

# HFlow: 在 HPC 系统上高效管理高通量应用

戴屹钦<sup>1†</sup>, 王睿伯<sup>1†</sup>, 袁昊<sup>1</sup>, 董勇<sup>1\*</sup>, 陈娟<sup>1</sup>, 张惠泽<sup>1</sup>, 邵明天<sup>1</sup>, 卢凯<sup>1</sup>, 樊春<sup>2,3</sup>

1. 国防科技大学计算机学院, 长沙 410073

2. 北京大学计算中心, 北京 100871

3. 北京大学长沙计算与数字经济研究院, 长沙 410000

\* 通信作者. E-mail: yongdong@nudt.edu.cn

† 同等贡献

收稿日期: 2025-05-28; 修回日期: 2025-09-08; 接受日期: 2025-11-13; 网络出版日期: 2026-01-19

国家自然科学基金青年科学基金 (批准号: 62302514, 62502531)、国家自然科学基金创新群体项目 (批准号: 62421002) 和新疆生产建设兵团科技发展专项资金 (批准号: 2025AB027) 资助项目

**摘要** 高通量计算通常需要执行众多规模较小、运行时间较短且相互独立的计算任务. 尽管高性能计算系统拥有丰富的计算资源, 但主流资源管理系统及现有高通量应用管理方案在吞吐量、应用兼容性和容错性方面存在显著缺陷, 导致高性能计算 (high-performance computing, HPC) 系统上针对高通量应用的资源管理效率低下. 针对这一问题, 本文提出 HFlow—一种融合集中式与分布式资源管理架构的资源管理解决方案. HFlow 通过混合作业管理机制实现高应用兼容性, 并基于细粒度任务划分算法与多级容错机制同步提升吞吐量与容错性. 在天河-2A 超级计算机上的实验评估结果显示, HFlow 能在维持高通量计算 (high-throughput computing, HTC) 应用管理效率的前提下成功兼容 HTC 应用资源管理需求, 其任务吞吐量显著优于主流资源管理系统及专用 HTC 方案, 并具备多级容错能力. 具体地, HFlow 相比于主流资源管理系统及相关通量型应用资源管理方案实现了 2.1~108.3 倍的任务吞吐量.

**关键词** 高性能计算, 高通量计算, 资源管理系统, 作业调度, 任务划分

## 1 引言

高通量计算<sup>[1,2]</sup> (high-throughput computing, HTC) 是一种用于执行大规模计算任务的计算范式. HTC 应用由大量 HTC 任务组成, 这些任务通常具有规模小 (通常每个任务使用一个计算核心)、运行时间短 (通常为几秒至十几秒)、松耦合 (任务之间没有依赖关系) 和数量大 (从数千到数十万甚至更多) 的特点. 目前, HTC 应用广泛出现在基因测序<sup>[3]</sup>、信息安全<sup>[4]</sup>、材料科学<sup>[5]</sup> 和机器学习<sup>[6,7]</sup> 等领域中. 高性能计算 (high-performance computing, HPC) 系统中集成了大量紧耦合的计算节点, 其丰富的计算资源使其成为 HTC 应用的良好运行平台. 然而, HPC 系统的资源管理系统主要是针对以批处理应用为主的传统 HPC 应用设计的. 与 HTC 应用不同, 批处理应用具有运行时间较长和作业规模较大的特点, 且不会在短时间内要求资源管理系统调度和监控大量任务.

引用格式: 戴屹钦, 王睿伯, 袁昊, 等. HFlow: 在 HPC 系统上高效管理高通量应用. 中国科学: 信息科学, 2026, 56: 378–393, doi: 10.1360/SSI-2025-0222

Dai Y Q, Wang R B, Yuan H, et al. HFlow: efficiently manage high-throughput applications on HPC systems. Sci Sin Inform, 2026, 56: 378–393, doi: 10.1360/SSI-2025-0222

目前,使用主流资源管理系统对 HTC 应用进行管理主要存在以下两方面问题<sup>[8]</sup>.一方面,主流集中式 HPC 系统资源管理系统(如 Slurm<sup>[9]</sup>, PBS<sup>1)2)</sup>和 LSF<sup>3)</sup>) 在管理 HTC 任务时,容易因单个集中式管理节点上不可避免的性能瓶颈造成吞吐量不足的问题.另一方面,HTC 应用中的大量任务可能会淹没资源管理系统的管理资源,从而影响其他非 HTC 应用的管理效率.鉴于上述原因,当前 HPC 系统难以高效利用海量计算资源服务于 HTC 任务.因此,HTC 应用在 HPC 系统上被称为一种尴尬的并行.通过分析天河系列 HPC 系统<sup>[10]</sup>上 HTC 应用的运行场景,本文提出在 HPC 系统上高效管理 HTC 应用所需要的能力特性.

- **高吞吐量:** 实现每秒数万个任务的吞吐量,以满足当今 HTC 应用的需求.吞吐量需求可认为是 HTC 应用的核心需求.

- **高兼容性:** 在不影响 HPC 应用管理效率的前提下,提高针对 HTC 应用的管理效率.

- **高容错性:** 自动监控 HTC 应用异常并从异常中恢复.

目前,一些 HPC 系统中的分布式资源管理方案(如 Flux<sup>[11]</sup>)已经表明,将资源管理能力分散到系统多个节点能够有效缓解集中式瓶颈,从而显著提升任务吞吐率.在此背景下,集中式与分布式资源管理系统的融合方案成为同时高效管理 HPC 与 HTC 应用的可行途径.事实上,已有若干典型工作尝试将集中式和分布式资源管理系统结合使用<sup>[12,13]</sup>,虽然这些方案多是针对特定应用需求设计的,但其融合思路对 HPC 与 HTC 应用的统一管理具有普遍意义.然而,现有融合工作大多仅将资源管理能力简单分布到每个计算节点上,缺乏对这种设计所带来的额外开销进行细粒度分析和优化,因此在任务吞吐率与容错性等方面仍存在进一步提升空间.

针对上述问题,本节提出了一种名为 HFlow 的资源管理解决方案.通过融合集中式与分布式资源管理架构,HFlow 提供了兼容 HPC 和 HTC 应用的通用混合作业管理机制.在 HFlow 框架中,由集中式资源管理系统管理 HPC 应用(与传统 HPC 系统应用管理模式相同).当 HTC 应用到达时,首先由集中式资源管理系统为 HTC 应用分配总体计算资源,然后由分布式资源管理系统在分配的计算资源上具体管理 HTC 应用,避免影响集中式资源管理系统对 HPC 应用的管理效率.上述设计确保了 HFlow 对 HPC 和 HTC 应用的高兼容性.此外,为了提高任务吞吐量和容错性,HFlow 还实现了细粒度的任务划分算法和多级容错机制.

本文在天河-2A 超级计算机<sup>[14]</sup>上对 HFlow 进行了实验评估,将 HFlow 与各种主流资源管理系统以及相关 HTC 应用资源管理方案进行了比较.实验重点评估了 HFlow 的任务吞吐量、兼容性和容错性.结果表明,HFlow 能够在几乎不影响 HPC 应用管理效率的前提下显著提升任务吞吐量并实现有效的多级容错.具体地,HFlow 相比于主流资源管理系统及相关 HTC 应用资源管理方案可以实现 2.1~108.3 倍的任务吞吐量.

本文的主要贡献如下.

- 本文提出了一种在 HPC 系统上融合集中式与分布式资源管理架构的资源管理方案 HFlow,其通过混合作业管理机制高效地实现对 HPC 和 HTC 应用的兼容资源管理.

- 本文为 HFlow 设计了细粒度任务划分算法和多级容错机制,以实现高吞吐量和高容错性.

- 本文在真实 HPC 系统中证明了 HFlow 的高吞吐量、高兼容性和高容错性.

## 2 相关工作

尽管集中式资源管理系统在 HPC 系统生态中已趋于成熟和稳定,但其固有的集中式瓶颈仍然限制了对 HTC 应用的高效管理.例如,典型的集中式系统 Slurm 每秒仅能调度数百个任务(详见第 5.2 节),远不足以满足在大规模计算核心上运行 HTC 应用的吞吐量需求.Teno<sup>[15]</sup>针对 HPC 系统中的 HTC 应用,将 Slurm 的集中式调度优化为分层调度以实现细粒度资源分配,并通过改进传统 Master-Worker 模型以提升吞吐率和资源利用效率.

