重新思考超额订阅下的线程调度：协调多运行时和多进程工作负载的用户空间框架

Aleix Roca巴塞罗那超级计算中心 巴塞罗那，西班牙
arocanon@bsc.es

Vicenç Beltran巴塞罗那超级计算中心 巴塞罗那，西班牙
vbeltran@bsc.es

arXiv:2601.20435v1 [cs.DC] 28 Jan 2026

摘要

高性能计算（HPC）与人工智能（AI）的融合，正推动日益复杂的并行应用程序和工作负载不断涌现。这些工作负载通常会在同一应用程序内或多个共置作业之间组合使用多个并行运行时，从而产生调度需求，给传统操作系统调度器带来巨大压力。当系统处于超额订阅状态（就绪线程数量多于核心数量）时，操作系统调度器依靠周期性抢占来复用核心，但这往往会引入干扰，导致性能下降。本文提出：（1）用户空间调度框架（USF），一种全新的无缝进程调度框架，完全在用户空间实现。USF 允许用户无需特殊权限即可实现自己的进程调度算法。我们使用其默认的协作式策略（2）SCHED_COOP 对 USF 进行评估。该策略旨在通过仅在阻塞时切换线程来减少干扰。这种方法缓解了锁持有者抢占（LHP）、锁等待者抢占（LWP）和可扩展性崩溃等已知问题。我们通过为 GNU C 库扩展 nOS-V 运行时，实现了 USF 和 SCHED_COOP，从而无需对应用程序进行侵入式修改，即可实现多个运行时（例如 OpenMP）之间的无缝协调。评估结果显示，在超额订阅的多进程场景中，性能最高可提升 2.4 倍，包括嵌套 BLAS 工作负载、使用 LLaMA-3 进行多进程 PyTorch 推理，以及分子动力学（MD）模拟。

CCS 概念：• **软件及其工程** → **多处理 / 多道程序设计 / 多任务处理**；**调度**；特定应用开发环境。



本作品采用知识共享署名 4.0 国际许可协议授权。*PPoPP '26*，澳大利亚新南威尔士州悉尼©2026年版权归所有者/作者所有。美国计算机协会 ISBN 979-8-4007-2310-0/2026/01https://doi.org/10.1145/3774934.3786451

关键词：高性能计算、线程超额订阅、运行时系统、用户空间调度

美国计算机协会参考文献格式：
Aleix Roca 和 Vicenç Beltran。2026年。重新思考超额订阅下的线程调度：用于协调多运行时和多进程工作负载的用户空间框架。载于 第*31*届美国计算机协会 *SIGPLAN* 并行编程原理与实践年度研讨会论文集（*PPoPP '26*），*2026*年*1*月*31*日至*2*月*4*日，澳大利亚新南威尔士州悉尼。美国计算机协会，纽约州纽约，15 页。
https://doi.org/10.1145/3774934.3786451

1 引言

高性能计算、数据分析与人工智能（人工智能）的融合，正在改变高性能计算工作负载 [17, 20, 26, 34, 55]。现代高性能计算工作流不再依赖单体应用程序，而是集成了多个库和并行编程模型。这大幅增加了执行环境的复杂性：多个并行应用程序可能共存并相互作用，从而带来严峻的调度挑战。

高性能计算应用传统上依靠运行时系统，在日益复杂的计算节点上调度并行工作负载；这些节点配备了大型多核处理器、NUMA 架构和加速器 [1, 2, 7, 45, 51]。这些运行时系统负责将应用程序并行性（并行循环和任务）映射到底层硬件，同时提升性能并缓解瓶颈。运行时通常采用忙等待屏障等低延迟同步机制，并为每个核心创建一个线程，以避免超额订阅。

在超额订阅情况下，操作系统调度器通过分时策略确保公平性，线程会根据固定的时间片被抢占。尽管研究表明，在某些情况下，受控的超额订阅有益 [3, 25, 28],，但操作系统缺乏对应用程序的感知能力，通常会导致调度噪声、干扰关键路径计算并污染缓存 [12, 56]。

在高性能计算 [16, 29, 52]中，运行时组合构成了重大挑战。当同一进程中使用多个运行时系统时，它们会各自创建独立的工作线程池，从而导致线程冗余和资源

争用。此外，由于运行时假定能够独占系统访问权限，并采用激进的同步技术，抢占会严重降低其性能。因此，开发人员过去一直避免组合使用运行时——并非因为这样做没有实用价值，而是因为这通常会导致超额订阅和较差的性能。然而，随着高性能计算应用变得日益复杂，并集成异构软件栈，严格的组件隔离正变得不切实际。

同样，高性能计算应用通常以独占模式运行，以避免其他工作负载造成的干扰，并确保执行过程可预测且高效。尽管如此，许多应用在执行过程中呈现出不同程度的并行度。这些工作负载会在并行阶段和顺序阶段之间切换；由于固有的算法约束，系统资源在某些阶段会利用不足。然而，不鼓励在同一节点上自由地共同执行多个进程来"填补空档"，因为超额订阅会带来有害影响。

现有的缓解超额订阅的解决方案通常依赖资源分区 [5, 60, 64, 68] ，或为所有相关组件使用单一统一运行时 [18, 35, 47]。然而，高性能计算软件栈的高度异构性使这些解决方案难以广泛采用，因为相互竞争的应用程序接口会增加额外的复杂性。鉴于这些挑战，我们认为有必要采用一种透明的方法，以促进其在高性能计算社区中的应用。

本文提出了 USF，一种支持多进程的用户空间调度框架，可用于创建临时的用户调度策略。与类似框架不同，[18, 27, 30, 47, 73],USF 无需修改应用程序，无需特殊权限，可运行于原生 Linux 内核之上，完全在用户空间中运行，并无缝支持线程局部存储（TLS）。我们利用 USF 实现了 SCHED_COOP，这是一种无缝的协作式策略，旨在缓解高性能计算环境中的超额订阅问题。SCHED_COOP 基于以下理念：高性能计算应用受益于长时间不间断的运行，并且应尽量减少干扰。由 SCHED_COOP 调度的应用程序不会被周期性抢占，而只会在阻塞时进行切换。我们通过扩展 GNU C 库，尤其是其 pthread API，完全在用户空间中实现了 USF 和 SCHED_COOP。我们的解决方案支持任何依赖 glibc 提供的 pthread 同步 API 的应用程序，涵盖绝大多数运行时系统，例如 OpenMP（gomp、libomp）、pthreadpool（ PyTorch）或 Rust。

