frequency. However, the traditional task offloading algorithms mainly aim at the single data transmission and single task execution scenario, which cannot effectively handle the offloading characteristics in a publish-subscribe system. Therefore, we propose Mort, a task offloading and management framework. It supports task decomposition using the static program analysis technique, and the task dependencies are extracted and the parallelism is increased. Nonlinear integer programming-based modeling and group-based resource fusion-based offloading algorithms are applied to optimize network data transmission. The coordination process-based scheduling model is used to schedule the offloading tasks with less overhead. Our comprehensive experiments show that Mort's data transmission optimization can reach 80%-90% of the optimal solution with a low overhead of only 2%.

Key words　edge computing; publish-subscribe system; dependent task offloading; combination optimization; time-sensitive

摘　要　在边缘协同计算中，单一设备已无法满足大量复杂任务对系统高计算能力和低时延的需求，通常需要将任务卸载到邻近具有丰富计算存储资源的边缘服务器上，同时，通过发布订阅模式构建统一的通信协议，支撑任务对数据共享和隐私保护的需求. 在发布订阅系统中，任务往往具有依赖性、执行周期性和高频率数据分发等特性，而传统的任务卸载算法主要针对数据单次传输和任务单次执行的场景，无法有效应对发布订阅系统中的任务卸载问题. 因此，设计一个任务卸载及管理框架 Mort. 该框架使用基于静态代码分析的任务分解方法解构任务之间的依赖性，支撑任务并行；采用基于非线性整数规划建模和基于分组及资源融合的卸载算法，优化网络数据传输；采用基于协程的调度模型调度卸载后的任务，减少任

收稿日期：2022-08-20；**修回日期**：2023-03-30
基金项目：浙江省自然科学基金项目(LY22F020018)；国家自然科学基金项目 (62272140)
This work was supported by Zhejiang Provincial Natural Science Foundation of China (LY22F020018) and the National Natural Science Foundation of China (62272140).
通信作者：李尤慧子(huizi@hdu. edu. cn)

务调度开销. 大量的仿真实验和真实场景实验表明，Mort 的数据传输优化能达到最优解的 80%~90%，且引入的系统开销仅约为 2%.

关键词 边缘计算；发布订阅系统；依赖性任务卸载；组合优化；时间敏感

中图法分类号 TP393

随着通信技术（如 5G）和人工智能等关键技术的发展，边缘计算（edge computing, EC）和物联网（Internet of things, IOT）等大型分布式信息系统的格局和规模发生了显著的变化. 根据国际数据中心的《中国物联网连接规模预测，2020—2025》和《2021 年 11 月爱立信移动市场报告》显示，截止到 2025 年中国物联网总连接量将达到 102.7 亿，将占到亚太地区（除日本外）总连接量的 84% [1]. 同时，这些接入设备产生的数据也在急剧增加. 预测显示，截止到 2027 年末移动网络数据总流量可能达到 370 EB[2]. 这些设备常分布于不同的地理位置，在多个方面存在差异（异构）：1）底层芯片，如 EMPC（economic embedded multi-chip package），MCU（microcontroller unit），FPGA（field programmable gate array），DSP（digital signal processor）；2）计算能力；3）上层操作系统，如 HelloX, Android IOT/Brillo, Windows 10 IOT Core, Ubuntu Core 16, FreeRTOS, TinyOS；4）网络通信协议，如 MQTT（message queuing telemetry transport），NB-IOT（narrow band IOT），ZigBee，CoAP（constrained application protocol），5G. 随着边缘业务日趋繁杂，其应用程序也变得复杂，如虚拟/增强现实[3]、智慧交通[4] 等. 它们的计算需求较大、延迟敏感，普通边缘设备不能提供其计算需要的资源，而云计算时延太高[5]. 因此，EC 和 IOT 面临 2 个问题亟需解决：1）设备软硬件异构性带来的通信难问题，繁杂的协议使得数据共享和隐私保护等操作变得困难；2）低成本和对丰富的计算资源需求之间的平衡问题. 低廉的设备成本促进了 EC 和 IOT 的发展，但却难以完全满足现阶段的计算需求.

通信难问题主要是因为统一规范的缺失，发布订阅系统（publish/subscribe system, PSS）[6] 常常作为一种解决方案. PSS 是一种数据共享的模式和规范，它并不限制底层采用何种方式进行通信. 在一般的实现中 PSS 采用 TCP/IP 方式，也可以使用如 NB-IOT, ZigBee, WIFI 等常见方式. PSS 一般由发布者、订阅者和事件服务组成，采用以数据为中心的数据分发模式，可以实现大规模分布式系统中各实体间数据的分发. 在这种模式下，发布者将数据封装成事件，订阅者订阅自己感兴趣的事件，事件服务则负责将各事件分发给订阅者. 在分发过程中，不用关心具体的实体（异构性），发布者和订阅者无需相互感知和连接，其交互完全匿名，具有灵活性、去耦合、异步通信等重要特点. PSS 在大型分布式消息系统中越来越受到重视，如基于千兆网的自动驾驶应用[7]、物联网数据共享应用[8]、电力物联网的应用[9] 和机器人通信[10].

对于计算资源不足的问题，学界常常采用任务卸载. 任务卸载是指将边缘终端的计算任务卸载到边缘服务器等资源丰富的计算节点上，从而同时满足计算能力和时延要求. 任务卸载可以分为 2 个子领域[11]：独立性任务卸载（independent task offloading）和依赖性任务卸（dependent task offloading）. 独立性任务是指任务之间没有数据依赖关系，彼此独立. 独立性任务卸载研究的问题往往是在一系列约束条件下，如计算能力、截至时间、电池容量等，如何将持续到达（或已经到达）的多个任务合理地卸载到附近资源丰富的计算节点上，以获得最大的收益（或减少损失），如文献 [12] 的执行时间和电池优化、文献 [13] 的计算资源优化等. 依赖性任务是指任务之间存在输入输出的依赖关系. 同独立性任务卸载相比，依赖性任务卸载问题还需要考虑任务之间的执行顺序. 对依赖性任务卸载的研究主要集中在仿生[14]、深度强化学习[15-17] 等智能算法和数学优化[18]、启发式[19-20] 等传统优化方法上.

在 EC 和 IOT 中，异构性带来的通信难等问题可以通过借助发布订阅模式提供统一的数据共享规范来解决. 计算资源的匮乏可以采用任务卸载，由边缘计算节点分担计算需求，同时降低延迟. 然而，当发布订阅系统和任务卸载结合时会产生新的问题，如现有的任务卸载算法不能很好地应用在发布订阅系统中. 发布订阅模式下的边缘业务/应用具有 3 个特点：1）由多个子任务构成，任务之间有依赖关系；2）任务具有周期性，在一段时间内重复执行单一功能，没有明确的完成时间；3）任务执行中，往往存在大量的数据分发. 因为具有周期性、高频数据分发等特点，大量现有基于完成时间的任务卸载算法无法很好地应对这类场景. 可以重复执行这些基于完成时间的任务卸载算法来模拟周期性，但会不可避免地引入

调度等额外开销. 在此类应用中, 数据传输带来的开销变得不可忽视, 因此本文聚焦于 PSS 下基于数据传输优化的依赖性任务卸载问题, 并设计实现了框架 Mort 来解决该问题.

针对依赖性和执行周期性的特点, Mort 实现了基于静态代码分析的任务分解和基于协程(可以理解为一种能够保存 CPU 上下文的函数)的调度模型. 底层操作系统更强调通用性, 它的调度策略更多的是考虑进程(线程)间的公平性, 而无关上层业务. 这些子任务间存在依赖关系, 执行时有先后次序, 若次序靠后的任务先被调度执行, 就会浪费此次调度机会和 CPU 时间. 因此需要一种方式控制这些任务按照依赖次序调度以减少执行周期时间. 本文中的任务以协程的方式实现, 从而可以在应用层进行调度控制. 虽然通过同步或设置进程优先级似乎也能达到同样的效果, 但同步操作依旧不能避免操作系统的错误调度, 而优先级设置则更多是一种对操作系统的建议, 不会得到明确的保证.

针对高频率数据分发特点, 为减少网络带宽资源的消耗和网络拥堵的可能, Mort 实现了 2 个任务卸载算法: 基于分组的任务卸载(grouping-based task offloading, GBTO)与基于分组和资源融合的任务卸载(grouping and fusion-based task offloading, GFBTO). GBTO 采用分组策略, 在邻近的计算节点中选择满足任务资源需求的节点, 并尽可能将任务卸载到其所需数据的源头. 考虑到满足依赖关系的多个任务卸载到同一个计算节点后可以共享计算资源, 由此本文在 GBTO 的基础上提出基于路径归并的计算资源优化算法 GFBTO, 进一步减少带宽资源的消耗. 然而, 应用 GFBTO 会使得计算节点的负载变高, 在某些场景中可能会降低系统响应性. 因此 GBTO 适用于 CPU 密集型任务组, 而 GFBTO 更加适合 I/O 密集型任务组. GFBTO 算法能够将更多的任务卸载到同一个计算节点, 这些任务可以在应用层得到调度, 使得其执行周期变短, 也适合于时间敏感型任务. 无论是 GBTO 还是 GFBTO 都更倾向于把任务卸载到其输入源处, 尽可能地让数据在本地处理(哪里产生哪里处理), 尽可能地减少数据在网络中传输, 同时也减少了由于数据传输带来的网络延迟, 任务能更加及时地计算出结果. 为了更好地使用这 2 种算法, Mort 会根据各计算节点 CPU、I/O 利用率和任务组的 I/O 密集型任务的占比来选择合适的卸载算法. 本文的主要贡献总结为 4 个方面:

1)设计实现了任务卸载及管理框架 Mort. 针对 PSS 中任务的依赖性、周期执行性和高频率数据分发特性, Mort 采用了任务分解、任务卸载和协程调度 3 个步骤, 增加了任务并行度、减少了网络数据传输量和减少了任务调度开销.

2)将面向发布订阅系统的任务卸载数据传输优化问题建模为非线性整数规划问题, 并设计了一种基于数据依赖分组策略的优化算法 GBTO 来解决它.

3)进一步挖掘任务间依赖特性, 在 GBTO 基础上设计了一种基于路径归并算法 GFBTO. 该算法使得卸载到同一计算节点的属于同一任务组的任务间能共享计算资源. 与 GBTO 相比, GFBTO 使得同一个计算节点上卸载的任务数量增加, 提高了计算节点的负载, 但同时也增加了任务并行度和减少了操作系统的调度切换开销. 因此 GBTO 更适合 CPU 密集型任务, 而 GFBTO 更适合 I/O 密集型和时间敏感型任务.

4)针对 GBTO 和 GFBTO, 设计了仿真和真实场景实验. 在仿真实验中, GBTO 和 GFBTO 的数据传输分别能达到 OPT 的 80% 和 90% 左右, 且 GBTO 比 Greedy 高出约 50%. 在真实场景中, 设计了一个分布式的文本统计作业, 分别应用 GBTO, GFBTO, Greedy 算法卸载, 其结果与仿真实验一致, 并且, Mort 本身引入的开销仅约为 2%.

1 任务卸载相关工作

1.1 独立性任务卸载

独立性任务是指卸载的任务之间没有任何关系, 相互独立. 数十年来, 独立性任务卸载领域已有大量成果, 它们可以被分为 2 类: 离线算法[12-13,21-25]和在线算法[11, 26-30]. 离线算法需要知道所有任务的信息, 然后统一卸载任务最大化目标. 文献 [21, 23−24] 主要关注如何最小化卸载时服务的响应时间. 文献 [21] 注意到服务响应时间可能受设备之间连接性的影响, 为协调分配, 设计了一个分布式卸载算法; 文献 [23] 提出一个 2 层的缓存迭代更新算法来解决该问题, 而且还能有效减少卸载过程中的数据传输; 文献 [24] 则是任务卸载和服务缓存在线联合优化机制, 将任务卸载和服务缓存联合优化问题解耦为任务卸载和服务缓存 2 个子问题. 在线算法不需要提前知道任务的信息, 它们可以在任意时刻到来. 文献 [11] 从最大化服务提供方收益的角度考虑, 利用原始对偶方法设计了一个在线近似算法, 并证明了其近似比为 $2(1+\varepsilon)\ln(1+d)$. 在动态的移动边缘计算环境中,

文献 [30] 考虑 IOT 设备的计算任务卸载问题，并基于强化学习设计了一个任务卸载的框架.