为解决 HTC 应用的吞吐量问题,一些研究提出了专用框架.HTDcr<sup>[16]</sup>通过应用分解、预分配、全局任务池和分层调度提升了吞吐量和任务执行效率.Falkon<sup>[17]</sup>作为轻量级任务执行框架,通过多级调度与动态资源分

1) Altair, 2025. <https://www.altair.com/pbs-professional/>.

2) OpenPBS, 2025. <https://openpbs.org/>.

3) IBM. Lsf, 2022. <https://www.ibm.com/products/hpc-workload-management>.

配实现了大规模任务的高效调度. HTCondor<sup>4)</sup>是一种支持 HTC 应用的资源管理系统,它依托高效的匹配机制和容错执行在异构、共享资源环境中实现高吞吐调度.

随着在 HPC 系统上运行 HTC 应用的场景日益增多,一些工作尝试实现 HPC 与 HTC 应用的融合调度.典型的分布式系统 Flux<sup>[11]</sup>通过将资源管理能力分布至多个节点缓解集中式瓶颈,从而提升整体吞吐率.HTCaaS<sup>[18]</sup>通过自动化的元作业拆分、资源选择和作业提交机制,在跨异构环境中实现大规模 HTC 作业的透明管理和执行.

在此基础上,部分研究进一步探索集中式与分布式资源管理的融合管理方案以实现 HPC 和 HTC 应用的统一融合管理.Radical-Pilot<sup>[19]</sup>(RP)是一种基于 pilot 的资源管理与任务执行框架,其核心思想是将传统作业提交与资源调度解耦.RP 首先通过集中式资源管理系统提交一个 pilot 作业,随后在 pilot 作业中,由 RP 的分布式调度代理接收、分发和执行后续实际任务.另一类代表性方案将具体的集中式系统 Slurm 与具体分布式系统 Flux 简单融合:Slurm 负责节点分配,而 Flux 在分配节点内调度任务.本文称其为 FluxP,并在第 4.2.1 节中对其实现思路进行详细介绍.本文提出的 HFlow 也属于此类融合框架.它通过框架设计实现 HPC 和 HTC 应用的融合管理,在保证 HPC 作业管理效率不变的前提下,通过细粒度调度优化提升任务吞吐量以满足 HTC 应用的需求,同时兼顾容错性.相比于 Radical-Pilot 和 FluxP, HFlow 的核心增量是针对 HPC 和 HTC 融合场景进行了更加完整的系统设计和更为细粒度的优化,以同时实现高吞吐量、高兼容性和高容错性.

### 3 背景知识

#### 3.1 高通量计算

高通量计算是一种旨在提升任务处理效率的计算范式,其核心目标是在单位时间内处理尽可能多的计算任务.高通量计算将工作负载分解为众多规模小、结构相似且相互独立的任务,通过任务的高并行处理提高计算效率.例如,基因组学相关 HTC 应用将需要处理的数十万乃至数百万的独立基因序列分布到多个任务上进行处理<sup>[20~23]</sup>.类似地,药物开发相关 HTC 应用需要对候选化合物进行大规模的虚拟筛选以确定潜在的候选药物,这通常涉及运行成千上万的化学计算和分子对接任务<sup>[24~27]</sup>.目前,考虑到 HPC 系统具有海量的计算资源,在 HPC 系统上运行 HTC 应用的实践越来越多<sup>[22, 23, 27, 28]</sup>.

#### 3.2 集中式资源管理系统

集中式资源管理系统 (centralized resource management system, CRM) 是 HPC 系统的主流资源管理系统.CRM 采用主从结构,通常在全系统中仅设置一个管理节点.该管理节点拥有 HPC 资源和工作负载的全局视图,负责管理整个系统中的资源和工作负载.典型的集中式资源管理系统包括 Slurm<sup>[9]</sup>, PBS 和 LSF.然而,单管理节点的计算和通信性能始终是有限的,这一瓶颈阻碍了 CRM 在短时间内管理大量并发计算任务,进而阻碍了 HTC 应用在 HPC 系统上的高效运行.例如,Slurm 的最大任务吞吐量通常为每秒数百个任务(详见第 5.2 节).在面对使用数千乃至数万计算核心的 HTC 应用时,这种吞吐量并不足以充分发掘和利用大规模计算资源的潜力.

#### 3.3 分布式资源管理系统

分布式资源管理系统 (distributed resource management system, DRM) 将资源管理和调度功能分散到多个计算节点上,以缓解单个管理节点的性能瓶颈.目前,DRM 大致可以分为以下三类.

- **多管理节点类:** 设置多个管理节点,各管理节点管理一个系统中节点和任务的子集,典型实现如 Slurm++<sup>[29]</sup>.

- **单管理节点多辅助节点类:** 保留单一管理节点负责全局资源管理和调度,设置多个辅助节点实时从管理节点动态卸载具体的资源管理任务,典型实现如 ESlurm<sup>[30]</sup>.

4) HTCondor overview. 2025. <https://htcondor.org/htcondor/overview/>.

• **嵌套体系结构类:** 定义实例概念, 各实例独立管理局部资源并执行作业调度, 且支持无限深度的嵌套派生. 实例的动态部署/终止特性赋予其原生兼容集中式系统的优势, 典型实现如 Flux<sup>[11]</sup> 和 RP<sup>[19]</sup>.

## 4 HFlow 方案设计

当前, 集中式与分布式资源管理系统的融合部署已逐渐成为解决 HPC 与 HTC 应用兼容问题的重要发展方向. 与两类系统的独立部署相比, 融合部署在体系架构与运维管理层面均展现出更显著的优势. 在资源层面, 融合部署能够有效避免独立部署带来的资源同步开销. 当两套资源管理系统分别管理部分资源时, 若存在资源重叠, 则在作业调度过程中需要维护资源状态一致性并引入锁机制, 从而增加额外的同步成本; 若资源完全不重叠, 虽然避免了同步问题, 但会导致 HPC 与 HTC 作业可用资源在部署时被静态划分, 缺乏灵活性, 难以适应两类作业比例的动态变化. 在运维层面, 融合部署能够显著降低系统使用与管理的复杂度. 独立部署意味着需要同时维护两套资源管理 API 与操作接口, 用户和管理员均需要额外学习和掌握不同的使用方法与运维机制, 增加了系统的复杂性与管理负担. 相比之下, 融合部署能够提供统一的接口与一体化的管理流程, 从而简化系统使用并降低运维成本. HFlow 采用的正是这种集中式与分布式资源管理系统融合部署的方案.

本节首先介绍 HFlow 的框架设计和混合作业管理机制, 说明 HFlow 如何保证高兼容性. 然后, 介绍 HFlow 的细粒度任务划分算法, 该算法通过对 HTC 应用进行细粒度的任务划分以实现高任务吞吐量. 接着, 介绍 HFlow 的多级容错机制, 该机制通过监控并重新提交异常的实例或任务, 保证在管理 HTC 应用时的高容错性. 最后, 介绍 HFlow 如何扩展到更复杂的 HTC 作业场景. 本文中应用、作业和任务的概念如下: 当 HPC 应用到达 HPC 系统时, 每个 HPC 应用对应于一个 HPC 作业; 当 HTC 应用到达 HPC 系统时, 每个 HTC 应用对应于一个 HTC 作业, 其中包含大量独立的 HTC 任务. 在下文中, 为了表述清晰, 使用 HPC 作业指代 HPC 应用和 HPC 作业的概念, 使用 HTC 作业指代 HTC 应用和 HTC 作业的概念.

### 4.1 HFlow 框架设计及混合作业管理机制

图 1 展示了 HFlow 的框架及其混合作业管理流程. HFlow 由 5 个部分组成: CRM (图中黄色部分)、DRM (图中浅橙色部分)、HFlow 标准接口 (图中紫色部分)、HTC 作业管理器 (图中橙色部分) 和 HTC 数据库 (图中灰色部分). CRM 和 HTC 数据库持久地运行在 HPC 系统上. DRM 和 HTC 作业管理器则在 HTC 作业到达 HPC 系统时临时实例化, 并在 HTC 作业结束时终止. 当 HPC 系统中没有 HTC 作业时, HPC 系统中仅有 CRM 和 HTC 数据库运行, 且由 CRM 负责管理 HPC 作业. 这与传统 HPC 系统中的资源管理环境几乎相同, 保证了 HFlow 对 HPC 作业的高效管理. 对于每个 HTC 作业, HFlow 则实时为其实例化两层 DRM 实例和一个 HTC 作业管理器, 由 CRM 和 DRM 共同管理该 HTC 作业. 这种动态部署方法可以最大限度地减少管理 HTC 作业对 HPC 作业管理效率的影响.