我们的评估表明：(1) 定制的无缝用户空间调度器有望突破当前调度的局限；(2) SCHED_COOP 能够有效缓解超额订阅问题，同时保留可用的并行度。通过最大限度地减少线程抢占和上下文切换开销，我们的策略提高了应用程序吞吐量和系统效率。由于我们的框架运行在 Linux 调度器之上，因此关键操作系统服务和非协作线程不受影响。

本文的主要贡献包括：

- USF，这是一个无缝的用户空间调度框架，允许用户创建自己的调度策略。
- SCHED_COOP，这是一种 USF 协作式调度策略，用于应对高性能计算和人工智能工作负载中的超额订阅。
- 对多运行时和多进程执行场景进行了详细评估。
- 一种避免严重性能损失的运行时嵌套实用方法。
- 超额订阅应用程序的调优指南。

2 背景

本节详细介绍 USF 和 SCHED_COOP 的主要组件；这两者通过使用 nOS-V [74]线程与任务库扩展 glibc 实现。

2.1 Linux 内核进程调度器

Linux 内核进程调度器拥有最终决定权，负责决定哪些线程在哪些核心上运行。调度器通过具有不同优先级级别的分层调度类实现 [10]。默认调度类是最早可运行虚拟截止时间优先（EEVDF）调度器 [9, 54]。该策略尝试公平地调度就绪线程，为每个线程分配一个时间片，并在超额订阅时抢占它们。EEVDF 等抢占式调度器能够保证进度，但由于抢占会在没有预先通知的情况下发生，因此可能中断关键操作，并损害对时延敏感的应用程序的性能。Linux 调度器仅向用户空间提供少量配置选项，这限制了用户影响调度决策的能力。

2.2 GNU C 库

GNU C 库 [21] 是 Linux 生态系统中最常用的库之一。glibc 提供 C 运行时、Linux 动态加载器、核心 C 应用程序接口（pthreads、输入/输出、数学库等）以及系统调用封装。其向后兼容性使其成为大多数编译器的默认选择，这些编译器默认将其链接到所有 C 应用程序。因此，glibc 是拦截进程初始化、线程创建、阻塞操作、亲和性管理，并同时透明覆盖广泛应用程序的理想汇聚点。我们在 glibc 内部实现了 USF 和 SCHED_COOP，并将其命名为 glibcv。

2.3 nOS-V

nOS-V [74] 是一个灵活的线程管理与任务调度库，支持多进程。我们将其用作 glibcv 的核心组件，详情见第 4 节。nOS-V 已在其原始论文中进行了完整说明，这里简要概述其关键细节。

nOS-V 为运行时系统提供底层服务。nOS-V 库定义了任务这一概念及其应用程序接口

提供了在工作线程中调度和执行任务的方式，这些工作线程由 nOS-V 透明地管理。概括而言，nOS-V API 支持创建任务、将任务提交给调度器、在任务完成时运行回调、暂停任务以及恢复任务。借助这些基本功能，可以构建具备依赖管理和标准功能（例如任务等待）的完整并行运行时系统，例如 NODES [39]（基于任务）或 libompv [61]（任务和 fork-join）。因此，nOS-V 可视为一种帮助构建运行时的库。

nOS-V 初始化一个工作池，并始终确保每个核心恰好固定运行一个工作线程。空闲工作线程会等待集中式调度器为其分配任务，并将任务执行至完成。不过，任务在等待用户定义的条件（例如 taskwait 等同步点）时可以自行挂起。发生这种情况时，nOS-V 会执行工作线程交换，以另一个工作线程替换挂起的工作线程，从而让核心保持繁忙。等待条件满足后，任务会重新加入调度器队列，之后分配给一个空闲工作线程。如果某个工作线程被分配了一个已绑定到其他工作线程的任务，nOS-V 会再次执行交换，使原工作线程及其任务都能在新分配的核心上恢复运行。nOS-V 提供了多种用于选择下一个待运行任务的算法；第 `taskwait` 节将详细介绍 SCHED COOP 所采用的策略。_

nOS-V 的一项关键特性是无缝多进程调度。nOS-V 在内部依靠共享内存段 [36]来保存任务、工作线程及其他相关结构。每个进程都会创建自己的任务，但由一个集中式调度器实例统一管理所有进程的任务。空闲工作线程可以接收属于任意进程的任务，但只会执行自己所属进程的任务。如果工作线程收到属于另一进程的任务，就会与该进程的一个工作线程交换，并将任务转交给它。nOS-V 会为每个进程分配可配置的时间片，但绝不会进行非自愿抢占。进程切换只会在调度点（任务挂起、结束或让出）发生。

3 SCHED COOP_

从用户的角度来看，SCHED_COOP 类似于 SCHED_RR（实时轮转调度）这类 Linux 调度策略。但与它们不同，SCHED COOP 是一种**虚拟**调度器，通过 USF 完全在用户空间中实现。

进程在启动时通过新的“chrt -c <app>”命令选项进入 SCHED COOP。子进程和线程在运行时默认继承该策略，并且只有在终止时才会退出。SCHED_COOP 线程会在固定的单核亲和性下不间断运行，直到应用程序条件（而非操作系统条件）迫使其等待。只有当线程阻塞或主动让出其核心时，另一个线程才可能在该核心上恢复运行。

尽管 SCHED_COOP 线程会贪婪地运行，但它们不会阻塞非 SCHED_COOP 进程，因为默认的操作系统调度器仍可能抢占它们。关键区别在于，SCHED_COOP 线程永远不会相互抢占。从这个意义上说，SCHED_COOP 类似于 SCHED_RR，即线程会一直运行，直到让出或阻塞。不过，与 SCHED_COOP 不同，SCHED_RR 线程无法被普通线程抢占（只能被优先级更高的 SCHED_RR 线程抢占），这可能导致系统停滞；因此，SCHED_RR 仅限 root 用户和系统用户使用。相比之下，普通用户可以使用 SCHED_COOP，而且在最坏情况下，其线程也只会阻塞其他 SCHED COOP 线程。_

SCHED_COOP 的目标是通过改善超额订阅场景中的调度决策，提高吞吐量和系统效率。因此，它并非面向所有工作负载，而是面向易发生超额订阅、且利用多个运行时系统的并行应用程序或多进程工作负载。

4 设计

本节采用自顶向下的三层方法。第4.1节从线程管理的角度说明 USF 和 SCHED_COOP 如何运作。第4.2节介绍如何通过将 glibc 与 nOS-V 耦合来实现这一行为。第4.3节深入探讨扩展后的 glibc 特性和 nOS-V API 的用法，第4.4节讨论该实现的局限性。