文献 [11−13, 21−30] 所述工作主要面向独立性任务，它们假设任务以时间次序到达（在线）或已经到达（离线），其目标是最小化任务完成时间或最大化资源收益等. 而在具有依赖性、执行周期性和高频率数据分发等特点的任务卸载问题中，这些算法的作用并不明显.

1.2 依赖性任务卸载

依赖性任务是指待卸载的任务之间存在输入的依赖关系[31]. 依赖性关系卸载问题是典型的 DAG 调度问题，属于 NP-Hard，相关工作较少. 一些工作借助智能算法尝试解决该问题[14-16]. 面对复杂的依赖性组合优化问题，文献 [14] 采用了基于队列优化的粒子群算法来减少整个任务的完成时间和执行开销. 与文献 [14] 不同，文献 [15] 提出了 PTRE（priority-topology-relative-entropy）算法，将任务 DAG 转化为任务向量序列，从而采用基于深度强化学习的图映射框架来达到目标. 文献 [16] 将资源受限的边缘计算环境下的多作业卸载问题建模为马尔可夫模型，并利用深度强化学习算法减少其传输和计算开销. 智能算法在很多 NP-Hard 问题上表现优秀，然而其耗时较长且没有理论保证其性能. 更多的工作尝试利用数学优化、改进的贪心策略或者精巧的启发函数来近似求解. 在最小化任务完成时间方面，文献 [19] 考虑一个应用由多个任务构成，结合服务端的函数配置和任务卸载，设计了一个近似算法，但它却忽视了边缘服务器的计算能力. 借助 DAG 中的关键路径思想，文献 [32] 提出了 HEFT 启发式算法以减少任务的最早完成时间. 注意到在卸载过程中，边缘服务器和移动设备侧重的点不同，文献 [33] 提出一个启发式算法，尽可能地提高服务器的资源利用率，减少移动设备的能量消耗. 文献 [34] 认为简单地将任务分配到远端云上，其无线通信开销可能比其收益更大，因此设计算法 HTA2，以减少异构移动系统中的能源开销. 另外一些工作[35-37] 则综合考虑任务的完成时间和能量消耗. 文献 [35] 提出 MAUI，即提供细粒度的应用代码到远端服务器的部署，能够在运行时决策将其代码卸载到何处，并最大化节省能量和减少时间消耗. 文献 [36] 从软件提供方角度出发，考虑最小化远端计算开销和移动设备能量消耗，并控制服务响应在一定合理的延迟内，设计了一个多项式时间算法 DTP. 文献 [37] 的目标同样是节省能量开销、计算开销和减少延迟，它设计了一个 3 阶段算法 SDR-AO-ST（semi-definite relaxaion, alternating optimization, and sequential tuning），能够有效减少系统开销.

文献 [14−16, 19, 32−37] 所述的工作都假设任务有确切的执行时间，数据传输是单次的. 然而，发布订阅系统中的任务不仅具有依赖关系，还具有周期性，没有明确的结束时间，执行中往往存在大量数据分发. 针对周期性和大量数据分发的优化不仅可以优化带宽资源（减少数据在网络中的传输），还能降低等待网络数据带来的高延迟（通过将任务卸载到数据源处），然而上述工作没能考虑这些问题. 因此本文基于 PSS 设计了面向数据传输优化的卸载算法.

2 Mort 架构

为高效地完成业务解析、任务卸载、任务管理以及数据分发等操作，本文设计了 Mort，图 1 是它的架构. 本节介绍其中主要的组件和系统工作流程.

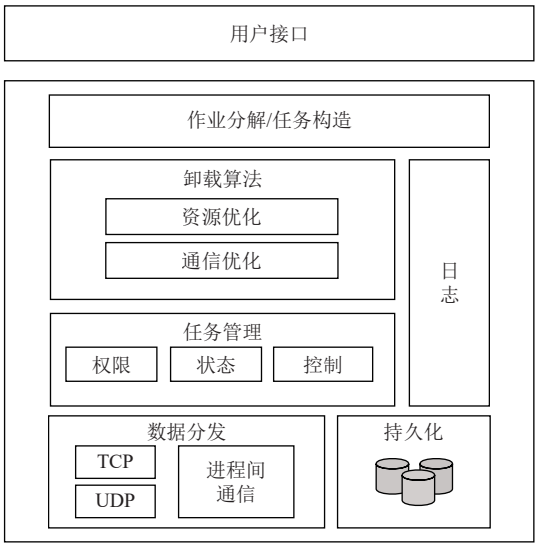


Fig. 1 Mort architecture

图 1 Mort 架构

2.1 作业分解/任务构造组件

用户通过用户接口编写代码和作业描述文件. 作业分解组件借助这些信息构建有向循环图（directed acyclic graph, DAG）任务组实例，如图 2 所示，其中的 d_{ij}, rt_j 分别表示数据依赖和计算资源需求. 某些任务除了依赖本组其他任务外还可能需要外部的数据，因此在构建任务组时会创建虚拟任务. 该组件构造的 DAG 任务组实例会提供给卸载算法组件.

2.2 卸载算法组件

卸载算法组件会结合邻近可达计算节点的状态

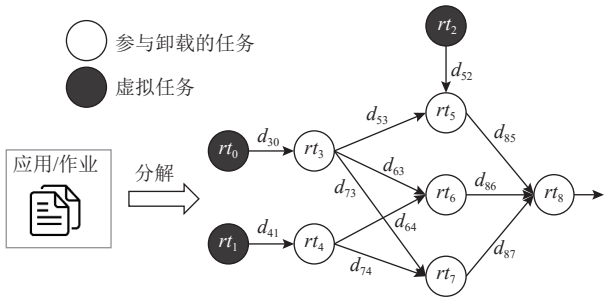


Fig. 2 Job decomposition

图 2 作业分解

信息(由任务管理组件提供)和任务组实例,执行 GBTO 或者 GFBTO 算法,生成详细的任务卸载计划. 随后管理组件依照该计划将任务部署到具体的计算节点上.

2.3 任务管理组件

任务管理组件会维护网络中所有可达计算节点的状态,包括计算节点的资源信息、其上运行任务的权限和状态等信息,为实际的任务卸载提供数据支撑. 同时,通过该组件还可以管理各任务的运行状态.

2.4 数据分发组件

该组件基于发布订阅模型实现,为其他组件提供数据服务. 它主要提供 2 个功能: 1)同步网络中边缘计算节点的状态信息; 2)完成数据的分发.

2.5 持久化组件

持久化组件主要是将设备实时产生的数据持久化. 数据持久化能为历史数据服务提供支持,而且当网络不稳定时还能在一定程度上提高系统的服务质量(quality of service, QoS).

2.6 样例分析

本节以一个例子介绍 Mort 在实际场景中是如何实施的,如图 3 所示. 考虑这样一个简单例子,城市规划中,红绿灯时长分配对路段交通拥堵有很大影响,极端情况下可能在几个路段间引起连锁反应. 在智慧城市中,可以根据路口人流量、车流量、天气等情况动态改变时长. 对于这类动态检测及控制应用,卸载时需要考虑到数据传输的开销、计算节点的负载等以尽量减少单次计算的时间,因此选择合适的计算节点尤为重要.

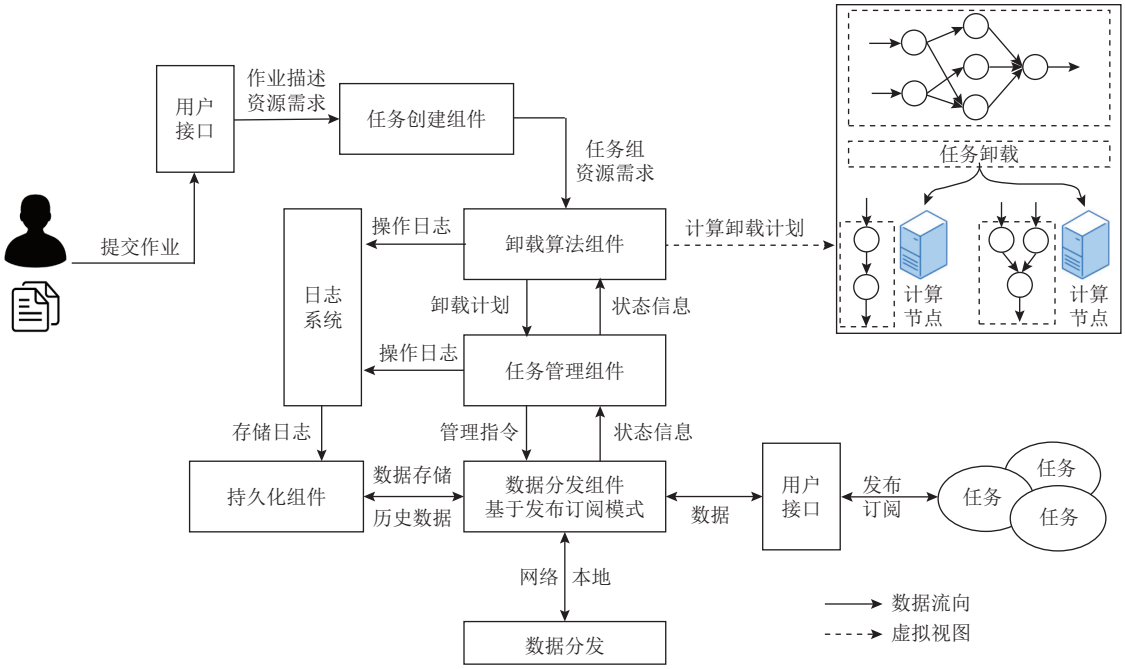


Fig. 3 System workflow

图 3 系统工作流

智能红绿灯场景中,可以将作业构造成 4 个子任务: 人流量检测任务、车流量检测任务、天气检测任务和决策任务. 城市管理人员将 Mort 部署在城市的每一个计算节点上,他们可以在任意一个节点提交作业描述文件. 任务构造组件读取描述文件后将构造这 4 个子任务,并将它们传递给任务管理组件. 任务管理组件会结合任务类型和邻近节点状态选择合适的卸载算法,将任务组和相关信息传递给卸载算法组件制定卸载计划. 卸载算法组件执行相关卸载算法后将卸载计划返回给任务管理组件,任务管理组件再通过数据分发组件将任务分发到各个计算节点执行. 任务执行过程中需要订阅的数据和发布的

数据由数据分发组件在各个节点间传输. 任务执行过程中产生的数据和操作记录等信息可能会记录在持久化组件中以供后续分析使用.

数据传输以发布订阅方式进行, 且只会发生在计算节点之间. 任务卸载到计算节点后, 在其生命周期内按照“订阅到数据—计算—发布数据”这样的方式循环执行. 在任务组构造过程中会为用户端构造一个任务用来订阅计算结果.

3 Mort 实现

本节将详细介绍 Mort 主要组成部分的实现.

3.1 作业分解/任务构造组件

作业分解/任务构造组件通过源代码静态分析得出任务间的依赖关系(通过订阅发布的数据类型和一些其他的控制流接口), 再结合作业描述文件进一步构造任务组实例.