#### 4.1.1 HFlow 框架针对 HPC 作业的管理流程

用户在登录节点通过 HFlow 标准接口提交 HPC 作业 ①, 该接口将 HPC 作业资源分配请求发送到 CRM 管理节点 ②. 在收到作业的资源分配请求后, CRM 管理节点执行资源分配, 为 HPC 作业分配唯一的作业号, 并向 HFlow 标准接口通知资源分配结果 ③. 同时, CRM 管理节点通知每个被分配的计算节点在该节点上的计算核心上加载 HPC 作业进程 ④ 和 ⑤. 当 HPC 作业运行时, CRM 负责监控和维护 HPC 作业的实时状态 ⑥.

#### 4.1.2 HFlow 框架针对 HTC 作业的管理流程

用户在登录节点上通过 HFlow 标准接口提交 HTC 作业 ⑦. 随后, HFlow 立刻初始化一个包含解析组件、提交组件和监控组件的 HTC 作业管理器 ⑧. 解析组件将大量的 HTC 任务拆分为多个 HTC 任务脚本 ⑨. 提交组件将整个 HTC 作业的资源分配请求提交给 CRM 管理节点 ⑩. CRM 管理节点执行资源分配, 为该 HTC 作业分配唯一作业号, 向 HTC 作业管理器和 HFlow 标准接口通知资源分配结果 ⑪. 同时, CRM 管理节点将资源分配结果通知所有分配的计算节点 ⑫.



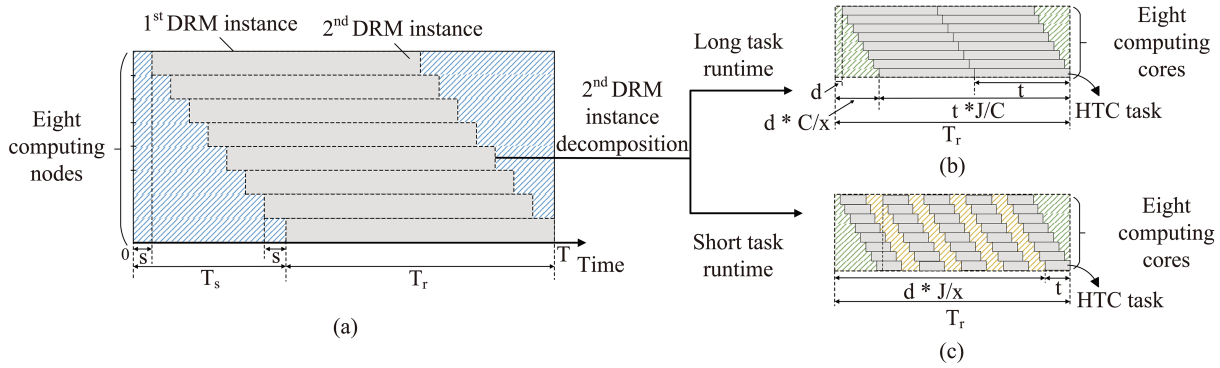


图 2 (网络版彩图) 采用平均任务划分方法时 HTC 作业运行时间分解示意图. (a) 第一层 DRM 实例时间分解. (b) HTC 任务运行时间较长时第二层 DRM 实例时间分解. (c) HTC 任务运行时间较短时第二层 DRM 实例时间分解.

Figure 2 (Color online) Schematic diagram of HTC job runtime decomposition with the average task division method. (a) Time decomposition of the 1st DRM instance. (b) Time decomposition of the 2nd DRM instance when the HTC task runs for a long-time. (c) Time decomposition of the 2nd DRM instance when the HTC task runs for a short time.

本节首先介绍现有使用 CRM 和 DRM 融合管理 HTC 作业的传统场景, 并分析该场景下平均任务划分过程引起的吞吐量限制问题. 然后, 本节介绍 HFlow 如何通过细粒度任务划分算法以优化子任务脚本数和各任务脚本中 HTC 任务数, 从而进一步提高任务吞吐量.

#### 4.2.1 传统 HTC 任务划分问题

目前, 现有的 CRM 和 DRM 融合管理方案采用完全平均的任务划分方法<sup>[13, 28]</sup>. 具体地, 第一层实例管理分配给该 HTC 作业的所有计算节点, 并在每个计算节点上都初始化一个第二层实例, 每个第二层实例平均地分担所有 HTC 任务. 图 2 展示了采用完全平均的任务划分方法时 HTC 作业运行时间的分解图. 图 2(a) 表示第一层实例运行时间分解, 其中包括多个第二层实例的运行时间, 图 2(b) 和 (c) 则表示第二层实例运行时间分解. 图中第一层实例共管理 8 个计算节点, 每个计算节点对应于一个第二层实例, 每个第二层实例管理该节点上的 8 个计算核心. 这种完全平均分配的任务划分方法造成了大量的资源闲置, 图中使用不同颜色的阴影表示不同原因造成的资源闲置, 具体原因如下.

首先, 第二层实例的串行初始化导致第一层实例资源闲置 (图 2(a) 蓝色阴影). 由于 DRM 实例的动态部署特性, 第一层实例需要为每个第二层实例实时分配资源, 该过程需要通过锁机制保证资源一致性, 导致资源分配呈现串行性. 第二层实例的串行初始化会导致第一层实例运行时间首端出现资源闲置. 当各第二层实例承载等量 HTC 任务时, 还会进一步导致第一层实例运行时间尾端出现资源闲置.

其次, 在第二层实例内串行地调度 HTC 任务会导致每个第二层实例内的资源闲置 (图 2(b) 和 (c) 绿色阴影). 当第二层实例调度 HTC 任务时, 需要为每个任务分配计算核心. 类似地, 为了保证资源的一致性, HTC 任务的调度也是串行的, 导致每个第二层实例运行时间的首端和尾端出现资源闲置. 此外, 如果每个 HTC 任务的运行时间较短, 串行调度还会导致第二层实例运行时间的中部出现资源闲置 (图 2(c) 黄色阴影).

针对上述问题, 本节提出一种细粒度的 HTC 任务划分算法. 该算法根据 HTC 作业特征和可用计算资源数量确定任务脚本的最优数量 (即第二层实例的最优数量) 以及各任务脚本中 HTC 任务的最优数量 (即每个第二层实例中的 HTC 任务数量), 以进一步提高任务吞吐量.

#### 4.2.2 优化第二层实例数量

如图 2 所示, 第一层实例资源闲置时间 (图中蓝色部分) 与第二层实例数量呈正相关, 第二层实例资源闲置时间 (图中绿色部分) 与其管理的计算核心数呈正相关. 当 HTC 任务运行时间较短时, 第二层实例资源闲置时间 (图中黄色部分) 还与任务数呈正相关. 因此, 减少第二层实例数量可缩短第一层实例中的资源闲置时间, 但会增加第二层核心数与任务数, 导致第二层实例中的资源闲置时间增长. 相反地, 增加第二层实例数量则可缩短

第二层实例中的资源闲置时间,但会增加第一层实例中的资源闲置时间.因此,优化第二层实例数量即求出能够使 HTC 作业总运行时间最短的第二层实例数量.本节暂时假设各第二层实例均分 HTC 总任务数,并在此基础上寻找最佳第二层实例数量.如何细粒度地调整各第二层实例中的任务数量参见第 4.2.3 节.