4.1 USF 与 SCHED_COOP 线程管理概述

USF 通过扩展标准 glibc API 无缝运行。概括来说，它对线程创建、销毁和阻塞 API 进行了调整。扩展后的 pthread 创建操作可防止线程自由运行。相反，线程会被阻塞并排队，直到有空闲核心可用。集中式调度器会监控排队等待的线程，并根据当前策略将它们派发到空闲核心。线程随后会在空闲核心上恢复执行，并持续运行，直到结束、阻塞或让出。当线程遇到阻塞操作（例如，pthread_互斥锁_lock 发生竞争）时，会通知调度器其所在核心即将空闲。作为响应，调度器会将另一个线程固定到即将空闲的核心上并恢复其运行。解除阻塞的路径（例如 pthread_互斥锁_unlock）会通知调度器有线程已准备好恢复运行，但这些线程不会立即恢复运行，而是会在调度器中排队。

SCHED_COOP 是一种用于定义简单算法的策略，用来选择下一个要运行的线程。默认情况下，所有线程的首选亲和性都设置为其上次运行所在的核心。一旦先前被阻塞的线程变为就绪状态，就会将其放入按进程、按核心划分的先进先出（FIFO）队列中。调度器会先将线程分配给与其亲和性匹配的空闲核心；如果没有可用核心，则会在同一 NUMA 域内查找，否则将其分配到其他任意 NUMA 节点。当当前进程的时间片到期时（默认值为 20 ms），调度器会在任务调度点进行评估，

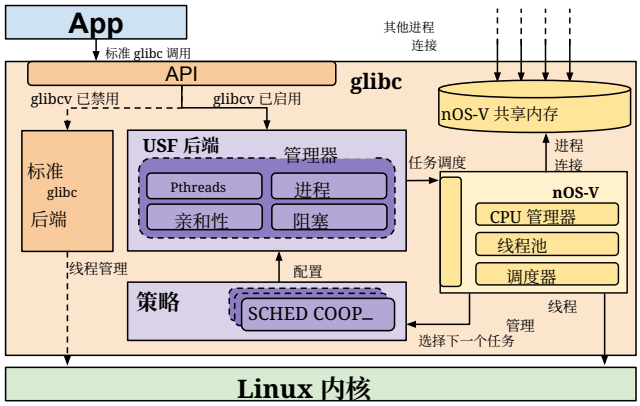

图 1. glibcv 架构图。启用后，应用程序的标准 API 调用会转发到 USF 后端，由它与 nOS-V API 衔接。nOS-V 根据所选策略调度线程。

调度器继续从下一个进程的队列中分配任务。调度逻辑是 nOS-V 的一部分，如第 2.3 节所述。

4.2 glibcv：耦合 glibc 与 nOS-V

图 1 展示了 glibcv 架构图。USF 和 SCHED_COOP 均在 glibc 中实现，并且都依赖 nOS-V 线程库来管理前文所述的大部分调度器逻辑。从概念上说，我们利用 nOS-V 将 glibc 转换为基于任务的运行时；所有 POSIX 线程都成为关联着一个任务的工作线程，大多数同步 API 都变为任务阻塞/恢复操作。由于每个任务始终绑定到同一个工作线程，因此它们与线程局部存储完全兼容。

创建 pthread 时，glibcv 会透明地将其转换为 nOS-V 工作线程，并为其分配一个映射到 pthread 入口函数的新任务。随后，任务会提交给 nOS-V 调度器，之后在具有固定单核亲和性的空闲核心上恢复运行，直到用户代码结束、阻塞或让出。用户代码结束时，任务也随之结束，nOS-V 会在该核心上调度另一个工作线程。遇到 pthread_mutex_lock 这类阻塞调用时，任务会暂停，nOS-V 也会在该核心上调度另一个工作线程。遇到 pthread_mutex_unlock 这类解除阻塞调用时，暂停的任务会重新提交给调度器。派生进程及其线程也会在 nOS-V 中注册，由它在单个多进程集中式调度器中管理所有任务。

请注意，由于我们不允许线程自由运行，而 nOS-V 会谨慎地确保每个核心只运行一个线程，因此线程必须按顺序运行，并且我们会避免 USF 线程之间发生抢占。有关 nOS-V 调度的更多详情，请参见第 2.3 节。

4.3 扩展的 API 调用

本节将说明使用 nOS-V 扩展 glibc API 的实现细节，如图 1 所示的 USF 后端模块。

4.3.1 Pthread 创建。 每当调用 pthread_create() 时，glibcv 都会用一个包装函数替换用户提供的函数指针。线程创建后，包装函数会调用 nosv_attach() API，将调用线程注册为 nOS-V 工作线程，把线程的入口函数分配给一个新的 nOS-V 任务，将该任务分配给工作线程，把任务提交给调度器，并立即阻塞当前线程。此时，glibcv 已按照提供的用户属性（亲和性除外）创建了一个 pthread，但也将其招募为 nOS-V 工作线程，这意味着它无法再自由运行。我们扩展了 pthread_t 对象，以容纳 nOS-V 任务对象及相关数据。

USF 使用线程缓存策略，与 D. Dice 和 A. Kogan [13], 提出的策略类似，以避免反复创建和销毁线程所带来的开销。当线程的用户函数结束时（或调用 pthread_exit() 并展开栈时），该线程会阻塞，并由另一个工作线程替换，而不是退出。如果另一个线程调用 pthread_join，则会屏蔽连接操作，转而将已完成的线程放入缓存。后续调用 pthread_create 时，会复用缓存中最近使用的线程，而不是创建新线程。关闭时，所有缓存中的线程都会解除阻塞、分离、销毁，并真正完成连接。分离线程时会调用 nosv_detach()，该函数会将线程从 nOS-V 注销，并终止其任务。

4.3.2 亲和性管理。在 USF 下，更改线程亲和性（例如，调用 pthread_setaffinity_np()）的操作会被视为提示。这是为了避免干扰 nOS-V 精细的线程放置。在这种情况下，glibcv 会将用户亲和性存储在 pthread_t 对象中，但不会更改它。为保持应用程序兼容性，在通过 pthread_getaffinity_np() 等应用程序接口查询时，glibcv 会返回先前存储的亲和性，而非实际亲和性。出于同样的目的，我们还扩展了 thesyscall() 系统调用入口点，以拦截 sched_setaffinity；由于其语义与 glibc 包装函数不同，该调用通常会被直接调用。我们还使用哈希表跟踪 tid-任务映射，这是调用 sched_set/getaffinity 所必需的。不过，如有需要，创建遵循用户亲和性的 USF 策略也很容易。