静态代码分析通常有控制流分析[38]和数据流分析[39]两种, 本文采用数据流分析方式. 通过分析追踪各份源代码中函数调用情况和参数传递信息来建立各任务之间的依赖关系. 这里结合一个例子阐述其实现方式. 假设有一项作业由 *TaskA*, *TaskB*, *TaskC* 这 3 个子任务构成. 如图 4 左边, 任务 *TaskA* 和 *TaskB* 分别订阅 $Topic\_V_1$(由任务 $TaskV_1$ 发布)和 $Topic\_V_2$(由任务 $TaskV_2$ 发布)数据类型, 同时发布 *Topic_A* 和 *Topic_B* 数据类型. 任务 *TaskC* 同时订阅 *Topic_A* 和 *Topic_B* 数据类型. *TaskA*, *TaskB*, *TaskC* 这 3 个任务分别调用函数 *Task*::*RegTask*(*name*, …)注册自身信息, 调用函数 *Task*::*RegSub*(*topic_list*)注册订阅数据类型, 调用函数 *Task*::*RegPub*(*topic_list*)注册发布数据类型. 如图 4 中间, 分析 *A* 代码时, 能够构建 *TaskA* 与 $TaskV_1$ 的数据流关系(通过查询任务管理组件获取 $TaskV_1$ 到 *TaskA* 的数据量和频率); 分析 *B* 代码时, 能够构建 *TaskB* 与 $TaskV_2$ 的数据流关系; 分析 *C* 代码时, 能够部分构建 *TaskA*, *TaskB*, *TaskC* 之间的数据流关系. 如图 4 右边(任务 $TaskV_1$ 和 $TaskV_2$ 不属于本次任务组, 所以用深色标识), 融合 *TaskA*, *TaskB*, *TaskC* 时, 可以完整构建任务 DAG 数据流, 此时再结合作业描述文件获取各个任务的资源需求等信息, 任务组 DAG 就能够构建完成. 在融合时, 各任务只能依赖已卸载或存在本任务组中的任务, 不能循环依赖. 当这些步骤执行后, 任务组构建完成.

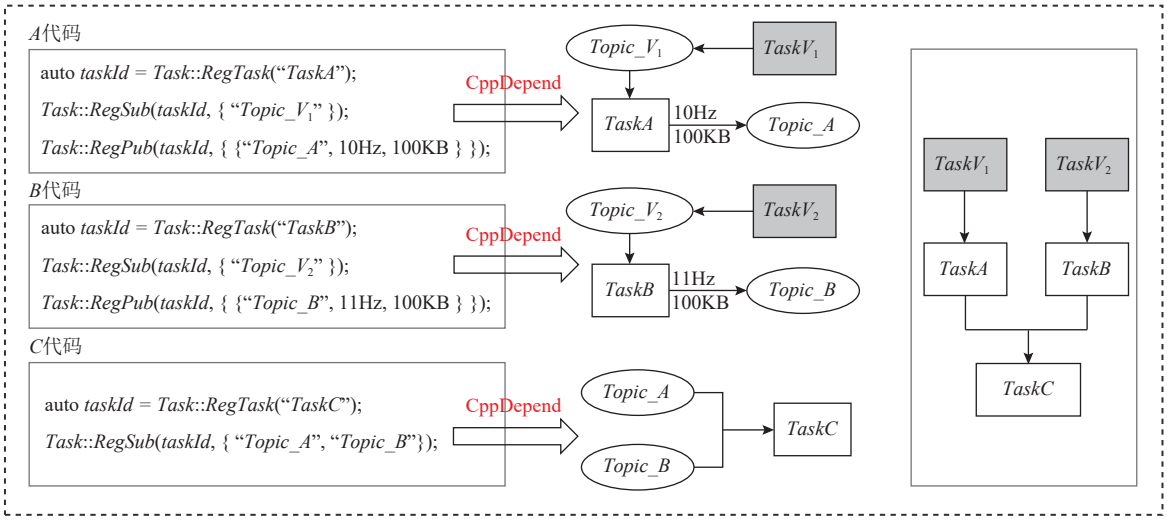


Fig. 4 Static program analysis to build task group

图 4 静态代码分析构造任务组

3.2 任务管理组件实现

任务管理组件主要负责响应系统环境的变化、接收用户命令、执行卸载算法组件给出的卸载计划以及调度卸载后被抽象成协程的任务, 如图 5 所示.

在 PSS 中, 数据的发布和订阅需要授权[40]. Mort 的权限验证模块使用了基于属性的访问控制(attributed based access control, ABAC)技术, 并增加了地理位置、资源利用率等动态属性. 当系统环境等上下文变化时, 可能会引起权限的变化, 而且用户也可以通过用户接口显式地改变某些属性值或权限. 为进一步对数据传输提供保护, 还可以将数据加密后进行传输, 或为计算节点引入证书等方法.

每一个计算节点上都会部署 Mort. 为了使这些节点上的任务管理组件信息同步, Mort 启动后会在

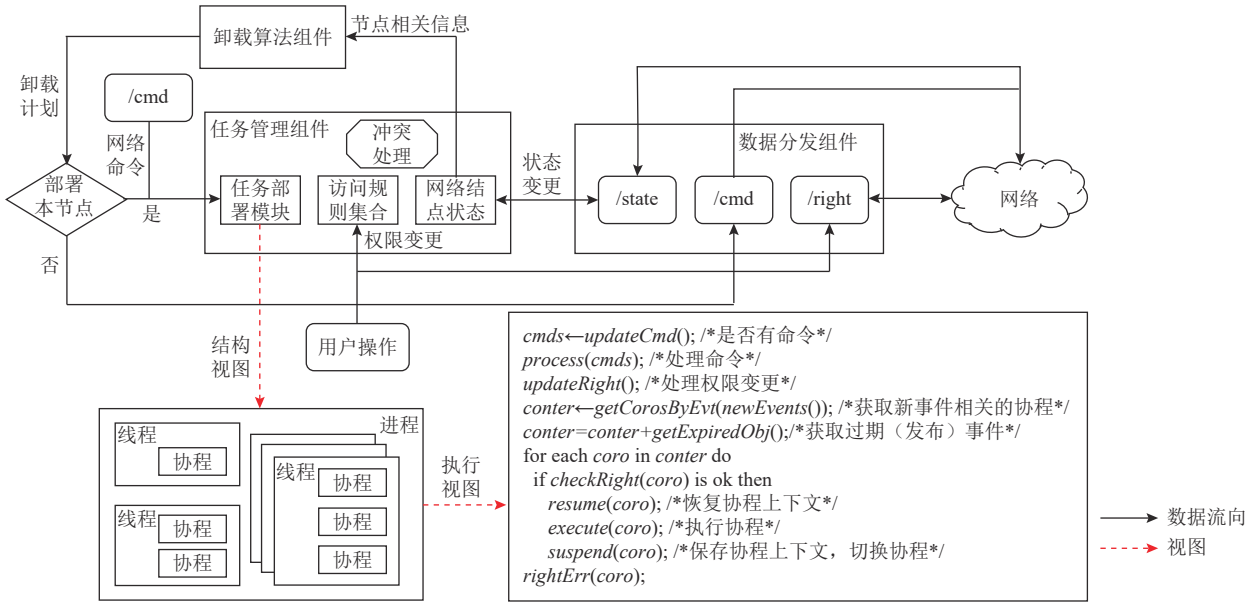


Fig. 5 Task management component logic implementation

图 5 任务管理组件逻辑实现

数据分发组件中维护/right, /state, /cmd 这 3 个数据域，分别用来同步权限、节点/任务状态和来自其他节点的命令等信息. 该组件还包含一个冲突处理模块，主要负责处理同步导致的节点状态的不一致. 不一致主要是因为多个用户同时设置不同的权限或命令而引发的冲突. 对于这 2 种冲突，会根据用户权限(higher-prior win)、操作先后次序(first-commit win)等情况判定和记录日志后续恢复等措施解决.

任务卸载后，从任务管理组件的角度来看(任务视图)，一个进程包含多个线程，一个线程包含多个协程，在线程调度协程时(执行视图)，它首先会处理新到来的命令(函数 *process*)、更新权限(函数 *updateRight*)，然后分发数据. 在分发数据阶段，每个任务都被抽象成了发布者或订阅者，订阅者响应数据到来事件，发布者产生过期事件(按一定频率发布数据). 分发数据主要是处理这 2 种事件和权限鉴定. 其中函数 *resume*、函数 *execute*、函数 *suspend* 操作是协程的调度操作. 函数 *resume* 表示恢复当前协程的 CPU 上下文；函数 *execute* 则是读取或者发布数据；而函数 *suspend* 是保存当前协程的 CPU 上下文，让出 CPU.

3.3 数据分发组件实现

该组件负责接收发布者的数据、将数据分发给订阅者和接收/分发来自任务管理组件的命令等功能. 数据分发组件的实现采用的是 OMG(object management group)发布的 DDS(data distribution service)规范[41]，业界也有很多成熟的实现，如 FastDDS[42]. Mort 除了基于 DDS 实现了分布式网络分发之外，还针对本地数据分发做了优化，如分发到本地的数据不再走网络部分，而采用进程间共享内存或进程内共享(被绑定在一起的属于同一任务组的任务间共享)内存的方式.

3.4 卸载算法组件实现

卸载算法组件负责在构造的任务组上执行卸载算法从而给出具体的卸载计划. 下面阐述其具体实现.

3.4.1 卸载算法组件建模

1)作业及任务模型

如图 2 所示，用户作业由 M 个依赖性子任务组成，记为 $\mathcal{J}=\{t_1, t_2, \cdots, t_M\}$. 每个子任务可以表示为 $t_i=\{\boldsymbol{d}_{i*}, rt_i\}$，其中 $\boldsymbol{d}_{i*}=(d_{ij})$ 是一个向量，表示任务之间的数据依赖：如果任务 t_i 不依赖 t_j(t_j 不需要传输数据给 t_i)，则有 $d_{ij}=0$；反之，任务 t_i 依赖 t_j，则依赖数据量为 d_{ij}(单位：bps). 一般地，这些子任务还可能依赖外部的任务，如图 2 中构造的 3 个虚拟任务 rt_0, rt_1, rt_2. 这 3 个任务不属于该次作业，而是之前已卸载的任务，现存在于某些计算节点上. $\boldsymbol{rt}$ 也是一个向量，$\boldsymbol{rt}=(rt_i)$ $i=1, 2, \cdots, M$. rt_i 表示该任务 i 的资源需求，为简单起见，此处指 CPU 核数. 本文任务模型中，用户作业指移动边缘计算中的业务作业，其包含的任务无明确的执行时间，它们可能会滞留计算节点重复执行. 每个任务不能再次划分，且只能分配到一个计算节点.

2)计算节点及卸载模型

假设当前网络中有 N 个计算节点，记为 $C=\{c_1, c_2, \cdots, c_N\}$，且每一个计算节点当前有 rc_j 个空闲的 CPU 核. 设 A_{ij}=1 表示将任务 i 卸载到计算节点 j，卸载

到每个计算节点的任务不能超过该节点的CPU空闲容量，因此有

$$\forall c_j \in C: \sum_{i=1}^{M} A_{ij} \times rt_i \leqslant rc_j, \tag{1}$$

$$\forall c_j \in C, t_i \in \mathcal{T}: A_{ij} \in \{0, 1\}. \tag{2}$$

假设网络中的计算节点总能容纳所有的任务（不考虑等待的时间），而且每个任务在其生命周期内只能卸载到一个计算节点上，所以对于本次作业的所有任务有

$$c_j \in C, t_i \in \mathcal{T}: \sum_{i=1}^{M} \sum_{j=1}^{N} A_{ij} = M, \tag{3}$$

$$\forall c_j \in C, t_i \in \mathcal{T}: \sum_{j=1}^{N} A_{ij} = 1. \tag{4}$$

需要注意，该模型的任务组中的任务可能依赖之前已经卸载了的外部任务. 这些约束需要结合实际情况考虑. 假设$\mathcal{S}$是在之前就已经被分配任务的集合，且对应的计算节点集合为$\mathcal{P}$，有

$$\forall t_i \in \mathcal{S}, c_j \in \mathcal{P}: A_{ij} = 1. \tag{5}$$

本文中的计算节点包括2个部分：边缘服务器和计算存储资源相对丰富的边缘设备，它们一般负责与一定区域的边缘设备和传感器交互，包括数据共享和命令传送.