本节使用  $x$  表示第二层实例的数量.  $s$  表示初始化每个第二层实例的平均时间开销.  $d$  表示在第二层实例中调度每个 HTC 任务的平均时间开销.  $J$  表示 HTC 作业中的任务总数.  $C$  表示可用计算核心总数.  $t$  表示每个 HTC 任务的运行时间,其中每个 HTC 任务使用一个计算核心.  $T(x)$  表示当第二层实例数量为  $x$  时, HTC 作业的总运行时间.  $T(x)$  由两部分时间组成,

$$T(x) = T_s + T_r, \quad (1)$$

其中  $T_s$  表示初始化第二层实例时在第一层实例首端造成的资源闲置时间,可由式 (2) 计算:

$$T_s = x \cdot s. \quad (2)$$

$T_r$  表示第二层实例的运行时间,与每个 HTC 任务的运行时间直接相关.如图 2(b) 所示,当每个 HTC 任务的运行时间大于第二层实例为其所管理的每个计算核心顺序调度一轮 HTC 任务所花费的时间(即  $t \geq d \cdot \frac{C}{x}$ ) 时,观察在第二层实例的最后一个核心上运行的任务,可知  $T_r$  的值由第二层实例首端的空闲时间和在最后一个核心上运行的多个 HTC 任务的总运行时间组成.此时,  $T_r$  可由式 (3) 计算:

$$T_r = d \cdot \frac{C}{x} + t \cdot \frac{J}{C}, \quad (3)$$

其中  $\frac{C}{x}$  表示每个第二层实例管理的计算核心数,  $\frac{J}{C}$  表示每个计算核心上的 HTC 任务数.如图 2(c) 所示,当每个任务的运行时间小于第二层实例为其管理的每个计算核心顺序调度一轮 HTC 任务所花费的时间(即  $t < d \cdot \frac{C}{x}$ ) 时,  $T_r$  的值由调度第二层实例中所有 HTC 任务的总时间开销和一个 HTC 任务的运行时间组成.此时,  $T_r$  可由式 (4) 计算:

$$T_r = d \cdot \frac{J}{x} + t, \quad (4)$$

其中  $\frac{J}{x}$  是每个第二层实例中 HTC 任务的数量.基于式 (1)~(4),  $T(x)$  可由式 (5) 计算:

$$T(x) = \begin{cases} x \cdot s + \frac{1}{x} \cdot C \cdot d + \frac{t \cdot J}{C}, & t \geq d \cdot \frac{C}{x}, \\ x \cdot s + \frac{1}{x} \cdot J \cdot d + t, & t < d \cdot \frac{C}{x}, \end{cases} \quad (5)$$

其中变量  $x$  作为第二层实例的候选值,其取值范围可以是  $[1, C]$  内能够整除  $C$  的任意整数.通过遍历每一个候选值对应的总运行时间  $T(x)$ ,便可找到在该范围内使得  $T(x)$  最小的候选值,并将其作为第二层实例的数量.

图 3(a) 和 (b) 的对比直观地说明了优化第二层实例数量的效果.具体地,优化前共有 10 个第二层实例,每个第二层实例对应于一个计算节点,优化后的第二层实例则减少为 5 个,每个第二层实例对应于两个计算节点.这种优化减少了 HTC 作业的总体运行时间,进而提高了任务吞吐量.

需要强调的是,式 (5) 中的 3 个参数  $s$ ,  $d$  和  $t$  均为与系统特性密切相关的静态参数,包括资源调度开销、实例启动延迟和任务执行开销等.在实际应用细粒度任务划分算法之前,应通过预实验确定其具体取值,并在后续所有计算中将其视为常量.具体而言,采用如下方式进行参数确认,以避免动态干扰对算法评估的影响,同时增强实验设计的客观性和可复现性.

- $s$  (第二层实例启动时间): 使用传统 HTC 任务划分方法运行大量 `sleep(0)` 任务 (数量大于系统核心数),记录各第一层实例的启动时间,并计算其平均值作为  $s$  的取值.

- $d$  (第二层实例调度单个任务的时间开销): 统计使用传统 HTC 任务划分方法运行大量 `sleep(0)` 任务时每个第二层实例的总运行时间,并除以其调度的任务数量,得到  $d$  的估算值.由于 `sleep(0)` 任务本身不消耗执行时间,所以该值可真实反映第二层实例调度单个任务的平均开销.

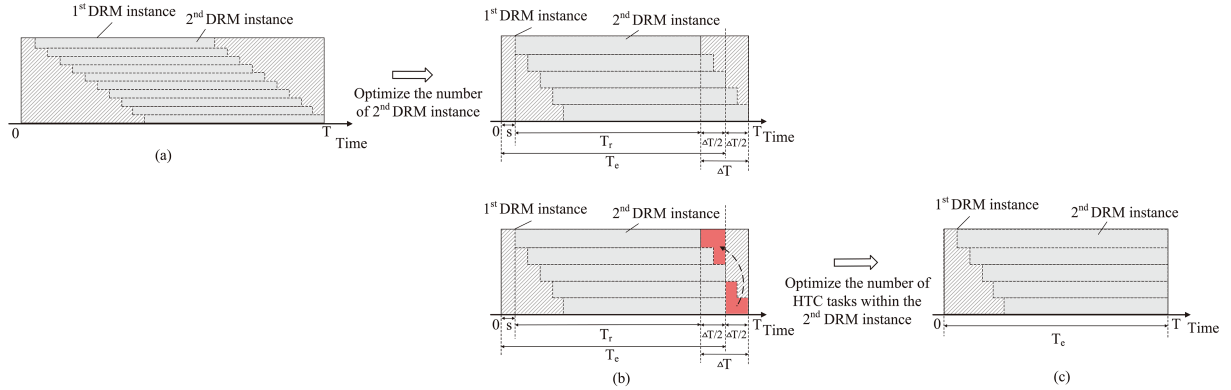


图 3 (网络版彩图) HFlow 任务划分算法效果示意图. (a) 未经优化的 HTC 作业运行时间; (b) 优化第二层实例数量后的 HTC 作业运行时间; (c) 优化第二层实例中 HTC 任务数量后的 HTC 作业运行时间.

**Figure 3** (Color online) Schematic diagram of the effect of the task partitioning algorithm in HFlow. (a) The unoptimized runtime of the HTC job; (b) the runtime of the HTC job after optimizing the number of 2nd DRM instances; (c) the runtime of the HTC job after optimizing the number of HTC tasks in each 2nd DRM instance.

- $t$  (单个 HTC 任务运行时间): 通过多次运行单个 HTC 任务以测量每个 HTC 任务的平均运行时长, 得到  $t$  的取值.

#### 4.2.3 优化第二层实例中的 HTC 任务数量

第 4.2.2 节方法假设各第二层实例均分 HTC 作业的所有任务. 实际上, 在确认第二层实例数量之后, 对各第二层实例中的 HTC 任务数量进行细粒度调整可以进一步提高任务吞吐量. 优化第二层实例中的 HTC 任务数量的核心考虑是, 更早初始化完成的第二层实例可以更快地开始调度和执行 HTC 任务, 从而允许它们容纳更多 HTC 任务. 观察图 3(b) 下图, 对第二层实例中 HTC 任务数量进行调整的目标是使所有第二层实例尽可能同时完成 HTC 任务的调度和执行, 从而减少第一层实例尾端的资源闲置. 具体方法是, 首先找到优化后的预期运行时间 (由  $T_e$  表示), 然后将在该预期时间之后运行的 HTC 任务转移到该预期时间之前的资源闲置时间内运行 (见图 3(b) 下图中的箭头), 从而使运行时间降低到  $T_e$ .

如图 3(b) 所示, 第一个第二层实例运行完成的时间和最后一个第二层实例运行完成的时间之间的间隔由第二层实例的串行初始化造成, 用  $\Delta T$  表示, 并由式 (6) 计算:

$$\Delta T = (x - 1) \cdot s. \quad (6)$$

基于  $\Delta t$ , 可以通过式 (7) 计算优化后 HTC 作业的预期总运行时间  $T_e$ :

$$T_e = T_r + s + \Delta T/2, \quad (7)$$

其中  $T_r$  表示 HTC 任务在各第二层实例中均匀分布时每个第二层实例的运行时间, 计算方法详见上一节. 基于上述分析, 使用式 (8) 调整每个第二层实例中的 HTC 任务数量:

$$H_i = \left\lceil \frac{J}{x} \cdot \left( 1 + \frac{T_e - T_i}{T_r} \right) \right\rceil, \quad (8)$$

其中  $H_i$  表示优化后第  $i$  个第二层实例中 HTC 任务的数量;  $\frac{J}{x}$  表示 HTC 任务在各第二层实例中均匀分布时, 每个第二层实例中 HTC 任务的个数.  $T_i$  表示从第一层实例初始化完成 (图 3 中时间为 0 的点) 到第  $i$  个第二层实例完成运行的时间.  $T_i$  由式 (9) 计算:

$$T_i = T_r + i \cdot s. \quad (9)$$

式 (8) 中,  $T_e - T_i$  表示第  $i$  个第二层实例相比于采用平均任务划分方法时需要减少或增加的运行时间,  $\frac{T_e - T_i}{T_r}$  表示第  $i$  个第二层实例需要减少或增加的运行时间与采用平均任务划分方法时每个第二层实例的运行时间之间的

比值. 由于  $\frac{J}{x}$  个 HTC 任务的运行时间已知为  $T_r$ , 因此可以利用上述比值来获得第  $i$  个第二层实例相比于采用平均任务划分方法时需要增加或减少的 HTC 任务数. 综合式 (6)~(9), 得到每个第二层 DR 实例中的 HTC 任务数量最终计算公式为

$$H_i = \left\lceil \frac{J}{x} \cdot \left( 1 + \frac{(1+x-2 \cdot i) \cdot s}{2 \cdot T_r} \right) \right\rceil. \quad (10)$$

图 3(b) 和 (c) 的对比直观地展示了优化第二层实例中 HTC 任务数量后的效果.