4.3.3 分叉与进程初始化。 Glibcv 可以管理多个进程。在进程启动时，如果 USF_ENABLE环境变量已设置，glibcv 会在调用任何构造函数之前初始化 nOS-V。如第 2.3 节所述，nOS-V 会连接到任何已存在的共享内存段，并注册新进程。随后，glibcv 会转换当前的

清单 1。 Glibcv 互斥锁加锁/解锁扩展。当互斥锁发生竞争时，线程会先被放入由自旋锁保护的每互斥锁 FIFO 等待队列中，然后在 nOS-V 中阻塞。解锁时，如果队列非空，则会取出一个线程并提交给调度器。

```
intpthread_mutex_lock(pthread_mutex_t *mutex){nosv_task_ttask =pthread_self()->task;pthread_spin_lock(&mutex->spinlock);if(pthread_mutex_trylock(mutex)){//竞争队列_add_tail(&mutex->queue,task);pthread_spin_unlock(&mutex->spinlock);nosv_pause();}else{pthread_spin_unlock(&mutex->spinlock);}return0;}intpthread_mutex_unlock(pthread_mutex_t *mutex){pthread_spin_lock(&mutex->spinlock);list_head_twait =&mutex->queue;nosv_task_ttask =queue_pop_head(wait);if(!task)//标准互斥锁解锁__pthread_mutex_unlock(mutex);pthread_spin_unlock(&mutex->spinlock);if(task)nosv_submit(task);return0;}
```

通过 nosv_attach() 将主线程转换为附加了任务的 nOS-V 工作线程。当进程终止或调用 exec 时，glibcv 会调用 nosv_shutdown() 注销该进程，并关闭其工作线程。对于 exec，操作完成后，新应用程序也会找到 USF_ENABLE 环境变量，并再次将进程注册到 nOS-V 中。

4.3.4 阻塞 API。我们扩展了互斥锁、条件变量、屏障、信号量、睡眠、让出和轮询（poll、epoll）应用程序接口，以检测阻塞操作。清单1 展示了互斥锁的伪代码。我们为`pthread_mutex_t`添加了 FIFO 等待队列，并使用自旋锁对其进行保护。发生竞争时，`pthread_mutex_lock()` 会将调用线程对应的 nOS-V 任务（每个 pthread 都关联一个）加入队列，并通过`nosv_pause()`将其阻塞，从而触发工作线程交换。调用`pthread_mutex_unlock`时，如果存在竞争中的任务，则将其中一个提交给 nOS-V 调度器（nosv_submit()），并将所有权转移给该任务；只有在没有任务排队时，才会完全释放互斥锁。

其他应用程序接口遵循相同的模式：竞争中的任务由标准 glibc 对象跟踪，并通过 nOS-V 调用进行管理，以实现阻塞、恢复、让出或等待。poll 和 epoll 的定时变体会定期使用原始应用程序接口执行非阻塞事件检查，然后调用 nosv_waitfor()，在循环中进行短暂的定时等待（默认 5 毫秒），直至用户设定的超时到期或事件发生。nosv_waitfor() 也会触发工作线程交换，但提供的超时时间一到，任务就会自动重新提交。

4.4 局限性

USF 对核心的高效利用，部分取决于其拦截等待操作（例如阻塞操作）的能力。如果等待操作未被拦截，nOS-V 就无法检测到空闲核心，也无法在该操作完成前在其上运行其他任务。对阻塞操作进行插桩很简单，因为这些操作通常会使用标准 glibc API 或内核系统调用封装器。然而，一些应用程序或运行时依赖忙等待屏障来实现低延迟同步。在 glibc 层面为这些情况插桩较为复杂，但若不这样做，可能会阻碍进度并导致应用程序死锁。当等待线程数量超过核心数量，或多个屏障同时处于活动状态时，就可能发生这种情况。相比之下，抢占式调度器会将这一问题转化为性能问题，因为在超额订阅的系统中，它们不依赖调度点来保证进度。正如我们将在第 5.2 节进一步详述的那样，我们针对这些情况的解决方案是手动调整忙等待屏障，使其每经过若干次忙等待迭代便执行一次让出操作（例如`sched_yield()`），这不仅有利于 SCHED_COOP 之类的策略，也有利于超额订阅情况下的抢占式调度器（参见第5.3节）。

USF 支持任何依赖 glibc 同步应用程序接口的应用程序。大多数运行时通常会提供兼容 POSIX 的后端，同时也提供适用于 Windows、macOS 或 Linux 的平台专用替代方案（例如自定义的基于 futex 的实现）。通过 glibc 的 futex 包装器扩展 USF，以支持基于 futex 的后端，是我们计划开展的未来工作之一。

USF 策略继承了 nOS-V 的安全性限制。nOS-V 通过共享内存段共享状态。连接到同一 nOS-V 实例的所有进程都可以访问这段共享内存，因此进程之间可能会暴露敏感信息。出于这个原因，我们只允许同一用户和组的进程连接到同一共享内存。未来，我们将研究如何加强 nOS-V 与 Linux 内核的集成，以克服这一限制。

由于 glibcv 扩展了静态分配的标准 glibc 对象（例如 pthread_互斥锁_t），且其大小是在编译时计算的，因此所有应用程序及其依赖关系都必须针对 glibcv 重新构建。幸运的是，构建

表 1. 评测机器规格。

机器	Marenostrum 5
内存	256GiB
网络	InfiniBand（100Gb/s）
CPU型号	英特尔 Sapphire Rapids 8480+
GPU	-
CPU架构	x86-64_
CPU 核心	56x2
L1, L2, L3	80KB, 2MB, 105 MB

Nix 等系统 [15] 能够自动化整个系统的构建，这大大简化了使用具有复杂依赖关系的应用程序（例如 PyTorch）测试 glibcv 的工作。

5 评估

在本节中，我们将在复杂程度逐步提高的场景中评估 USF 和 SCHED_COOP。我们首先概述评估方法，该方法遵循自底向上的方法。我们的第一个假设是，用户空间调度器结合对软件栈的手动调整，可以提升超额订阅应用程序的性能。第二个假设是，无需侵入式修改，也能轻松实现类似的性能提升。为验证这些假设，第5.1 节介绍了评估场景——嵌套运行时和多进程协同执行——并说明实验设置。接着，第5.2 节讨论干扰因素，通过统一运行时等待策略并禁用线程绑定，确保比较公平。第5.3 节通过受控的矩阵乘法基准测试验证这两个假设，并比较手动调整与无缝的 USF 方法。第5.4 节将分析扩展到采用多种运行时组合的乔列斯基分解基准测试，突出异构软件栈带来的挑战。接下来，第5.5 节和第5.6 节分别展示 USF 在实际人工智能微服务和分子动力学场景中的应用，彰显其在真实多进程环境中的优势。