3）问题模型

众多边缘设备和传感器在高速、大量地产生原始数据，这些数据稍后在网络中共享、处理. 卸载任务时考虑数据的传输优化可以有效地减少整个网络中数据的传输量. 因此本工作中的卸载算法需要解决的问题是：给定一组依赖性任务$\mathcal{J} = \{t_1, t_2, \cdots, t_M\}$（包括它们的CPU需求和数据依赖）和一组边缘计算节点$C = \{c_1, c_2, \cdots, c_N\}$（包括它们的CPU空闲容量），如何将这些任务卸载到这些计算节点上，使得整个网络中传输的数据最小，即卸载后任务本地处理的数据量最大

$$\max \sum_{i=1}^{M} \sum_{j=i+1}^{M} \left[\sum_{k=1}^{N} A_{ik} \times A_{jk} \right] \times d_{ij}, \tag{6}$$

$$\text{s.t.式}(1) \sim (5). \tag{7}$$

这是一个非线性整数规划模型，可以将整数约束用等价的二次项替换，从而达到松弛变量的效果. 因此，式（2）可以改为

$$\forall c_j \in C, t_i \in \mathcal{J}: A_{ij}^2 - A_{ij} = 0, \tag{8}$$

$$\forall c_j \in C, t_i \in \mathcal{J}: A_{ij} \in [0, 1]. \tag{9}$$

许多工作已经证明，在异构环境中的DAG调度是NP-hard[43]，式（6）（7）描述的模型同样是一个NP-hard问题.

3.4.2 GBTO算法实现

本节提出了一个基于分组的任务卸载算法GBTO近似地解决该问题，并给出复杂度说明. 该算法分为2个部分：预处理和任务卸载.

1）预处理

在预处理中，借鉴了字符串匹配算法KMP的思想，尽量挖掘对象自身的模式来加速后面的卸载过程，图6和算法1描述了其细节.

图6中，圆形表示任务，圆形中间的rt_i表示该任务的CPU需求，圆形上边的数字表示任务编号，而箭头上边的d_{ij}表示任务之间数据传输量. 在预处理时，首先求得任务组的拓扑序列，该操作由算法1中的函数*topologicalSequence*完成. 然后从后往前遍历拓扑序列，根据其数据依赖的大小主动选择一个它的直接前驱任务并为一组（如图6中的虚线有向箭头和算法1中的行②~⑮，其中函数*merge*表示集合的并集

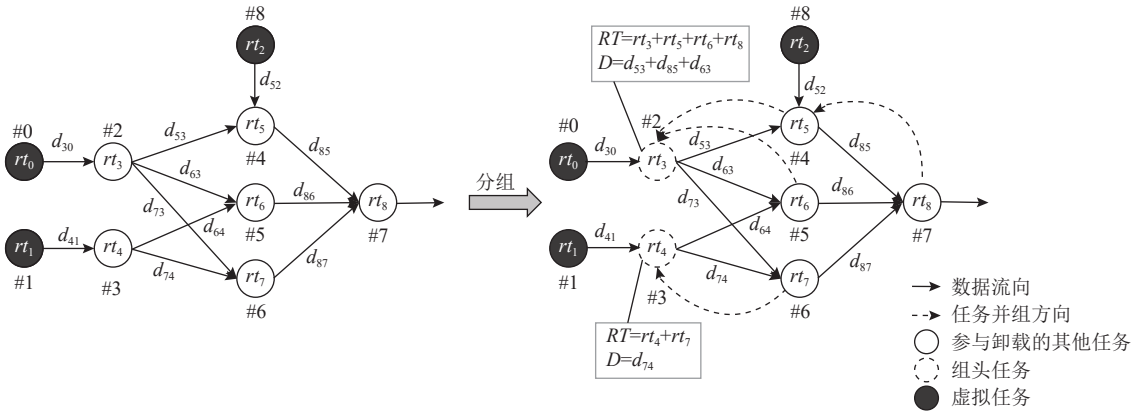


Fig. 6 Illustration of algorithm 1

图6 算法1的图解

操作). 待所有任务选择完之后，整个作业就被分为了若干组(如图 6 中被分成了 3 组: [0, 2, 4, 5, 7], [1, 3, 6], [8])，这些组稍后会作为一个整体考虑. 每一个组都有一个领头任务(不算虚拟任务，如图 6 中的虚边框任务 2 和任务 3)，它记录着该组的所有数据依赖量D和资源需求量RT. 虚拟任务不参与分组.

算法 1. GBTO 预处理.

输入: 任务集合$\mathcal{J} \leftarrow \{t_1, t_2, \cdots, t_M\}$、任务组集合$G \leftarrow \{\{t_1\}, \{t_2\}, \cdots, \{t_M\}\}$、任务拓扑序列$Q \leftarrow \varnothing$、任务组依赖数据量矩阵$\boldsymbol{d}$;

输出: 任务组集合G.

/*计算任务集合拓扑序列*/

① $Q \leftarrow topologicalSequence(\mathcal{J})$;

② while $Q \neq \varnothing$ do

③ $fathers_i \leftarrow \varnothing$;

/*取任务集合拓扑序列的第 1 个元素，并从队列中删除该元素*/

④ $t_i \leftarrow Q.top()$ and $Q.pop()$;

⑤ for t_j in $\mathcal{J}$ and $t_j \neq t_i$ do

⑥ if $d_{ji} \neq 0$ do /*如果t_j向t_i传输数据*/

⑦ $fathers_i \leftarrow fathers_i \cup t_j$; /*将$t_j$放到$t_i$前驱(父)任务集合中*/

⑧ end if

⑨ end for

⑩ if $fathers_i = \varnothing$ do

⑪ 继续遍历Q的下一个元素;

⑫ end if

/*从t_i前驱任务集合中选出与t_i有最大数据传输的前驱任务，取其序号*/

⑬ $j \leftarrow \max\{d_{ji} | t_j \in fathers_i\}$;

⑭ $G_j \leftarrow merge(G_i, G_j)$;

⑮ end while

⑯ return G.

2)任务卸载

预处理过程构造了若干个初始任务组，该阶段将以它们为单位进行 2 步骤卸载. 图 7 和算法 2 详细描述了此过程.

① 预卸载(如图 7 的左半部分，算法 2 行②~⑭). 每一个组的领头任务尝试卸载本组任务到各个计算节点c_j，并记录其收益$Profit_j$，即该组任务卸载到计算节点c_j上时能保留的依赖数据总量. 卸载时可能会出现$RT > rc_j$的情况，即节点不能容纳该组的所有任务. 此时领头任务将依据组内拓扑序列从后向前依次模拟移除任务，直到剩下的任务可以被该节点容纳，同时在可被卸载的节点中记录能取得的最大收益，即$Profit_{\max}$.

② 正式卸载(图 7 的右半部分，算法 2 行⑮~㉚). 待所有组预卸载之后，从中选取收益最大的组进行正式卸载(算法 2 行⑮，图 7 中则是组 1 在$profit_1$取得最大). 被选中的组在卸载之前先移除前一步骤中模拟移除的任务. 这些被移除的任务会根据实际情况或加入现存的组或构造新的任务组. 具体地，对于每一个被移除的任务，在还未卸载的前驱中选择依赖数据量最大的前驱，然后并入该前驱所属的组. 如果没有这样的前驱存在，则该任务独自构成一新组(算法 2 行⑯~⑳，而图 7 中则是组 1 移除了 1 个任务，该任务被并入到了组 2). 正式卸载某一组后需修改计算节点容量，以及计算节点访问标志，后续剩余组更新时，只需要重新计算发生变化的节点即可(算法 2 行㉕，及图 7 中组 2 需要重新计算其$Profit_1$). 最后

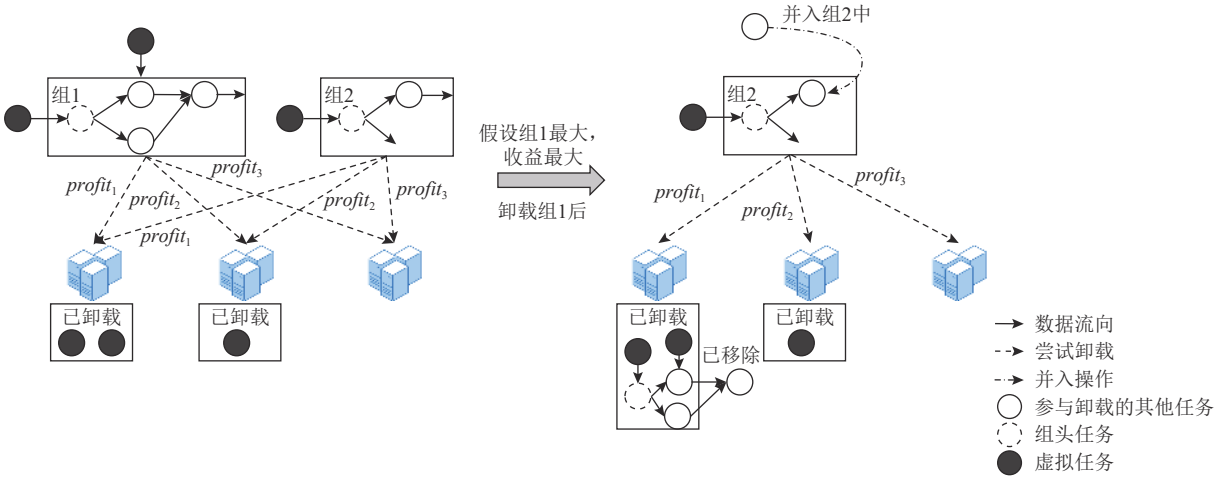


Fig. 7 Illustration of algorithm 2

图 7 算法 2 的图解

重复执行算法 2 行②~㉕过程，直到所有任务卸载完成.

算法 2. GBTO 任务卸载.

输入：网络中的计算节点集合$C \leftarrow \{c_1, c_2, \cdots, c_N\}$、算法 1 生成的任务组集合$G \leftarrow \{g_1, g_2 \cdots, g_s\}$、已卸载任务集合$\mathcal{S} \leftarrow \{t_{s1}, t_{s2}, \cdots, t_{sp}\}$、卸载标记矩阵$\boldsymbol{A}$、任务组数据依赖量矩阵$\boldsymbol{d}$、访问标记向量$\mathcal{V} \leftarrow$ {true,true, ⋯, true}、计算节点资源向量$\boldsymbol{rc}$、任务资源需求向量$\boldsymbol{rt}$;

输出：卸载标记矩阵$\boldsymbol{A}$.