#### 4.2.4 针对 HTC 作业的细粒度任务划分算法流程

如算法 1 所示, 算法的输入包括总可用计算核心的数量, HTC 作业中 HTC 任务的总数, 初始化每个第二层实例的平均时间开销, 在第二层实例中调度每个 HTC 任务的平均时间开销, 以及每个 HTC 任务的运行时间, 算法的输出则是第二层实例的数量和每个第二层实例中的 HTC 任务数量.

---

**算法 1** HFlow 细粒度任务划分算法.

**输入:** 总可用计算核心数量  $C$ , HTC 作业中的总 HTC 任务数  $J$ , 初始化每个第二层实例的平均时间开销  $s$ , 在第二层实例中调度每个 HTC 任务的平均时间开销  $d$ , 每个 HTC 任务的运行时间  $t$ .

**输出:** 第二层实例数量  $x$ , 每个第二层实例中的 HTC 任务数量的集合  $H = \{H_1, H_2, \dots, H_x\}$ .

```

1:  $X \leftarrow \{k \mid 1 \leq k \leq C, C \bmod k = 0\}$ ;
2: 对于任意  $x \in X$ , 使用式 (5) 计算  $T(x)$ ;
3:  $x \leftarrow \operatorname{argmin}_{x \in X} T(x)$ ;
4:  $H_{tmp} \leftarrow 0$ ;
5:  $i \leftarrow 1$ ;
6: while  $i \leq x - 1$  do
7:   使用式 (10) 计算  $H_i$ ;
8:    $H_{tmp} \leftarrow H_{tmp} + H_i$ ;
9:    $i \leftarrow i + 1$ ;
10: end while
11:  $H_x \leftarrow J - H_{tmp}$ ;
返回:  $x, H$ .

```

---

算法 1 首先确定第二层实例数量的候选集合  $X$  (第 1 行). 然后, 对于每一个第二层实例数量的候选值, 算法使用式 (5) 计算其对应的 HTC 作业总运行时间  $T(x)$  (第 2 行), 并选择能使 HTC 作业总运行时间最短的候选值  $x$  作为第二层实例的数量 (第 3 行). 最后, 算法使用式 (10) 来确定每个第二层实例所管理的 HTC 任务的数量 (第 6~11 行). 由于该算法选取的候选集合  $X$  中候选值的数量为  $\log C$  个, 且算法中仅包含单层循环, 因此算法复杂度为  $O(\log C)$ .

#### 4.3 HFlow 针对 HTC 作业的多级容错机制

在 HTC 作业执行期间, HTC 作业管理器中的监控组件 (见图 1) 持续监控两层 DRM 实例和各 HTC 任务的状态. 当检测到状态异常时, HFlow 触发以下多级容错机制.

- **系统级容错:** 指第一层实例发生故障时触发的容错机制. 由于第一层实例负责管理系统中所有分配给该 HTC 作业的计算资源, 因此第一层实例的故障的影响是系统级的. 当监控组件发现第一层实例运行异常时, 监控组件无法通过第一层实例及依附于其存在的第二层实例获得所有 HTC 任务的完成状态. 此时, 监控组件检查存储在 HTC 数据库中的所有 HTC 任务的状态, 并将所有未完成的任务作为一个新的 HTC 作业, 由提交组件重新提交到 HPC 系统上运行.

- **实例级容错:** 指第二层实例发生故障时触发的容错机制. 当监控组件发现某个第二层实例运行异常时, 监控组件无法通过该第二层实例获得其管理的所有 HTC 任务的完成状态. 此时, 监控组件在 HTC 数据库中检查由异常的第二层实例管理的 HTC 任务的状态, 并将未完成的任务组织成一个新的任务脚本. 随后, 提交组件要求第一层实例初始化一个新的第二层实例, 并且将新生成的任务脚本提交到该第二层实例中运行.

• **任务级容错:** 指正在运行的 HTC 任务发生故障时触发的容错机制. 当监控组件发现某个 HTC 任务运行异常时, 监控组件将不再生成新的任务脚本, 而是由提交组件直接在第一层实例下重新提交该 HTC 任务. 这种方法避免了频繁生成第二层实例, 同时确保第一层实例在发现有可用计算资源时能够立即调度异常的 HTC 任务.

#### 4.4 HFlow 针对复杂 HTC 作业的扩展

上述针对 HFlow 框架、算法和机制的描述均默认每个 HTC 任务的运行时间是已知且相同的. 实际上, 这一假设已经满足大部分的 HTC 作业场景, 因为 HTC 作业通常用于处理大量独立子任务 (如数据分块处理和蒙特卡罗 (Monte Carlo) 模拟等), 这些任务在设计时会被均匀分割为相同规模的计算单元, 使得 HTC 任务同质化明显. 此时, 即便每个 HTC 任务的运行时间是未知的, 也可以通过运行单个 HTC 任务提前得知所有 HTC 任务的运行时间. 然而, 现实世界中仍然可能存在更为复杂的 HTC 作业, 即各 HTC 任务运行时间不同的 HTC 作业. 为了扩大 HFlow 适用性, 本节对 HFlow 进行了扩展.

当用户提交的 HTC 作业中的 HTC 任务的运行时间不不同时, HTC 作业管理器的解析组件首先对 HTC 任务进行分类, 将运行时间相同的 HTC 任务分为同一类. 然后解析组件 HTC 对用户申请的计算资源先进行一次划分, 以确定每类 HTC 任务所使用的计算资源. 具体地, 任务解析组件使用式 (11) 确定每类 HTC 任务所需要的计算核心数量:

$$C_i = \frac{J_i \cdot t_i}{\sum_{i=1}^y (J_i \cdot t_i)} \cdot C, \quad (11)$$

其中  $C_i$  表示分配给第  $i$  类任务的计算核心数量,  $J_i$  表示第  $i$  类任务的任务总数,  $t_i$  表示第  $i$  类任务的运行时间,  $J_i \cdot t_i$  表示第  $i$  类任务占用的核心秒数. 为使每类 HTC 任务尽可能同时运行完成以最小化资源闲置, 式 (11) 保证分配给每类 HTC 任务的核心数之间的比例等于该类型 HTC 任务运行所需要的核心秒数与所有 HTC 任务运行所需要的总核心秒数之间的比例. 在为每类任务分配总计算资源后, 对于每一类 HTC 任务, HTC 作业管理器将其视为一个独立的 HTC 作业, 并对每一个 HTC 作业执行后续管理操作 (图 1⑨~⑱). 此时, 每一个 HTC 作业中的 HTC 任务运行时间相同, 并对应于一个两层 DRM 实例.

## 5 实验评估

### 5.1 实验设置

• **HFlow 实现:** 在 HFlow 的具体实现中, 本节使用 Slurm<sup>[9]</sup> 作为集中式资源管理系统的具体实现, 使用 Flux<sup>[11]</sup> 作为分布式资源管理系统的具体实现.

• **硬件平台:** 本节在天河 -2A 超级计算机<sup>[14]</sup> 上部署并评估了 HFlow. 天河 -2A 超级计算机上的每个计算节点都有 64 GB 的内存和两个 12 核 2.2 GHz 的英特尔至强处理器. 本节中最大规模的实验使用了天河 -2A 超级计算机上的 64k 个计算核心.

• **实验路线:** 实验重点评估了 HFlow 的任务吞吐量、兼容性和容错性. 第一轮实验比较了各种针对 HTC 作业的资源管理方案的任务吞吐量 (详见第 5.2 节). 第二轮实验评估了 HFlow 的兼容性 (详见第 5.3 节). 第三轮实验评估了 HFlow 的多级容错机制 (详见第 5.4 节). 第四轮实验评估了 HFlow 中任务划分算法的开销 (详见第 5.5 节).