5.1 评估场景与设置

我们重点研究单节点内两种具有代表性的超额订阅场景：嵌套运行时，即外层运行时中的线程调用内层运行时的并行区域，导致活动线程数量迅速增加（例如，某个应用程序有 n 个 oneTBB 线程，每个线程都调用一个会生成含有 n 个线程的 OpenMP 区域的 BLAS例程，最多可产生 n^2 个线程）；以及多进程工作负载，即多个并行应用程序交错执行串行阶段和并行阶段，以最大限度地利用资源。我们的实验比较了标准 Linux 调度器（基线）与 SCHED_COOP 下的性能。我们使用生产系统

如表 1所述，该系统不允许修改内核，这体现了我们透明的用户空间方案的价值。我们的软件环境包括：Linux 5.14、glibc 2.40-6、nixpkgs 25.05、clang 20、gcc 14.2.1、OpenBLAS 0.3。28 [22], BLIS 1。1 [53, 59], PyTorch 2。6 [43], Transformers 4.51。3 [63], oneTBB 2020。3 [49], MPICH 4.3。0 [23], LAMMPS29Aug2024_更新 2 [46, 57], DeePMD-kit 3.1。1 [62, 69, 70],nOS-V 3。2 [74], Nanos6 4。2 [6],以及 NODES 1。3 [39]。

5.2 干扰因素

为了进行公平比较，我们首先为运行时和其他库选取合理的参数设置，以免它们干扰有关 Linux 进程调度器与 SCHED_COOP 之间差异的结果。之所以需要这些调整，是因为大多数运行时和部分并行库并非为在同一地址空间中共存而设计。我们测试了采用和未采用这些调整的应用程序，最终决定将其纳入实验，因为这能提升基线调度器和 SCHED COOP 调度器的性能。_

大多数运行时都支持等待策略，例如 OpenMP（OMP_WAIT_POLICY）和 OmpSs-2 [44]。等待策略定义了工作线程在无任务可做时的行为。默认情况下，这两种运行时都采用混合等待策略：工作线程会先进行一段时间的主动等待，然后才采取开销较大的操作系统阻塞/恢复路径。然而，主动等待任务的运行时工作线程可能会浪费中央处理器周期，并延迟另一个有就绪任务待执行的运行时工作线程的执行。因此，我们在所有实验中都将运行时配置为采用类似的"被动"等待策略（超额订阅时推荐使用），使线程立即进入操作系统阻塞路径，而非主动等待更多任务。线程绑定是另一项功能，允许用户配置工作线程是否绑定以及绑定到哪些系统核心（例如，OpenMP 中的 OMP_PLACES）。然而，我们早期的评估表明，多个运行时或超额订阅的进程若将工作线程绑定到相互重叠的 CPU 集合，就会彼此干扰，严重损害性能。所有测试均通过配置运行时跳过线程绑定来避免这些问题。

将分析范围扩展到运行时系统之外后，我们观察到一种反复出现的模式：一些库会实现自定义的忙等待屏障，而不依赖标准 API。在活跃线程数不超过可用核心数的系统中，忙等待屏障可以有效地让线程短暂停顿。然而，在超额订阅的系统中，调度噪声会显著延迟屏障完成。单个被抢占线程就可能阻塞整个线程组；更糟的是，忙等待线程会持续消耗中央处理器周期，直到时间片耗尽——这可能反而延迟释放屏障所需的线程。当多个忙等待屏障同时运行时，这个问题会进一步加剧

清单 2. 通过运行时嵌套并行化的简化矩阵乘法代码：OmpSs-2（Nanos6）和 BLIS（OpenMP）。问题被划分为 NB 个块，每个块的大小为 TS。“oss task” pragma 会创建一个具有指定依赖关系的任务，而“oss taskwait”会阻塞执行，直到所有先前的任务都已完成。

```
for (k = 0; k < NB; k++)
  for (i = 0; i < NB; i++)
    for (j = 0; j < NB; j++)
      #pragma oss task inout(C[i][j]) \
        in(A[i][k]) in(B[k][j])
      cblas_dgemm(TS,
        A[i][k], B[k][j], C[i][j]);
#pragma oss taskwait
```

同时，这也是并行基础线性代数子程序库内核并发执行时经常遇到的场景。在评估中，我们遵循一种常见模式：在忙等待屏障中让出，以便其他线程继续执行。具体来说，我们只需修改 OpenBLAS、BLIS 和 MPICH 中的一行代码，加入 `sched_yield()` 系统调用（详见第5.3节）。

嵌套并行运行时的一个副作用是，生成的内核粒度可能过小。嵌套并行循环或类似结构会被细分为数量呈指数增长的微内核。在运行时系统中，细粒度内核本身就构成了挑战 [4, 8, 40, 66]，因为运行时和系统开销会变得至关重要。不过，在本研究中，我们重点分析超额订阅，并假设工作负载足够大。

5.3 手动与无缝调度比较

在本节中，我们使用一个简单的矩阵乘法（矩阵乘法）基准测试来评估第 5.1 节中提出的两个假设。我们的矩阵乘法（参见清单 2）是嵌套运行时场景的一个示例：我们采用阻塞方式将问题划分为 OmpSs-2 任务（外层运行时），并使用由 LLVM 的 OpenMP 并行化的 BLIS 处理每个分块（内层运行时）。每个 OmpSs-2 任务由一个 Nanos6 线程执行，并且每个任务都会在 BLAS 库中开启一个 OpenMP 并行区域。

我们将所有输入矩阵的大小固定为 32768×32768，并通过改变任务大小（外层运行时的并行度）和 OpenMP 线程数（内层运行时的并行度），评估基准测试对超额订阅的响应。例如，在我们的 56 核系统中，使用 2 个 OpenMP 线程和 16384 的任务大小（同时最多有 $(32768/16384)^2 = 4$ 个并行任务），此时共有 $2 \times 4 = 8$ 个线程同时处于运行状态，而系统大部分时间都未得到充分利用。使用 56 个 OpenMP 线程和大小为 512 的任务时，我们有 229376 个繁忙线程，系统处于严重超额订阅状态。我们将每个内核放入循环中运行，持续时间至少为