① while $G \neq \varnothing$ do

② for G中的每一组g_i do

③ for C中的每一个计算节点c_j do

④ if V_{ij} = false do

⑤ 继续遍历下一个计算节点;

⑥ end if

⑦ $\mathcal{E}_{ij} \leftarrow$将$g_i$卸载到$c_j$上要删除的任务;

/*g_i部分任务卸载到c_j后，新增数据依赖量*/

⑧ $P_{\text{in}} = sum\{d_{rs} | t_r \in g_i, t_s \in \mathcal{S}, A_{sj} = 1\}$;

/*删除部分任务后减少的数据依赖量*/

⑨ $P_{\text{out}} = sum\{d_{rs} | t_r \in g_i - \mathcal{E}, t_s \in \mathcal{E}_{ij}\}$;

/*卸载成功后带来的数据收益量*/

⑩ $Profit_j = D_i + P_{\text{in}} - P_{\text{out}}$;

⑪ $V_{ij} \leftarrow$ false;

⑫ end for

/*取所有卸载方案中收益最大的一个*/

⑬ $Profit_{\text{imax}} = \max\{Profit_j\}$;

⑭ end for

/*选取本轮具有最大收益的任务组分配方案，并取方案的任务和计算节点的下标*/

⑮ $i, j \leftarrow \max\{Profit_{\text{imax}}\}$ and $g_i = g_i - \mathcal{E}_{ij}$;

/*移除选取方案中需要删除的任务*/

⑯ for $\mathcal{E}_{ij}$中的每一个任务$t$do

⑰ for 每一个在$G - g_i$中的任务组$g_p$do

/*分 2 种情况：第 1 种是如果t的当前未卸载最大数据量前驱在g_p中，那么将t并入g_p中；第 2 种是如果t没有前驱结点或都已卸载，那么单独作为一个任务组*/

⑱ $\begin{cases} g_p = g_p \cup t, \text{if } t\text{的最大father在 } g_p\text{中}, \\ G = G \cup \{t\}, \text{其他}, \end{cases}$

⑲ end for

⑳ end for

/*开始卸载选中的分配方案*/

㉑ for g_i中的每一个任务t_p do

/*卸载后修改计算节点的剩余资源*/

㉒ $rc_j = rc_j - rt_p$;

㉓ end for

㉔ $G = G - g_i$; /*移除已卸载的任务*/

㉕ V_{ij} = true;

㉖ for g_i中的每一个任务t_p do

㉗ A_{pj} = 1;

㉘ end for

㉙ end while

㉚ return $\boldsymbol{A}$.

3）GBTO 复杂度分析

①预处理过程. 比较耗时的 2 个操作为计算任务的拓扑序列和递归分组. 拓扑排序在任务构造时已经完成，故此处不用计算. 合并函数 *merge* 是线性的，时间复杂度为$O(M)$，且是本地执行，不需要额外的空间，空间复杂度为$O(M)$.

②任务卸载阶段. 假设预处理将任务分为了$\mathcal{H}$组，则在第 1 次预卸载过程中需要执行$\mathcal{H} \times N$次尝试. 后面的预卸载中，每组只需要重新计算发生变化的计算节点即可，因此，总共的时间复杂度为

$$O(\mathcal{H} \times \mathcal{H} \times N) + O((\mathcal{H} - 1) + (\mathcal{H} - 2) + \cdots + 1) = \max\left(O\left(\mathcal{H}^2 \times N\right), O\left(\mathcal{H}^2\right)\right), \quad (10)$$

其中$\mathcal{H} \in [1, M]$，而在绝大多数情况下$\mathcal{H}$接近 1，因此 GBTO 最坏时间复杂度为$O(N \times M^2)$，空间复杂度为$O(M)$.

3.4.3 GFBTO 算法实现

GBTO 算法会依据任务组的 CPU 需求判断能否将该组任务分配到某个计算节点上，而计算组 CPU 总需求时只是将组内各任务的 CPU 需求简单相加. 在一个任务组中，任务之间的关系可以分为 2 类：一类是有（直接/间接）前驱后继关系；一类是无依赖关系. 一个任务只有在它的所有前驱任务执行完获取到输入数据后才能运行. 因此具有前驱后继关系的任务间没有并行的必要性，而无依赖关系的多个任务却可以. 如图 8 所示，若让该任务组内无依赖关系的任务并行，则需要的 CPU 可以从 9 减少至 5（其中

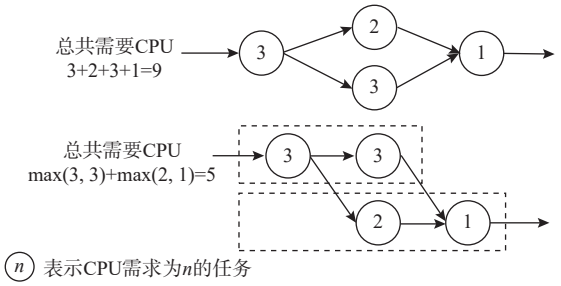


Fig. 8 CPU merge

图 8 CPU 融合

数字表示 CPU 需求). 引入 CPU 资源融合操作后, 能减少组 CPU 需求, 能让更多有依赖关系的任务卸载到同一节点, 就能进一步减少网络中的数据传输. 采用资源融合策略后, 计算节点的负载会变高, 可能会增加 CPU 密集型任务的响应时间, 因此, GBTO 和 GFBTO 的适用场景不同. 任务管理组件会检测各计算节点的状态, 同时会给出使用何种算法的建议.

GFBTO 算法的难点在于如何快速确定一组任务中的CPU 最大并行数. 针对这一难点本文设计了一个基于路径归并的资源融合(road merging-based resource fusion, RMBRF)近似算法. 图 9 和算法 3~5 详细展示了其过程.

GBTO 算法中, 一组任务卸载时, 会根据组内信息判断其是否卸载, 而 RMBRF 通过拓扑排序、路径加入和路径传播 3 个步骤重新计算了组 CPU 总需求信息. 后 2 个步骤细节为:

1)路径加入

从任务组的前若干个任务执行开始, 到所有任务执行完成(或所有任务都得到执行 1 次)会存在多条数据流向, 每一条都是一个依赖路径, 图 9 中的每一个标有数字序列的矩形框就是一条路径. 算法 3 执行时, 会记录从某一个任务往后存在的所有路径, 即算法 3 中的路径集合 $\mathcal{L}_{\text{cur}}$. 路径是从后向前传播的, 当前任务需要选择一条路径加入, 该路径通过调用函数*theBestRoad*得到. 函数*theBestRoad*依照如下原则选择路径: 计算每一条路径中的最大 CPU 需求(算法 4 行①~④), 然后比较当前任务的 CPU 需求和所有路径中的最大 CPU 需求, 若当前任务 CPU 需求较大, 则选择 CPU 需求最小的路径加入, 否则选最大的加入(算法 4 行⑤~⑨). 这一策略使得当前任务加入某一条路径后, 所有路径最大总 CPU 增长最小.

2)路径传播

类似地, 由于存在多个前驱任务, 当前任务还需要选择将某一条路径向前传播给哪一个前驱. 该前驱通过调用函数*theBestFather*(算法 5)得到. 函数*theBestFather*中选择前驱结点, 采用和函数*theBestRoad*一样的策略, 不同的是此时根据单条路径从前驱节点集合中选择单个任务. 图 9 的步骤①~④清晰地描述了路径加入和路径传播 2 个过程, 但求出的结果不是最优, 而是近似最优.

算法 3. RMBRF.

输入: 任务集合$g \leftarrow \{t_1, t_2, \cdots, t_s\}$、任务资源需求向量$\boldsymbol{rt}$、每个任务的前驱结点集合$\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_s$;

输出: 任务集合总的 CPU 需求*cpuNeeds*.

① $cpuNeeds \leftarrow 0$;

/*计算任务集合数据依赖关系拓扑序列*/

② $Q \leftarrow topologicalSequence(g)$;

③ $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_s \leftarrow \varnothing$; /*$\mathcal{L}_i$表示逆任务拓扑序列中能到达任务$i$的路径集合*/

④ $first \leftarrow$ 取拓扑序列最后一个任务的下标;

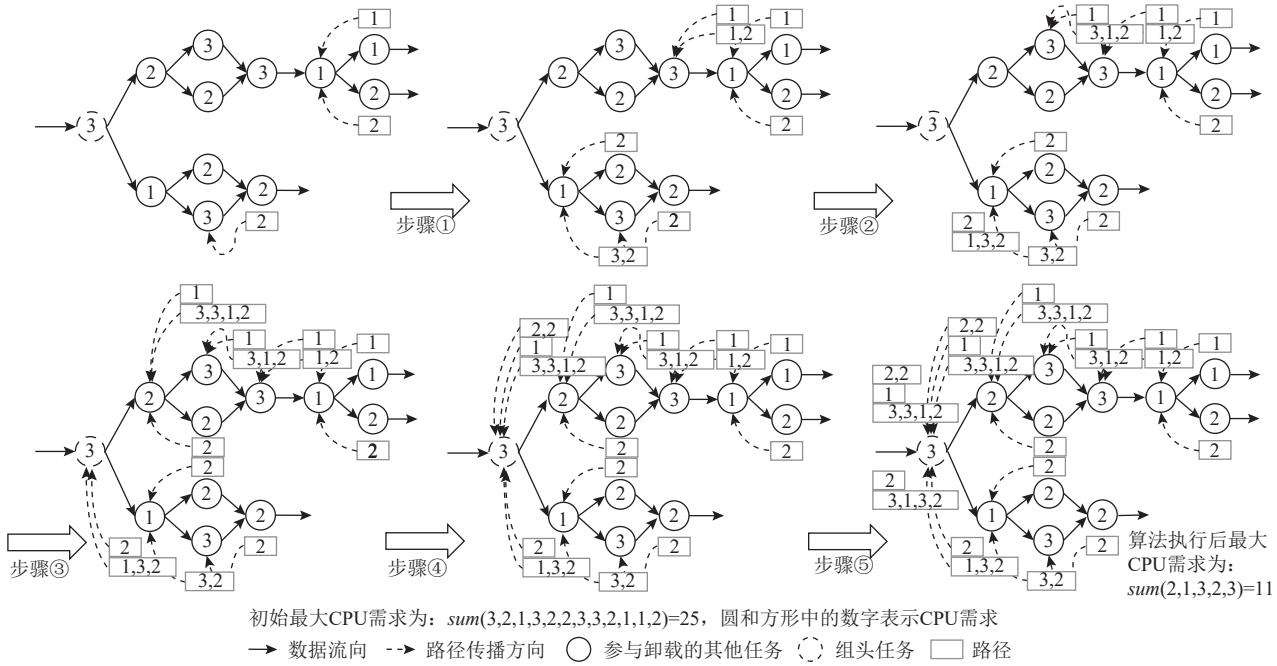


Fig. 9 Road merging-based resource fusion (algorithm 3)

图 9 基于路径归并的资源融合(算法 3)

⑤ while $Q \neq \varnothing$ do

⑥ $t_{cur} \leftarrow Q.top()\&\&Q.pop()$; /*取队列第 1 个任务，并从队列中删除*/

⑦ $l_i \leftarrow theBestRoad(\mathcal{L}_{cur}, rt_{cur})$; /*返回当前任务要加入的路径*/

⑧ $l_i = l_i \cup t_{cur}$;

⑨ for $\mathcal{L}_{cur}$中的每一条路径$l$do

⑩ $j \leftarrow theBestFather(\mathcal{F}_{cur}, l)$; /*为每条路径选一个前驱结点传播*/

⑪ $\mathcal{L}_j = \mathcal{L}_j \cup l$;

⑫ end for

⑬ end while

⑭ for$\mathcal{L}_{first}$中的每一条路径$l$do

⑮ $cpuNeeds = cpuNeeds + \max(l)$;

⑯ end for

⑰ return $cpuNeeds$.

算法 4. *theBestRoad.*

输入：需要计算的任务的可达路径集合$\mathcal{L}_{cur}$、资源需求量rt_{cur}；

输出：当前任务要加入路径序列l.

① $maxs \leftarrow \varnothing$; /*记录每条路径中最大的资源需求量*/

② for $\mathcal{L}_{cur}$中的每一条路径l_i do

③ $maxs = maxs \cup \max(l_i)$; /*将路径的最大资源需求量并入$maxs$中*/

④ end for

⑤ if $rt_{cur} > \max(maxs)$ do

⑥ return l, which $\max(l) = \min(maxs)$;

⑦ else

⑧ return l, which $\max(l) = \max(maxs)$;

⑨ end if

算法 5. *theBestFather.*

输入：需要计算的任务的前驱结点集合$\mathcal{F}_{cur}$、当前路径l；

输出：当前路径需要传播到的前驱结点的下标j.