### 5.2 任务吞吐量评估

任务吞吐量是指 HPC 系统单位时间内可以完成的任务的数量. 这一指标反映了 HPC 系统短时间内管理大量任务的能力. 通常, HPC 作业并不需要过高的任务吞吐量. HTC 作业任务数庞大, 对任务吞吐量的要求较高. 任务吞吐量需求可看作 HTC 应用的核心需求. 本实验分别在每个 HTC 任务的运行时间相同和不同的两种情况下对各类 HTC 作业资源管理方案的任务吞吐量进行评估, 并将 HFlow 与多种 HTC 作业资源管理方案进

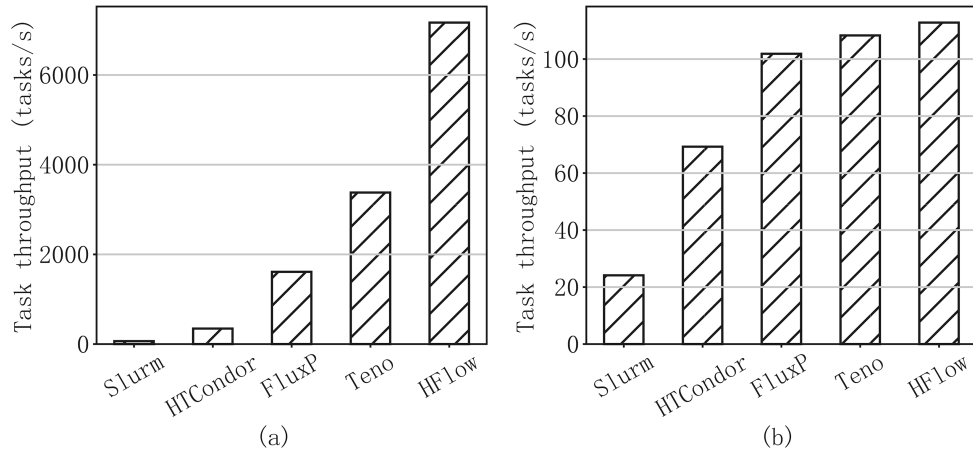


图4 各资源管理方案的任务吞吐量对比。(a) 运行 sleep(0) 任务时; (b) 运行各种 sleep 任务时。

Figure 4 Comparison of task throughput of various resource management schemes when running (a) sleep(0) tasks and (b) mixed sleep tasks.

行对比。需要注意的是, 本实验没有选取那些针对特定作业或平台设计的资源管理方案, 因为它们并不适用于通用资源管理场景。用于对比的 HTC 作业资源管理方案包括如下 4 种。

- **Slurm:** 使用批处理作业的提交方式直接向集中式资源管理系统 Slurm<sup>[9]</sup> 提交 HTC 作业。Slurm 主要针对 HPC 应用设计。

- **Teno:** 使用 Teno<sup>[15]</sup> 工具管理 HTC 作业。Teno 基于 Slurm 实现分层调度以提高任务吞吐量, 因此可以看作集中式资源管理系统针对 HTC 应用的优化版本。Teno 的目标是同时支持 HPC 和 HTC 应用。

- **HTCondor:** 使用 HTCondor<sup>[21]</sup> 管理 HTC 作业。HTCondor 是一个专为 HTC 作业设计的资源管理系统, 目前在科学计算、大数据分析和工程模拟等领域均有所应用。

- **FluxP:** 使用集中式资源管理系统 Slurm 和分布式资源管理系统 Flux<sup>[11]</sup> 的简单融合系统管理 HTC 作业, 并使用如第 4.2.1 节所述的传统方案进行部署 (即一个计算节点对应于一个第二层实例的部署方案)。FluxP 的目标是在集中式资源管理系统高效管理 HPC 应用的基础上引入分布式资源管理系统以增强 HTC 应用管理效率。

本实验采用当前高通量测试领域标准化的 sleep(0) 任务评估当 HTC 任务运行时间相同时各资源管理方案的任务吞吐量。由于该 sleep(0) 任务实际运行时间为零, 其产生的 HTC 作业总运行时间完全由资源管理方案的管理开销构成, 因此可有效测试系统吞吐量极限。在 1200 个计算核心的硬件环境下, 所有资源管理方案均被要求执行 10000000 个 sleep(0) 任务 (每个任务占用 1 个核心)。任务运行完毕后, 使用 HTC 作业总运行时间除以任务总量获得各资源管理方案的任务吞吐量。

实验结果如图 4(a) 所示。由于集中式资源管理系统固有的集中式瓶颈, Slurm 的任务吞吐量较低, 具体为每秒 66.2 个任务。HTCondor 作为专为 HTC 应用设计的资源管理系统, 通过 ClassAd 资源匹配机制等技术将吞吐量提升至每秒 346.3 个任务。然而, 由于 HTCondor 未采用分布式技术, 其吞吐量仍然受限。FluxP 采用了集中式和分布式资源管理系统融合的方式, 它使用 Slurm 分配计算资源, 然后将调度能力分布到每个已分配的计算节点上, 这种分布式设计使得任务吞吐量进一步提高到每秒 1610.9 个任务。然而, 由于采用的是简单的任务划分机制, 资源空闲时间较多 (详见第 4.2.1 节), FluxP 的任务吞吐量仍存在较大的优化空间。通过对集中式资源管理系统 Slurm 的优化, Teno 的分层调度思想使得任务吞吐量达到每秒 3378.8 个任务。HFlow 在本实验中实现了最高的任务吞吐量。通过集中式和分布式资源管理系统的无缝融合, 同时结合细粒度任务划分算法, HFlow 的任务吞吐量可达每秒 7168.4 个任务。总体上看, HFlow 的任务吞吐量是 Slurm 的 108.3 倍, 是 HTCondor 的 20.7 倍, 是 FluxP 的 4.4 倍, 是 Teno 的 2.12 倍。总体上看, 与其他资源管理方案相比, 在任务运行时间相同时, HFlow 实现了 2.12~108.3 倍的任务吞吐量。

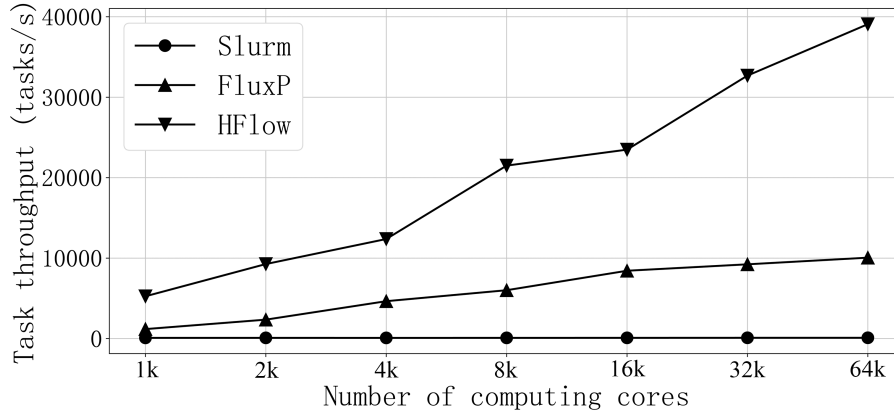


图 5 任务吞吐量随计算核心数量的变化.

Figure 5 The variation of task throughput with the number of computing cores.

为了评估 HTC 任务运行时间不同时各资源管理方案的吞吐量, 本实验生成了 60000 个运行时间不同的 HTC 任务, 包括 sleep(1), sleep(2), sleep(4), sleep(8), sleep(16) 和 sleep(32), 每种任务有 10000 个. 本实验使用各种资源管理方案在 1200 个核心上执行这些任务, 每个任务使用一个计算核心, 实验结果如图 4(b) 所示. Slurm 和 HTCCondor 的任务吞吐量仍然较低, 分别为每秒 24.1 个任务和每秒 69.26 个任务. FluxP 实现了每秒 101.9 个任务的吞吐量, Teno 实现了每秒 108.3 个任务的吞吐量. HFlow 的任务吞吐量最高, 为每秒 112.8 个任务. 总体上看, 与其他资源管理方案相比, 在任务运行时间不同时, HFlow 实现了 1.04~4.68 倍的任务吞吐量.