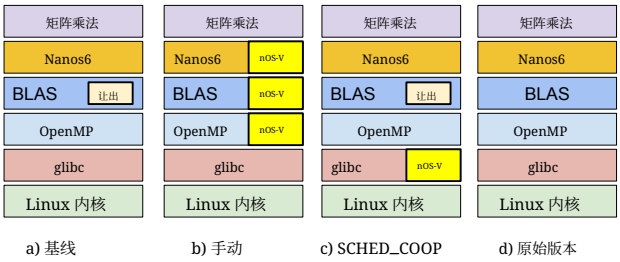

图 2。评估的矩阵乘法软件栈。a）采用让出的基线。b）手动集成 nOS-V。c）无缝集成 nOS-V。d）未修改（不让出）。

60 秒，并测量以 $\frac{size}{seconds} \cdot \frac{loops}{} \cdot 10^{-6}$ 计算得出的性能指标 MOPS/s。

我们比较了四种矩阵乘法版本，如图 2 所示，按改动幅度从小到大排列：原始版本是未经修改的实现。基线与原始版本相同，但在屏障处加入了让出机制（第 5.2 节），并作为我们的参照点。SCHED_COOP 使用与基线相同的软件栈，但将 glibc 替换为 glibcv，以便通过 nOS-V 启用我们的调度器。手动版本实现了与 SCHED_COOP 相同的调度策略，但它并未通过 glibcv 透明地启用该策略，而是在 Nanos6、OpenMP 和 BLAS 库中显式调用 nOS-V 线程库来手动集成。手动配置旨在通过不受无缝方案约束、并在每个库层级加入少量临时优化来最大化收益，从而确定 glibcv 性能的上限。从本质上说，手动配置通过 `nosv_attach` 将不同层级创建的 pthread 连接起来，并使用 nOS-V 阻塞原语替代 pthread 原语。

图 3展示了实验结果。对于每个评估过的矩阵乘法版本，热图颜色有助于我们大致区分出三个区域：左下三角区域对应资源利用不足的配置（线程数少于核心数，且任务粒度较粗）。右上三角区域表示因指数级碎片化而产生的过细粒度块（线程数多于核心数，且任务粒度较细）。中间的对角区域对应粒度合理的超额订阅配置。我们认为上方和下方的三角区域设计不佳：暴露的并行度要么不足，要么过多。因此，我们关注的是中间区域，其中包含大量有效配置。请注意，只启用单个运行时的配置，即 (1-32768, *) 和 (*, 1)，对应的是不涉及运行时组合的场景。

根据我们的第一个假设，我们分别比较基线与手动配置（见图3 a 和 b），以证明 nOS-V 用户空间调度器能够提升超额订阅应用程序的性能。乍看之下，热图着色显示，手动版本能够扩大相较于基线性能更高的区域，尤其是在中间区域。手动配置下的性能

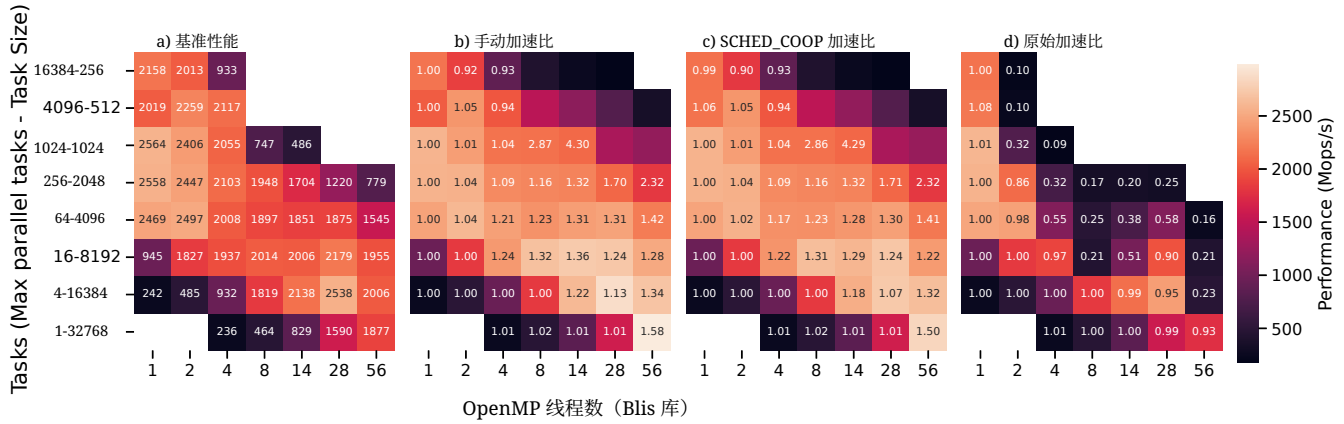

图 3. 使用两个嵌套运行时对矩阵乘法进行评估。图 a) 中的数字表示性能（越高越好），而图 b)、c) 和 d) 显示相对于图 a) 的逐元素加速比。所有热图中的颜色均表示性能，而非加速比。白色方块表示实验超时（>15 分钟）。没有加速比的方块表示由于超时而缺少基线参考。

加速比细节显示，根据配置不同，性能提升幅度为 1% 至 36 %。因此，我们可以得出结论：第一个假设成立。请注意，尽管某些配置下的加速比最高可达 4 倍，但我们认为这些结果并不重要，因为与获得的最佳结果相比，其性能较低。

根据我们的第二个假设，我们将基线与 SCHED_COOP（分别参见图 3 a) 和 c)），以证明无缝方案在超额订阅时也能提升性能。采用相同的方法，通过比较两张热图的颜色渐变，我们可以观察到类似的性能提升，加速比介于 2% 至 28 % 之间。因此，我们也认为第二个假设得到了验证。此外，最佳的 SCHED_COOP 配置（28,4-16384）涉及两个运行时，相比使用单个运行时的最佳基线配置（1, 1024-1024），性能提升了近 9.8%。此外，最佳手动配置（28,4-16384）相比最佳基线提升了 11.77%。这表明，如果采用适当的调度策略，在同一地址空间中混合使用运行时可能有益。

在本实验中，使用 nOS-V 版本的矩阵乘法之所以能提升性能，主要是因为基础线性代数子程序库的忙等屏障管理方式。实际上，即使 sched_让出通常用于让出当前核心，Linux 也可能不会立即让出，而是等待线程的时间片到期 [3]。具体情况取决于所使用的 Linux 版本，但这种用法仍然有益，因为线程会尽可能早地让出，而不是等到下一次时钟中断。作为对比，请参阅图 3 d) 中原始版本的结果。该版本与基线相同，但使用原始的忙等屏障，不使用 sched_让出。可以看到，许多配置下性能明显下降，尤其是超额订阅程度最高的情况，因为大部分时间都花在忙等上。相比之下，矩阵乘法 SCHED_COOP 版本总是让出，因为原始的 sched_yield 在内部使用 OS-V 让出 API。在这种场景下，尽早让出是有益的，因为在超额订阅严重时，这能让线程更早退出屏障。