① $rt_m \leftarrow \max(l)$;

② if $rt_m > \max(\mathcal{F}_{cur})$ do

③ return j, which $rc_j = \min(\mathcal{F}_{cur})$;

④ else

⑤ return j, which $rc_j = \max(\mathcal{F}_{cur})$;

⑥ end if

3）GFBTO 复杂度分析

在 GBTO 算法的基础上，GFBTO 增加了 RMBRF 操作. RMBRF 包含拓扑排序、路径加入和路径传播 3 个步骤. 设任务组中边集为E，则$|E| \in \left[0, \left(M \times \frac{M-1}{2}\right)\right]$. 拓扑排序在任务构造期间已经求得，此处不需要再次计算. 而路径加入和路径传播只会访问每条边 1 次、每个顶点 1 次，因此，时间复杂度均为$O(M+|E|)$. 同时，每个结点和边也只会存储 1 次，因此空间复杂度同样是$O(M+|E|)$.

3.4.4 卸载算法的选择

GFBTO 算法倾向于将更多的任务卸载到同一个计算节点，如果这些任务的计算占比很大，CPU 则会疲于上下文切换，各个任务的有效执行时间将减少，不利于低延时的任务. 因此 GFBTO 算法会倾向于将 I/O 密集型任务放在一起，在部分任务等待 I/O 时 CPU 也能得到更好的利用.

然而，精确地衡量一个任务的计算与 I/O 占比很难，Mort 从 2 个方面进行估计：1）源代码静态分析. 4.1 节介绍了如何从源代码中得到任务组 DAG，在 DAG 中很容易得知一个任务的输入和输出，因此 Mort 假设，一个有更多输入源或输出源的任务的 I/O 占比更高. 2）计算节点状态监控. 源代码分析的结果可能存在偏差，为了能够监控和均衡这一偏差，任务管理组件会实时收集节点的 CPU 和 I/O 利用率.

结合以上 2 点，Mort 采用如下的方式选择卸载算法：计算多输入、多输出任务在整个任务组中的占比，若占比超过一定阈值（默认 70%，是一个经验值，即预估一个作业有 70% 的时间处于 I/O，当处于 I/O 的时间较多时，更多的线程/协程能有效利用 CPU，该值可以配置）就选择 GFBTO，否则选择 GBTO. 同时依照节点的 CPU 利用率对节点进行排序，GFBTO 从高到低选择节点进行卸载，GBTO 由低到高选择节点进行卸载.

以上 2 点考虑目前只用于选择卸载算法，不作为算法运行时的参考因素，不影响算法执行结果.

4 仿真实验

本节将进行仿真实验，比较 Greedy（采用降序首次适应思想实现的贪心算法）、GBTO、GFBTO、OPT（采用 LINGO 18 数学优化软件求解 3.4 节数学模型的最优解）之间的性能.

4.1 仿真实验设置

1）计算节点分布数据集

计算节点分布数据集[44]开放了共享单车骑行数据集，将之处理后得到单车停靠坐标分布图（共 400 多万个点）. 将其按密度划分为多个区域（半径 200 m），

每个区域部署 1 个计算节点, 从而得到 97 个计算节点, 如图 10 所示. 圆形小点为单车停靠点(其中有大量圆形小点的点坐标重叠), 星型为部署的计算节点. 为确保任务能一次性卸载到这些节点上, 节点的总计算能力与作业总计算需求成正相关, 且每个节点的计算能力占比与其管理的点的数量占比正相关. 共享单车停靠坐标, 可能是人类活动的热点区域, 也可能是城市公共智能设施倾斜的地方, 大量的原始数据会在这里产生, 因此计算节点适合布置在这些地方.

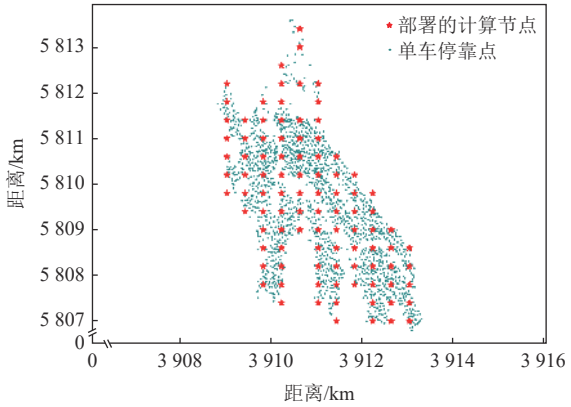


Fig. 10 Distribution of computing nodes

图 10 计算节点分布

2)任务数据集

任务数据集采用的是阿里巴巴 2018 年公开的集群作业数据集[45]. 数据集中包含上千万个作业, 每个作业由 1~30 个任务组成. 实验时将数十数百个原始作业融合成一个大型的作业, 使其具有上百或上千个任务. 每个任务的 CPU 需求参照数据集中的数据, 并适当地缩小, 其范围为 1~3(具有 1 个 CPU 需求的任务占比 60%, 具有 2 个 CPU 需求的任务占比 30%, 具有 3 个 CPU 需求的任务占比 10%). 本文假设任务之间的数据依赖量均匀分布, 其范围为 64~8 192 bps.

4.2 仿真实验结果分析

4.2.1 变化虚拟任务分布模式

图 11 中任务数量固定为 200, 计算节点数目固定为 90, 只需改变虚拟任务的分布模式. 第 1 个(ATCP)是虚拟任务更倾向于分配到计算能力强的计算节点上; 第 2 个(ATCP-R)是虚拟任务更倾向于分配到计算能力弱的计算节点上; 第 3 个则是采用随机分布(Random). 很明显, 在 ATCP 中 GFBTO 的结果超过了 OPT, 原因有 2 点: 1)GFBTO 算法采用了资源融合策略. 资源融合使得每个计算节点能够容纳更多的任务, 而 OPT 模型并没有资源合并操作. 2)虚拟任务被优先分配到了计算能力强的节点上. 结合第 1 点, 这使得 GFBTO 算法执行时会更少地移除任务(因为融合后, 有充足的 CPU 资源), 也就能保留更多的组内任务, 前驱任务产生的数据也就尽可能地被本地消耗, 而不用传输到网络中去. 在 ATCP-R 实验组中, Greedy 效果尤其差. 这是由于大多虚拟任务分配在计算能力差的计算节点上, 而贪心策略看重当前次的收益使得这些节点的空间被浪费, 陷入了较小的局部极值.

4.2.2 变化任务数量

图 12 中计算节点数目固定为 90, 虚拟任务分配方式固定为随机分配, 任务数量从 50 增加到 300. 可以看到, 随着任务数量的增加, 各算法的结果都有所增加, 这是因为任务增多, 总的数据量也相应增加. 与其他实验组规律比较, 不同的有 2 组, 分别是任务数量为 50 和 200 时的对比组. 任务数量为 50 时,

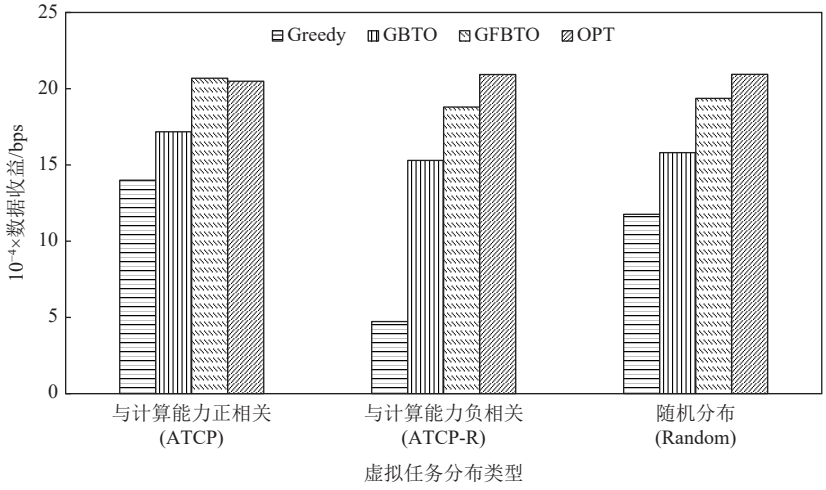


Fig. 11 Data profits and simulated task distribution mode for simulation experiment

图 11 仿真实验中数据收益与虚拟任务分布模式

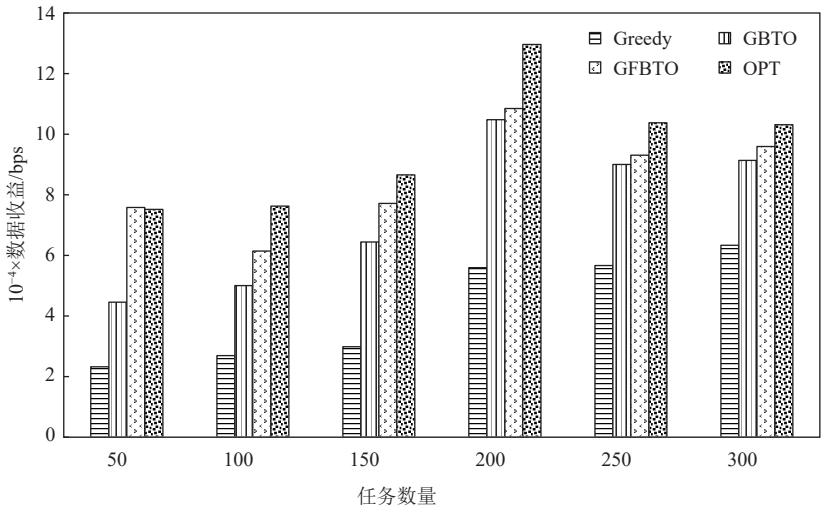


Fig. 12 Data profits and task amount

图 12 数据收益与任务数量

GFBTO 超过了 OPT, 可能的原因是, 此时任务数相对于计算节点数目较少, 资源融合后资源较充足, 更多有依赖关系的任务被卸载到同一计算节点, 与 4.2.1 节中的 ATCP 组相同. 任务数为 200 时, 各算法结果都相对较高, 经查阅原始数据发现, GFBTO 的各任务的平均数据量比其他组高约 20%. 该实验中, 各算法结果规律同 4.2.1 节中的 Random 组类似.

4.2.3 变化计算节点数量

图 13 中, 任务数量固定为 200, 虚拟任务分布为随机分配, 而计算节点的数量从 30 增加到 80. 该实验最能体现算法的不同: 在计算节点较少时, 可选择的也少; 更多有依赖关系的任务被集中在一起, 各个算法的差异也就越小. 而随着计算节点数量的增加, 选择变多, 算法之间的策略差异被放大. 可以看到随着选择变多, 所有算法情况都在变糟, 但 GBTO 和 GFBTO 表现很稳定, 受影响较小.

实验结果表明, 任务在本地的数据消耗量方面, GBTO 平均高出 Greedy 65% 上下, GFBTO 在 GBTO 的基础上平均提升了约 10%, 而且 GBTO 和 GFBTO 分别达到了 OPT 的 82% 和 94% 左右.

5 真实场景实验

本节构造了一个文本统计与分析应用在真实的硬件环境中测试算法的性能.