为进一步量化任务吞吐量与计算核心规模的动态关系, 本实验基于 sleep(0) 任务开展任务吞吐量的扩展性测试: 在各计算核心规模配置下, 任务规模与核心数保持 1000:1 的固定比例, 每个任务使用一个核心. 例如, 当计算核心数量为 64k 时, 共运行 64M 个 sleep(0) 任务. 实验结果如图 5 所示. 在所有规模下, Slurm 的任务吞吐量最低, 均不超过每秒 105 个任务. 相比之下, FluxP 和 HFlow 的任务吞吐量都随着计算核心数量的增加而显著增加, 且 HFlow 的任务吞吐量增幅更大. 在 64k 核心规模下, FluxP 的任务吞吐量达到每秒 10060.5 个任务, HFlow 的任务吞吐量则接近每秒 40k 个任务 (每秒 39080.8 个任务). 这样的任务吞吐量水平完全可以满足当前天河 HPC 系统上所有 HTC 作业的需求.

### 5.3 兼容性评估

本实验评估 HFlow 的兼容性, 即评估 HFlow 是否能够在不影响 HPC 作业管理效率的前提下支持 HTC 作业. 在本实验中, 以 HPC 作业的平均等待时间和平均降速作为衡量 HPC 作业管理效率的指标. 其中, 平均等待时间表征各 HPC 作业自提交至被调度执行的时间间隔均值, 平均降速表征各 HPC 作业的等待时间与运行时间总和相对于运行时间的比值. 本实验使用 HFlow 同时运行 HPC 作业和 HTC 作业, 衡量运行不同数量的 HTC 作业对 HPC 作业管理效率的影响. 本实验中运行的 HPC 作业是从天河 -2A 系统的历史作业序列中取出的 100 个 HPC 作业, 每个 HTC 作业包含 10000 个 sleep(0) 任务. 本实验中 HPC 作业和 HTC 作业使用的计算资源没有交集. 这是因为本实验的核心目标是验证管理 HTC 作业中海量的 HTC 任务是否会淹没资源管理系统的管理资源, 进而影响 HPC 作业的管理效率. 若允许两类作业共享资源, 则 HPC 管理效率的波动将源自计算资源争用而非 HTC 任务规模对资源管理系统造成的管理压力.

实验结果如表 1 所示. 从表中可以看出使用 HFlow 管理 HTC 作业几乎不会影响 HPC 作业的平均等待时间和平均降速. 具体数据表明, 与仅运行传统 HPC 作业场景相比, 混合负载场景下 (HPC 作业与 1~3 个 HTC 作业并发执行) 的 HPC 作业管理效率保持稳定, 验证了 HFlow 的兼容性. 该特性归因于 HFlow 的核心架构设计: 在 HFlow 中集中式资源管理系统 (本实验采用 Slurm) 仅需为 HTC 作业执行单次全局资源分配, 后续海量 HTC 任务的具体管理任务均由分布式资源管理系统承担, 从而尽可能地降低 HTC 作业对传统 HPC 作业管理效率的影响.

表 1 使用 HFlow 管理 HTC 作业对 HPC 作业管理效率的影响.

Table 1 The impact of using HFlow to manage HTC jobs on the management efficiency of HPC jobs.

Number of HTC jobs	0	1	2	3
Average job waiting time of HPC jobs (s)	35.9	35.8	35.9	36.0
Average job slowdown of HPC jobs	4.6	4.6	4.6	4.6

表 2 HFlow 多级容错机制评估结果.

Table 2 Evaluation results of the multi-level fault-tolerant mechanism of HFlow.

Failure location	Number of submitted tasks	Number of resubmitted tasks	Number of successful tasks
No failure	122084	0	122084
Failures on 1st DRM instance	138217	16133	122217
Failures on 2nd DRM instance	134356	12272	122148
Failures on HTC task	123304	1220	122084

#### 5.4 多级容错机制评估

本实验使用 PLSG (protein-ligand simulation using GROMACS)<sup>[31]</sup> 作为 HTC 作业以评估 HFlow 的多级容错机制. PLSG 在 16k 个计算核心上总共运行 122084 个 HTC 任务, 每个任务使用一个计算核心.

本实验手动模拟多级故障, 检查各级故障情况下总共提交的 HTC 任务总数、重新提交的 HTC 任务数和运行成功的 HTC 任务的总数. 第一级故障是第一层实例的故障, 通过杀死第一层 DRM 守护进程实现. 第二级故障是第二层实例的故障, 通过一次性随机杀死 10% 的第二层 DRM 守护进程实现. 由于一个第二层实例中只要有一个守护进程被杀死, 该第二层实例就会出现故障, 因此杀死 10% 的第二层 DRM 守护进程会引起远超 10% 的第二层实例故障. 第三级故障是单个 HTC 任务的故障, 通过一次性随机杀死 10% 的正在运行的 HTC 任务实现. 尽管实际生产环境中故障触发源具有多样性 (如硬件失效、软件崩溃及网络拥塞等), 但本实验设计的层级化故障注入模型已涵盖 HFlow 管理 HTC 作业的核心故障模式 (即系统级故障、实例级故障和任务级故障, 见第 4.3 节).

实验结果如表 2 所示. 在第一层实例发生故障并运行异常时, HFlow 监控组件从 HTC 数据库中检索未完成任务, 并将未完成任务组织为一个新的 HTC 作业, 然后重新初始化两层 DRM 实例以管理该 HTC 作业. 在新的两层 DRM 实例初始化后, 未完成的 HTC 任务被继续提交到系统上运行. 在上述过程中, 那些在第一层实例发生故障时正在运行的 HTC 任务会被 HFlow 提交两次. 由于每个任务使用一个核心, 所以重提交的任务总数不会超过计算核心的总数 (即 16k 个). 然而, 由于 HTC 数据库更新的延迟, 一些已完成的任务没有及时将完成状态刷新到 HTC 数据库, 重提交的 HTC 任务数量略高于 16k 个 (具体为 16133 个), 因此成功完成的 HTC 任务数量略大于 HTC 任务总数 (具体为 122217 个).

在第二层实例发生故障并运行异常时, HFlow 监控组件从 HTC 数据库中检索由异常第二层实例管理的未完成任务, 将它们组织成一个新的作业脚本, 并在当前的第一层实例下重新初始化一个新的第二层实例以管理这一作业脚本. 理论上, 只有那些由异常第二层实例管理的且在该第二层实例发生故障时正在运行的任务会被提交两次. 然而, 由于 HTC 数据库更新延迟, 一些已成功完成但其完成状态没有及时被刷新到 HTC 数据库的 HTC 任务也会被 HFlow 重提交. 具体地, 重提交的 HTC 任务数为 12272 个, 成功完成的 HTC 任务数量为 122148 个.

当 HTC 任务发生故障并运行异常时, HFlow 会立即在第一层实例上重新提交任务. 由于此时两层 DRM 实例运行正常, 因此监控组件可以通过定时查询 DRM 实例所管理的 HTC 任务实时状态来及时发现 HTC 任务的故障. 上述容错过程无需对 HTC 数据库进行检索操作, 确保了异常任务的精确重提交而不受 HTC 数据库更新延迟的影响. 因此, 重提交的任务数等于出现异常的 HTC 任务数 (具体为 1220 个), 成功完成的任务数量等于 HTC 任务总数 (具体为 122084 个).

表 3 HFlow 任务划分算法的时间开销.

Table 3 The time cost of the HFlow's task division algorithm.

Number of HTC tasks	1k	10k	100k	1M	10M
Division cost when the task runtime is the same (s)	2.8	2.9	2.9	3.2	3.4
Division cost when the task runtime is different (s)	2.9	2.9	3.3	3.6	4.0

## 5.5 任务划分算法时间开销评估

如前所述, HFlow 在为 HTC 作业分配计算资源后, 将大量的 HTC 任务划分为多个作业脚本 (见图 1⑨) 并提交到 HPC 系统执行. 由于在任务划分的过程中, 已分配的计算资源处于空闲状态, 所以任务划分的时间开销将直接影响任务吞吐量. 本实验评估面对不同规模的 HTC 作业时 HFlow 任务划分算法的时间开销. 实验共使用两种 HTC 作业, 第一种 HTC 作业包括大量的 `sleep(0)` 任务, 用于评估任务运行时间相同时任务划分算法的时间开销. 第二种 HTC 作业包括大量的 `sleep(1)`, `sleep(2)`, `sleep(4)`, `sleep(8)`, `sleep(16)` 和 `sleep(32)` 任务 (各类 `sleep` 任务数量相等), 用于评估 HTC 任务运行时间不同时的划分开销.

表 3 所示的实验结果表明, 对于两种类型的 HTC 作业, 任务划分的时间开销稳定在 2.8~4.0 s 之间. 此外, 随着 HTC 作业规模的增长, 任务划分的时间开销增长较为缓慢, 这得益于任务划分算法较低的时间复杂度 (具体为  $O(\log C)$ , 见第 4.2.4 节). 总体上看, 相较于较为庞大的 HTC 任务的运行时间, 任务划分的时间开销是可以接受的.