总之，Manual 和 SCHED_COOP 都能扩大可实现更高性能的配置范围，而 SCHED_COOP 的优势在于能够透明地完成这一点。USF 无需内核修改即可实现定制的用户空间调度策略，例如 SCHED_COOP。这有望有效释放运行时组合和超额订阅场景的潜力；迄今为止，由于这些场景通常会带来性能损失，人们往往避而不用。它还简化了对输入配置空间的探索。用户无需在不同运行时之间寻找合适的线程平衡，而可以专注于寻找能够展现足够并行度、同时保持任务粒度足够粗的配置。

5.4 运行时组合

在本节中，我们展示即使使用相对简单的基准测试，运行时组合也会迅速变得多么复杂。为此，这次我们采用乔列斯基分解基准测试，并组合使用多种不同的运行时实现，包括 GNU 的 gomp、LLVM 的 libomp 和英特尔的 tbb。与矩阵乘法类似，我们的乔列斯基首先使用外层运行时将问题划分为多个块，然后针对每个块调用 BLAS API，由其进一步划分并使用内层运行时（基于 OpenMP 或 pthread）并行计算内核。为简洁起见，我们仅展示部分相关基线和 SCHED_COOP 配置的结果。我们将输入规模和任务规模分别固定为 32768×32768 和 1024。结果见表2。SCHED_COOP 的性能优于

表 2。结合多个运行时、针对三种并行度的乔列斯基评估结果。对于每种并行度，我们报告基线的性能（单位为百万次操作/秒，越高越好）以及 SCHED_COOP 的加速比。低并行度使用 8x8 个线程（每个核心 1.14 个线程），中并行度使用 14x14（每个核心 3.5 个线程），高并行度使用 28x28 （每个核心 14 个线程）。gnu 表示 GNU 的 OpenMP，llvm 表示 LLVM 的 OpenMP，pth 表示 pthread BLAS 后端，opb 表示 OpenBLAS。

Out	Inn	BLAS	Mild	中等	High
gnu	llvm	opb	1468，1.1倍	580，2.36倍	347，3.14倍
tbb	llvm	opb	1619，1.1x	602，2.47x	363，3.22x
tbb	gnu	blis	2004，1.06x	694，3.23x	358，4.28x
tbb	pth	blis	1590, 1.34x	356, 6.28x	103, 14.4x
gnu	pth	blis	1413, 1.33x	254, 6.86x	86, 14.7x

所有情况下的基线，尤其是在 Pthreads 情况下。这是因为 BLIS pthread 后端不会复用线程，而是在每次调用 dgemm 时创建并销毁线程。相比之下，USF 会以透明方式缓存线程并抑制操作系统噪声（见第 4.3.1 节）。我们发现，pthread 缓存使基准 SCHED_COOP 性能提升了约 4 倍。在其他情况下，pthread 缓存没有效果，因为运行时能够高效地复用 pthread。请注意，外层的"gnu"和内层的"llvm"使用静态构建，以防止符号冲突。

关键结论是，即使对于简单的应用程序，我们也可能面对运行时、实现方式和编译选项的多种组合，这会迅速使处理超额订阅的策略变得复杂。手动协调所有这些情况，需要投入大量精力去理解和适配每一种可能性。相比之下，SCHED_COOP 能够以透明方式管理所有这些情况，同时让用户保有控制权。

5.5 微服务案例研究

本节研究一个 Python 多进程人工智能推理基准测试。Python 在人工智能开发中占据主导地位，但其全局解释器锁（GIL）会使线程串行执行，因此开发者转而使用多个进程搭配原生并行库。这种常见模式有助于我们展示 SCHED COOP 在超额订阅的多-_进程环境中的表现，并将其与标准 Linux 调度器下的各种资源分区方案进行比较。我们重点关注基于 CPU 的推理；由于 CPU 普遍可用、灵活且经济高效，这一方向正受到越来越多的研究关注 [31, 42, 67, 71, 72] 。

我们的基准测试模拟客户端按照泊松分布定期发送请求，由专用节点上的四个 Python 服务器进程处理。网关进程接收请求、模拟规划逻辑，并将请求并行转发给三个推理服务器，同时阻塞等待所有响应返回。每个服务器运行一个不同的模型：Meta 的 LLaMA 3.2 (1B)[58],OpenAI 的 GPT-2 (124M)[48],以及经过微调的 Meta RoBERTa-large (355M) [24] ，用于情感分析。所有模型都运行在 PyTorch 上，PyTorch 使用高效的 C++库和 OpenBLAS 与 OpenMP 来绕过 GIL 的限制。因此，每个用户请求都会在每个进程中创建一个线程（总共四个），其中三个线程会生成 pthread 线程团队，以进行并行推理计算。在请求频率较高时，重叠的请求会导致系统超额订阅。

我们通过逐步提高请求频率来评估基准测试的行为。每次执行都会生成 28 个请求，每个请求包含 128 个词元（~168 个词），并模拟每个请求包含 8 个批次的分组。每个模型最多输出 32 个词元（~42 个词）。我们不限制处理请求的外层线程数量，但会固定用于计算每次推理的内层 BLAS 线程数。我们通过对单个请求运行独立的强可扩展性测试，计算出每个模型的理想内层线程数。具体而言，LLaMA 使用 28 个核心时耗时 5.4 秒，GPT-2 使用 8 个核心时耗时 1.8 秒，RoBERTa 使用 8 个核心时耗时 1.2 秒。

我们将 SCHED COOP 下运行基准测试的结果与四种基线场景进行比较。bl-none 不使用资源分区。bl-eq 将核心平均分配给所有 Python 推理服务器，但网关除外，网关使用 2 个核心。bl-opt 根据先前获得的可扩展性结果采用优化分区方案，将核心按比例分配给负载最重的服务器，并为网关服务器预留两个核心。该优化分区方案将节点的 112 个核心划分为：LLaMA 使用 71 个核心（64%），GPT-2 使用 23 个核心（21%），RoBERTa 使用 16 个核心（14%）。请注意，即使采用这种资源分区，随着请求频率增加，各分区内部仍会出现超额订阅。不过，分区有助于限制总体超额订阅。bl-none-seq 不使用分区，而是通过顺序计算推理（不使用 OpenMP 并行）来避免超额订阅。