5.1 场景实验设置

1)软硬件平台

本实验选择了 4 个树莓派 4B(ARM-Cortex-A72, 64b 1. 5GHz, 4-core, 4GB)、1 个树莓派 3B(ARM-Cortex-A53, 64b, 1. 4GHz, 4-core, 1GB)和 1 台 PC(Intel i5-

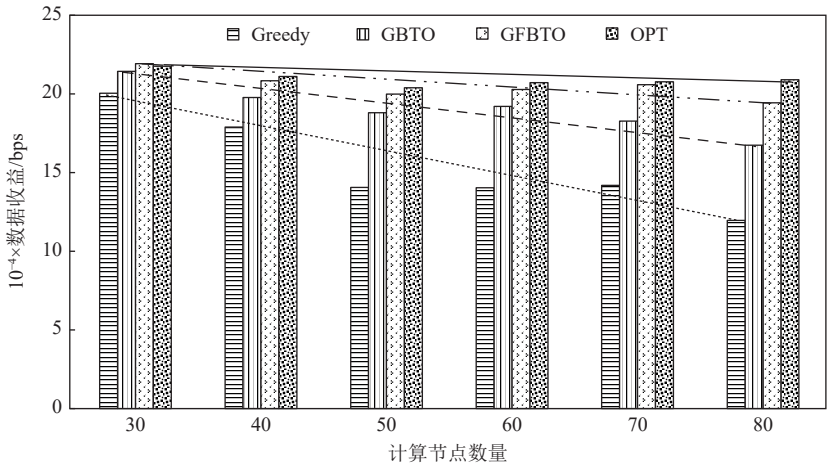


Fig. 13 Data profits and computing nodes amount for simulation experiment

图 13 仿真实验中数据收益与计算节点数量

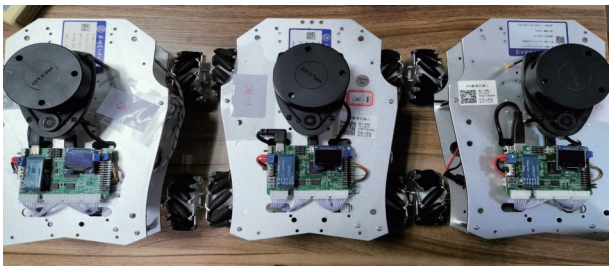


Fig. 14 Hardware platforms

图 14 硬件平台

10500, 3. 10 GHz, 3. 10 GHz, 6-core, 8GB)作为计算节点, 如图 14 所示. 实验中将设备分别编号: 计算节点 node 0(树莓派 3B), 计算节点 node 1~node 4(都是树莓派 4B), 计算节点 node 5(PC). 为确保所有任务都能够被卸载, 各计算节点的计算能力都按其本身的 CPU 核数成比例放大. 每个机器都以 Ubuntu20. 04 LTS 作为操作系统, 并部署了 Mort. 实验过程中, 各任务将以 50 Hz 的速度在 300MB 局域网内发布数据.

2)作业数据集

如图 15 所示, 本文构造了一个文本统计与分析应用. 该应用由多个任务组成, 每个任务负责统计某个区间内数字出现的频率. RawData1 和 RawData2 分别是一个 1MB 的文本文件, F_1~F_6 是应用包含的子任务. 应用中, F_1统计文本中 [0, 100)间的数; F_2统计 [100, 199)间的数; 而F_3统计大于 199 的数; F_4统计 3 个区间中的偶数; F_5统计 3 个区间中的奇数; F_6融合所有的结果. 各任务统计过程中会将结果发布出去. 图 15 中任务上边数字表示每个任务的 CPU 需求, C* 表示全局数据域中的具有C*数据类型的数据域.

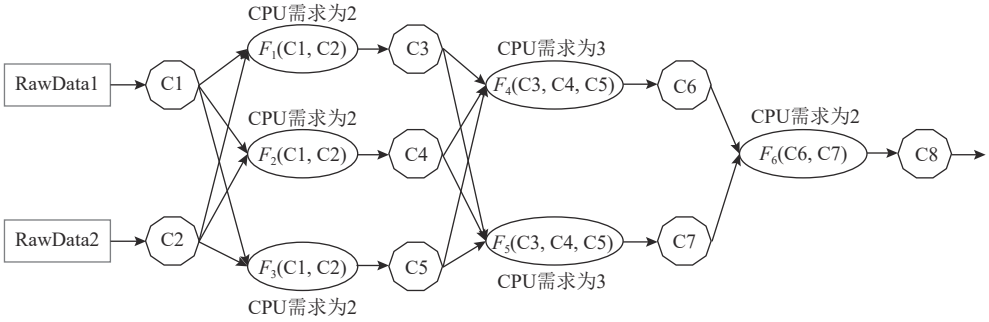


Fig. 15 Real scenario experiment task instance

图 15 真实场景实验作业用例

5.2 场景实验结果分析

实验中分别将 node 4 和 node 5 作为 RawData1 和 RawData2 的数据源传感器. 实验时, node 0 从 node 4 上读取 RawData1 的数据, node 2 从 node 5 上读取 RawData2 的数据. 图 16 和图 17 展示了 3 种算法的效果和仿真实验一致. 图 18 是按照 GBTO 算法生成的执行计划将各个任务分配到 2 个计算节点上(node 0: F_1, F_2, F_3, F_4, F_5, F_6; node2: F_2, F_3, 并且 RawData1 在 node 0 上, RawData2 在 node 2 上)Mort 的各项性能表现. 图 18(a)(b)分别是 2 个计算节点上订阅到的数据延迟统计. 对于 node 0, C2, C4, C5 这 3 种数据类型来自网络, 平均延迟约为 6.34 ms, 而剩下的则是本地处理的数据

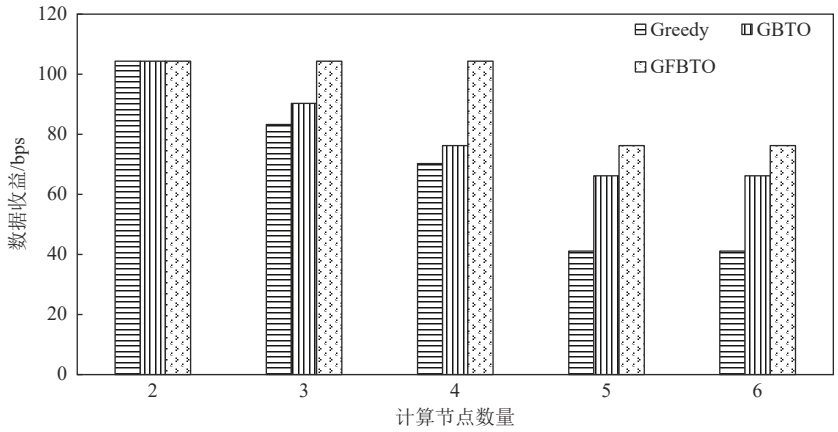


Fig. 16 Data profits and computing nodes amount for scenario experiment

图 16 场景实验中的数据收益与计算节点数量

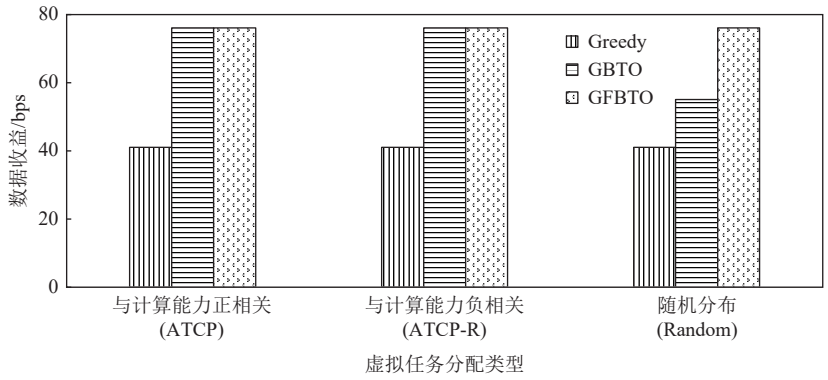


Fig. 17 Data profits and simulated task distribution mode for scenario experiment

图 17 场景实验中数据收益与虚拟任务分布模式

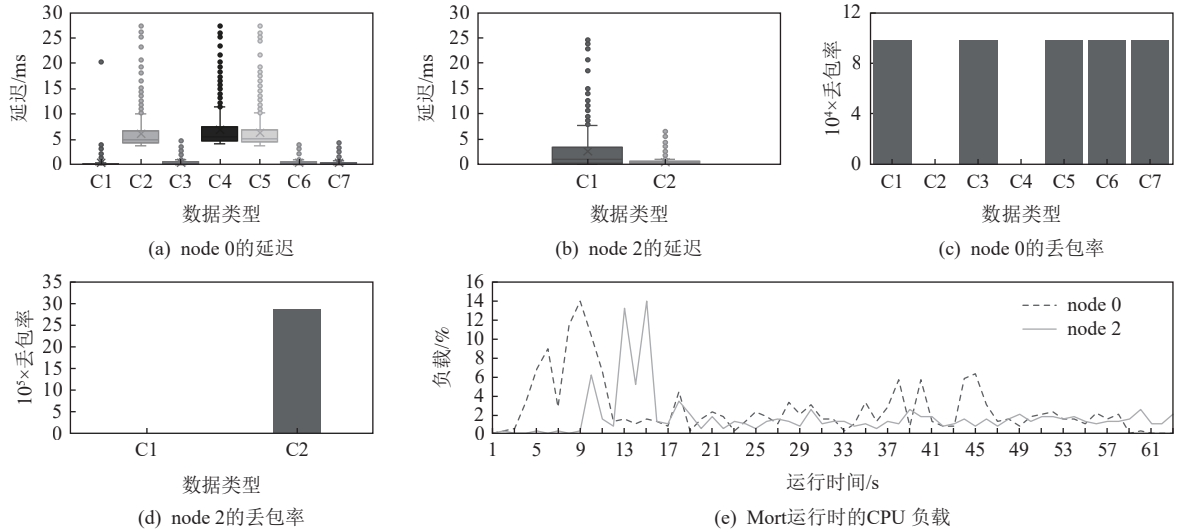


Fig. 18 Mort performance

图 18 Mort 性能表现

类型，平均延迟约为 0.410 ms；对于 node 2，C1 是网络中的数据类型，订阅平均延迟为 3.650 ms，C2 是本地处理的数据类型，平均延迟为 0.4 ms. 图 18(c)(d)是 2 个计算节点的订阅数据的丢包率，其丢包率都为 0，其含有小数是因为计算时浮点数特性造成的. 图 18(e)则是 Mort 运行前、运行时、运行后 CPU 负载变化. node 0 共运行了 4 个发布者和 7 个订阅者；node 2 共运行了 2 个订阅者和 4 个发布者. 可以看到 2 个节点分别在第 3 秒和第 9 秒时启动 Mort，CPU 使用率约有 14% 的增幅。从 Mort 的运行机制分析，负载的大幅升高主要有 3 个原因：1)初始化，包括状态集初始化、访问控制初始化、持久化组件初始化、本地数据域初始化等.2)服务发现，服务分发组件会主动发现网络中其他 Mort 实例.3)任务构造和卸载，初始化操作完成后，Mort 会处理用户应用(如果存在)，包括任务构造和卸载.从第 12 秒和第 16 秒开始，2 个节点的 CPU 使用率大幅下降，比初始时仅高了约 2%。这一阶段，Mort 主要是完成数据分发和状态维护等工作. 在第 59 秒时关闭 Mort.实验表明，Mort 生命周期内，计算节点的负载绝大多数时间处于较低水平.

6 结　　论

本文设计并实现了任务卸载及管理框架 Mort，主要是解决发布订阅系统中的数据共享问题. 任务来自上层业务，需要先对业务进行任务分解. 对于任务分解，采用静态代码分析方法追踪函数调用和参数信息，从而构建 DAG 任务组. 对于任务卸载，将问题建模成优化网络数据传输的非线性整数规划问题. 在偏向于 CPU 密集型任务和计算节点负载高场景中，设计了 GBTO 算法. 在偏向于 I/O 密集型任务和时间敏感场景中，设计了 GFBTO 算法. 仿真和实验表明，GBTO 和 GFBTO 算法能达到最优值的 80% 和 90% 左右，且引入的 Mort 负载只有约 2%. 目前，在

计算任务卸载计划中,无论采用GBTO还是GFBTO,主要依据经验(历史数据分析),未来工作中将探索更好的方式,并考虑包含其他的优秀卸载算法,拓宽使用场景.