## 6 结论

本文提出了一个在 HPC 系统上高效管理 HTC 应用的资源管理方案 HFlow. HFlow 无缝集成集中式资源管理系统和分布式资源管理系统, 并在框架设计和作业管理机制方面保证了对传统 HPC 和 HTC 应用的良好兼容. 此外, HFlow 还采用了细粒度的任务划分算法和多级容错机制以提高任务吞吐量同时保证高容错性. HFlow 在天河-2A 超级计算机上进行了实验评估. 实验结果表明, HFlow 具有良好的应用兼容性, 在任务吞吐量方面优于主流资源管理系统和相关通量型应用资源管理方案, 同时能够实现有效的多级容错.

## 参考文献

- Shaffer T, Hazekamp N, Blomer J, et al. Solving the container explosion problem for distributed high throughput computing. In: Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020. 388–398
- Kosar T, Kola G, Livny M. A framework for self-optimizing, fault-tolerant, high performance bulk data transfers in a heterogeneous grid environment. In: Proceedings of the Second International Symposium on Parallel and Distributed Computing, 2003. 137–144
- Huerta E A, Haas R, Jha S, et al. Supporting high-performance and high-throughput computing for experimental science. *Comput Softw Big Sci*, 2019, doi: 10.1007/s41781-019-0022-7
- Houshmand S, Aggarwal S, Flood R. Next gen PCFG password cracking. *IEEE Trans Inform Forensic Secur*, 2015, 10: 1776–1791
- Carter E A. Challenges in modeling materials properties without experimental input. *Science*, 2008, 321: 800–803
- Ward L, Sivaraman G, Pauloski J G, et al. Colmena: scalable machine-learning-based steering of ensemble simulations for high performance computing. In: Proceedings of the 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), 2021. 9–20
- Casalino L, Dommer A C, Gaieb Z, et al. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. *Int J High Perform Comput Appl*, 2021, 35: 432–451
- Yang X J, Dou Y, Hu Q F. Progress and challenges in high performance computer technology. *J Comput Sci Tech*, 2006, 21: 674–681
- Yoo A B, Jette M A, Grondona M. SLURM: simple linux utility for resource management. In: Feitelson D G, Rudolph L, Schwiegelshohn U, eds. *Job Scheduling Strategies for Parallel Processing*. Lecture Notes in Computer Science, vol. 2862. Berlin-Heidelberg: Springer, 2003. 44–60
- Liao X, Xiao L, Yang C, et al. MilkyWay-2 supercomputer: system and application. *Front Comput Sci*, 2014, 8: 345–356

- 11 Ahn D H, Bass N, Chu A, et al. Flux: overcoming scheduling challenges for exascale workflows. In: Proceedings of the 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2018. 10–19
- 12 Di Natale F, Neale C, Stanton L, et al. A massively parallel infrastructure for adaptive multiscale simulations: modeling ras initiation pathway for cancer. In: Proceedings of the SC19: International Conference for High Performance Computing, Networking, Storage and Analysis, 2019. 1–16
- 13 Lawrence Livermore National Security, LLC and Flux developers. Flux learning guide. 2025. [https://flux-framework.readthedocs.io/en/latest/guides/learning\\_guide.html](https://flux-framework.readthedocs.io/en/latest/guides/learning_guide.html)
- 14 TOP500. Tianhe-2a, 2025. <https://www.top500.org/system/177999/>
- 15 Yu W, Shen Y X, Li L, et al. Teno: an efficient high-throughput computing job execution framework on Tianhe-2. In: Proceedings of the 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2018. 408–415
- 16 Jiang J, Huang D, Chen H, et al. HTDcr: a job execution framework for high-throughput computing on supercomputers. *Sci China Inf Sci*, 2025, doi: 10.1007/s11432-022-3657-3
- 17 Raicu I, Zhao Y, Dumitrescu C, et al. Falkon: a fast and light-weight task execution framework. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007. 1–12
- 18 Rho S, Kim S, Kim S, et al. Htcaas: a large-scale high-throughput computing by leveraging grids, supercomputers and cloud. In: Proceedings of Research Poster at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012
- 19 Merzky A, Turilli M, Titov M, et al. Design and performance characterization of RADICAL-pilot on leadership-class platforms. *IEEE Trans Parallel Distrib Syst*, 2022, 33: 818–829
- 20 Wu X L, Beissinger T M, Bauck S, et al. A primer on high-throughput computing for genomic selection. *Front Genet*, 2011, 2: 4
- 21 Erickson R A, Fiene M N, McCalla S G, et al. Wrangling distributed computing for high-throughput environmental science: an introduction to HTCondor. *PLoS Comput Biol*, 2018, 14: e1006468
- 22 Wu X L, Sun C, Beissinger T M, et al. Parallel Markov chain Monte Carlo-bridging the gap to high-performance Bayesian computation in animal breeding and genetics. *Genet Sel Evol*, 2012, 44: 29
- 23 Jiang M, Bu C, Zeng J, et al. Applications and challenges of high performance computing in genomics. *CCF Trans HPC*, 2021, 3: 344–352
- 24 Boldini D, Friedrich L, Kuhn D, et al. Machine learning assisted hit prioritization for high throughput screening in drug discovery. *ACS Cent Sci*, 2024, 10: 823–832
- 25 Reker D, Rybakova Y, Kirtane A R, et al. Computationally guided high-throughput design of self-assembling drug nanoparticles. *Nat Nanotechnol*, 2021, 16: 725–733
- 26 Pandey M, Fernandez M, Gentile F, et al. The transformational role of GPU computing and deep learning in drug discovery. *Nat Mach Intell*, 2022, 4: 211–221
- 27 Liu T, Lu D, Zhang H, et al. Applying high-performance computing in drug discovery and molecular simulation. *Natl Sci Rev*, 2016, 3: 49–63
- 28 Natale F D, Bhatia H, Carpenter T S, et al. A massively parallel infrastructure for adaptive multiscale simulations: modeling RAS initiation pathway for cancer. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2019
- 29 Wang K, Zhou X, Qiao K, et al. Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In: Proceedings of International Symposium on High-performance Parallel & Distributed Computing, 2015
- 30 Dai Y, Dong Y, Lu K, et al. Towards scalable resource management for supercomputers. In: Proceedings of the SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, 2022. 1–15
- 31 Lemkul J A. Gromacs tutorial protein-ligand complex. 2024. <http://www.mdtutorials.com/gmx/complex/index.html>

# HFlow: efficiently manage high-throughput applications on HPC systems

Yiqin DAI<sup>1†</sup>, Ruiibo WANG<sup>1†</sup>, Hao YUAN<sup>1</sup>, Yong DONG<sup>1\*</sup>, Juan CHEN<sup>1</sup>, Huize ZHANG<sup>1</sup>, Mingtian SHAO<sup>1</sup>, Kai LU<sup>1</sup> & Chun FAN<sup>2,3</sup>

1. *College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*

2. *Computer Center, Peking University, Beijing 100871, China*

3. *Changsha Institute for Computing and Digital Economy, Peking University, Changsha 410000, China*

\* Corresponding author. E-mail: yongdong@nudt.edu.cn

† Equal contribution

**Abstract** High-throughput computing (HTC) typically executes a vast number of small-scale, short-duration, and mutually independent computational tasks. Although high-performance computing (HPC) systems possess abundant computational resources, mainstream resource management systems and existing HTC-oriented solutions exhibit significant deficiencies in throughput, application compatibility, and fault tolerance, resulting in inefficient resource management for HTC applications on HPC systems. To address this challenge, this paper proposes HFlow—a resource management solution integrating centralized and distributed resource management architectures. HFlow achieves high application compatibility through a hybrid job management mechanism and concurrently enhances throughput and fault tolerance via a fine-grained task partitioning algorithm coupled with a multi-level fault tolerance framework. Experimental evaluations on the Tianhe-2A supercomputer demonstrate that HFlow maintains HPC application management efficiency while successfully supporting HTC resource management requirements. Specifically, HFlow delivers task throughput  $2.1\times$  to  $108.3\times$  higher than mainstream resource management systems and dedicated HTC solutions, alongside robust multi-level fault tolerance capabilities.

**Keywords** high performance computing, high throughput computing, resource management system, job scheduling, task division