Mort的任务管理组件本质上属于分布式节点协调模块,其还存在改进之处,如冲突处理.学界已有很多工作,未来考虑将它们融入组件中.此外,在数据隐私保护方面,Mort目前提供了基于属性的访问控制方式,后续可以引入证书等机制,从计算节点方面进一步提升其安全性.

作者贡献声明: 殷昱煜提供研究思路、审阅论文并提供修改意见;苟红深完成相关实验及论文撰写,并和李尤慧子共同确定算法思路及论文修订;黄彬彬和万健审阅论文并提出修改意见.

参 考 文 献

[1] Internet Data Center. China IOT ecosystem and trends (Chinese Version) [EB/OL]. [2022-05-01]. https://www.idc.com/getdoc.jsp?containerId=IDC_P31060(in Chinese)
(互联网数据中心. 中国物联网发展趋势(中文版)[EB/OL]. [2022-05-01]. https://www.idc.com/getdoc.jsp?containerId=IDC_P31060)

[2] Ericsson. Ericsson mobility report June 2021 [EB/OL]. [2022-05-01].https://www.ericsson.com/zh-cn/about-us/company-facts/ericsson-worldwide/china/5g-for-consumers/ericsson-mobility-report-june-2021-chinese

[3] Hu Miao, Luo Xianzhuo, Chen Jiawen, et al. Virtual reality: A survey of enabling technologies and its applications in IOT[J]. Journal of Network and Computer Applications, 2021, 178: 102970

[4] Chen Zonggan, Zhan Zhihui, Kwong S, et al. Evolutionary computation for intelligent transportation in smart cities: A survey[J]. IEEE Computational Intelligence Magazine, 2022, 17(2): 83−102

[5] Shi Weisong, Cao Jie, Zhang Quan, et al. Edge computing: Vision and challenges[J]. IEEE Internet of Things Journal, 2016, 3(5): 637−646

[6] Wirawan M I, Wahyono D I, Idfi G, et al. IOT communication system using publish-subscribe [C] //Proc of the 2018 Int Seminar on Application for Technology of Information and Communication. Piscataway, NJ: IEEE, 2018: 61−65

[7] Takrouni M, Hasnaoui A, Mejri I, et al. A new methodology for implementing the data distribution service on top of gigabit Ethernet for automotive applications[J]. Journal of Circuits, Systems and Computers, 2020, 29(13): 2050210

[8] Karatas F, Korpeoglu I. Fog-based data distribution service for Internet of things (IOT) applications[J]. Future Generation Computer Systems, 2019, 93: 156−169

[9] Dahlmanns M, Pennekamp J, Fink B I, et al. Transparent end-to-end security for publish/subscribe communication in cyber-physical systems [C] //Proc of the 2021 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems. New York: ACM, 2021: 78−87

[10] Maruyama Y, Kato S, Azumi T. Exploring the performance of ROS2 [C/OL] //Proc of the 13th Int Conf on Embedded Software. New York: ACM, 2016 [2023-03-20].https://dl.acm.org/doi/pdf/10.1145/2968478.2968502

[11] Ma Zhi, Zhang Sheng, Chen Zhiqi, et al. Towards revenue-driven multi-user online task offloading in edge computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 33(5): 1185−1198

[12] Chen Min, Hao Yixue. Task offloading for mobile edge computing in software defined ultra-dense network[J]. IEEE Journal on Selected Areas in Communications, 2018, 36(3): 587−589

[13] Jošilo S, Dán G. Wireless and computing resource allocation for selfish computation offloading in edge computing [C] //Proc of the 2019 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2019: 2467−2475

[14] Ma S, Song S, Yang Lingyu, et al. Dependent tasks offloading based on particle swarm optimization algorithm in multi-access edge computing [J]. Applied Soft Computing, 2021, 112: 107790

[15] Mao Ning, Chen Yuanfana, Guizani M, et al. Graph mapping offloading model based on deep reinforcement learning with dependent task [C] //Proc of the 2021 Int Wireless Communications and Mobile Computing (IWCMC). Piscataway, NJ: IEEE, 2021: 21−28

[16] Tang Zhiqing, Lou Jiong, Zhang Fuming, et al. Dependent task offloading for multiple jobs in edge computing [C/OL]//Proc of the 29th Int Conf on Computer Communications and Networks (ICCCN). Piscataway, NJ: IEEE, 2020 [2023-03-20]. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9209593

[17] Lu Haifeng, Gu Chunhua, Luo Fei, et al. Research on task offloading based on deep reinforcement learning in mobile edge computing[J]. Journal of Computer Research and Development, 2020, 57(7): 1539−1554 (in Chinese)
(卢海峰, 顾春华, 罗飞, 等. 基于深度强化学习的移动边缘计算任务卸载研究[J]. 计算机研究与发展, 2020, 57(7): 1539−1554)

[18] Zhao Gongming, Xu Hongli, Zhao Yangming, et al. Offloading dependent tasks in mobile edge computing with service caching [C]//Proc of the 2020 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2020: 1997−2006

[19] Liu Liuyan, Tan Haisheng, Jiang Shaofeng, et al. Dependent task placement and scheduling with function configuration in edge computing [C/OL]//Proc of the Int Symp on Quality of Service. New York: ACM, 2019 [2023-03-20].https://dl.acm.org/doi/pdf/10.1145/3326285.3329055

[20] Sundar S, Liang Ben. Offloading dependent tasks with communication delay and deadline constraint [C]//Proc of the 2018 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2018: 37−45

[21] Jošilo S, Dán G. A game theoretic analysis of selfish mobile computation offloading [C/OL]//Proc of the 2017 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2017 [2023-03-20]. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8057148

[22] Zhu Tongxin, Li Jianzhong, Cai Zhipeng, et al. Computation scheduling for wireless powered mobile edge computing networks [C]//Proc of the 2020 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2020: 596−605

[23] Ma Xiao, Zhou Ao, ZhangShan, et al. Cooperative service caching and workload scheduling in mobile edge computing [C]//Proc of the 2020 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2020: 2076−2085

[24] Zhang Qiuping, Sun Sheng, Liu Min, et al. Online joint optimization mechanism of task offloading and service caching for multi-edge device collaboration[J]. Journal of Computer Research and Development, 2021, 58(6): 1318−1339 (in Chinese)
(张秋平, 孙胜, 刘敏, 等. 面向多边缘设备协作的任务卸载和服务缓存在线联合优化机制[J]. 计算机研究与发展, 2021, 58(6): 1318−1339)

[25] Tao Ouyang, Li Rui, Chen Xu, et al. Adaptive user-managed service placement for mobile edge computing: An online learning approach [C]//Proc of the 2019 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2019: 1468−1476

[26] Huang Liang, Feng Xu, Zhang Luxin, et al. Multi-server multi-user multi-task computation offloading for mobile edge computing networks [J]. Sensors, 2019, 19(6): 1446

[27] Yu Shuai, Chen Xu, Yang Lei, et al. Intelligent edge: Leveraging deep imitation learning for mobile edge computation offloading[J]. IEEE Wireless Communications, 2020, 27(1): 92−99

[28] Nduwayezu M, Pham Q V, Hwang W J. Online computation offloading in NOMA-based multi-access edge computing: A deep reinforcement learning approach[J]. IEEE Access, 2020, 8: 99098−99109

[29] Wang Jin, Hu Jia, Min Geyong, et al. Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning[J]. IEEE Communications Magazine, 2019, 57(5): 64−69

[30] Min Minghui, Xiao Liang, Chen Ye, et al. Learning-based computation offloading for IOT devices with energy harvesting[J]. IEEE Transactions on Vehicular Technology, 2019, 68(2): 1930−1941

[31] Zhang Weiwen, Wen Yonggang, Wu D O. Energy-efficient scheduling policy for collaborative execution in mobile cloud computing [C] //Proc of the 2013 IEEE INFOCOM. Piscataway, NJ: IEEE, 2013: 190−194

[32] Topcuoglu H, Hariri S, Wu M Y. Performance-effective and low-complexity task scheduling for heterogeneous computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(3): 260−274

[33] Fan Yinuo, Zhai Linbo, Wang Hua. Cost-efficient dependent task offloading for multiusers[J]. IEEE Access, 2019, 7: 115843−115856

[34] Gai Keke, Qiu Meikang, Zhao Hui. Energy-aware task assignment for mobile cyber-enabled applications in heterogeneous cloud computing[J]. Journal of Parallel and Distributed Computing, 2018, 111: 126−135

[35] Cuervo E, Balasubramanian A, Cho D K, et al. MAUI: Making smartphones last longer with code offload [C] //Proc of the 8th Int Conf on Mobile Systems, Applications, and Services. New York: ACM, 2010: 49−62

[36] Kao Y H, Krishnamachari B. Optimizing mobile computational offloading with delay constraints [C] //Proc of the 2014 IEEE Global Communications Conf. Piscataway, NJ: IEEE, 2015: 2289−2294

[37] Chen M H, Liang Ben, Dong Min. Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point [C/OL]//Proc of the 2017 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2017 [2023-03-20].https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8057150

[38] Allen F E. Control flow analysis [C/OL]//Proc of the ACM SIGPLAN Notices. New York: ACM, 1970 [2023-03-20].https://dl.acm.org/doi/pdf/10.1145/390013.808479

[39] Kildall G A. A unified approach to global program optimization [C]//Proc of the 1st Annual ACM SIGACT-SIGPLAN Symp on Principles of Programming Languages. New York: ACM, 1973: 194−206

[40] Alnefaie S, Alshehri S, Cherif A. A survey on access control in IOT: Models, architectures and research opportunities[J]. International Journal of Security and Networks, 2021, 16(1): 60−67

[41] Schlesselman J M, Castellote G P, Farabaugh B. OMG data-distribution service (DDS): Architectural update [C] //Proc of the 2004 IEEE Military Communications Conf. Piscataway, NJ: IEEE, 2005: 961−967

[42] Wu Tianze, Wu Baofu, Wang Sa, et al. Oops! It's too late. Your autonomous driving system needs a faster middleware[J]. IEEE Robotics and Automation Letters, 2021, 6(4): 7301−7308

[43] He Kun, Meng Xiaozhu, Pan Zhizhou, et al. A novel task-duplication based clustering algorithm for heterogeneous computing environments [J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 30(1): 2−14

[44] Amazon. Shared bikes trip data [EB/OL]. [2022-05-01].https://s3.amazonaws.com/tripdata/JC-202110-citibike-tripdata.csv.zip

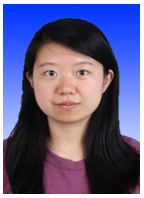
[45] Alibaba. Microservices traces [EB/OL]. [2022-05-01].https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021

Yin Yuyu, born in 1980. PhD, professor. Member of CCF. His main research interests include service computing, edge computing, and business process management.

殷昱煜, 1980 年生. 博士, 教授. CCF 会员. 主要研究方向为服务计算、边缘计算、业务流程管理.

Gou Hongshen, born in 1996. Master. His main research interests include service computing, edge computing and workflow scheduling.

苟红深, 1996 年生. 硕士. 主要研究方向为服务计算、边缘计算和工作流调度.



Li Youhuizi, born in 1989. PhD, associate professor. Member of CCF. Her main research interests include edge computing, privacy security, mobile edge computing, and energy efficiency system.

李尤慧子, 1989 年生. 博士, 副教授. CCF 会员. 主要研究方向为边缘计算、隐私安全、移动边缘计算和高效能系统.

Huang Binbin, born in 1984. PhD, associate professor. Her main research interests include cloud computing, mobile edge computing and workflow scheduling.

黄彬彬, 1984 年生. 博士，副教授. 主要研究方向为云计算、移动边缘计算和工作流调度.

Wan Jian, born in 1969. PhD, professor. His main research interests include distributed systems, computer networks, and big data analytics.

万　健, 1969 年生. 博士，教授. 主要研究方向为分布式系统、计算网络和大数据分析.