所有测试中，网关服务器的 nice 优先级设为 0（高优先级），而推理服务器的优先级设为 20（较低优先级），以便尽可能快速地处理并响应用户请求。SCHED_COOP 无需设置 nice 优先级，但我们仍设置了这些优先级，以考虑其他系统进程造成的干扰。

图 4 （上）展示了 SCHED_COOP 和三个基线的吞吐量与延迟。正如预期，`bl-eq` 中的静态分区因各分区之间负载不均衡，在吞吐量和延迟方面的表现都最差。`bl-opt`在一定程度上缓解了负载不均衡；不过，随着请求速率提高，LLaMA 服务器会成为瓶颈，导致整个后端停滞。`bl-none` 将完全控制权交给 Linux 调度器，表现出色，但最终仍会因超额订阅加剧的同步噪声而崩溃。`bl-none-seq`由于没有超额订阅，它在所有请求速率下都能保持稳定的延迟，并在最高请求速率下实现较高的吞吐量

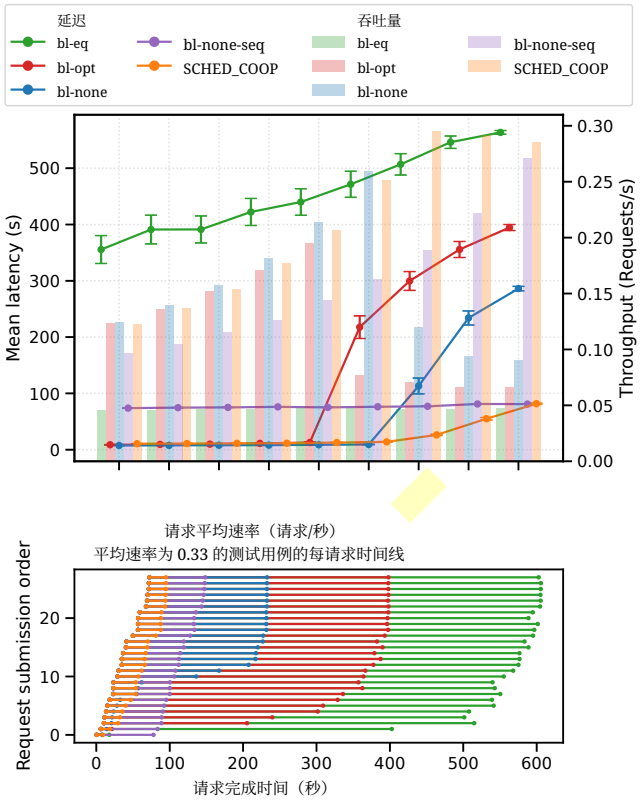

图 4. 对智能体微服务基准测试的评估。上图展示了请求频率增加时的延迟和吞吐量。下图详细展示了速率为 0.33 的测试用例中每个请求的开始和结束时间。

噪声。不过，由于服务器缺少 OpenMP 并行度，单次推理时间更长——尤其是在较低请求速率下，此时大多数核心处于空闲状态。最后，SCHED_COOP在整个请求速率范围内都实现了低延迟和持续的吞吐量。图 4（下）展示了平均请求速率为 0.33 时各请求的延迟，此时bl-none发生崩溃。随着正在处理的请求不断累积、超额订阅程度不断加剧，Linux 公平调度器仍会将 CPU 时间平均分配给所有线程。这使各请求的处理进度趋于一致，包括大多数 OpenMP 忙等待屏障及其产生的相关噪声；这些情况会持续不变，直到所有请求几乎同时完成。相比之下，SCHED_COOP 能够更快地处理请求并提前完成，从而避免让出处理器的线程过度累积所产生的噪声；与bl-none相比，性能最高可提升至 2.4 倍。

顺带一提，人们可能会合理地认为，非抢占式调度器可能更倾向于吞吐量而非延迟。然而，在这种情况下，BLAS 库持续进行的同步充当了调度点，使所有参与的进程即使没有非自愿抢占也能继续推进。

5.6 LAMMPS 与 DeePMD-kit 案例研究

本节研究将 LAMMPS 分子动力学模拟器与 DeePMD-kit 相结合的方案。DeePMD-kit 是一种深度学习软件，用于模拟原子间势能和力场。密度泛函理论（DFT）等经典量子力学计算能够提供高度准确的结果，但在大规模模拟中计算成本过高。利用这些第一性原理方法产生的数据训练的专用人工智能模型，只需极低的计算成本便可达到相当的准确度 [37, 62]。LAMMPS 可以使用 MPI 将物理模拟域划分为多个空间区域，每个 MPI 秩计算其分配区域内原子的力和能量，并与相邻秩交换边界数据。在每个秩内，DeePMD-kit 使用 PyTorch 进行推理，而 PyTorch 在内部利用采用 OpenMP 并行化的 OpenBLAS 执行其计算内核。

模拟中特定的原子空间分布和 MPI 划分可能会变得不均衡，因为移动的粒子会跨越秩间边界，并在运行时动态地重新分配各秩的工作负载。因此，要高效地将进程和线程分配到核心上以最大限度地利用资源，并非易事。不过，执行多个独立模拟或系综以探索不同路径 [14, 41]，是一种常见做法，这些模拟或系综通常会依次运行，或作为不同的作业运行。然而，系综可能会增加一个我们可以利用的并行度层级，通过用一个模拟填补另一个模拟的空隙来提升性能。但与此同时，在计算阶段负载较重时，这也会进一步加剧系综、进程和线程之间发生调度干扰的风险。

我们分析了采用 MPI 与 OpenMP 混合并行的双 LAMMPS 系综设置中的多种执行和超额订阅场景。在独占模式下，系综依次运行，每个系综使用 56 个 MPI 进程，每个进程配备 2 个 OpenMP 线程，并绑定到互不重叠的核心集合。协同执行场景与此类似，但系综并行运行。在协同定位模式下，我们通过将 MPI 进程数减少到 28 个（每个秩仍保留 2 个 OpenMP 线程）来避免超额订阅，并将每个秩绑定到互不重叠的核心集合。在 SCHED_COOP 模式下，系综由我们的调度器管理，每个系综使用 56 个 MPI 进程和 2 个 OpenMP 线程。除独占模式外，所有配置都有两种变体：“套接字”变体中，每个系综的进程均匀分布在机器的两个套接字上（一半位于一个套接字，另一半位于另一个套接字）。而在“节点”变体中，每个系综被限制在不同的套接字上。

我们训练了一个标准 DeePMD-kit *se e2 a* 模型，其_ _描述符大小为 [25, 50, 100]，，拟合网络为 [240,240, 240]，，截断半径为 6 Å。该模拟生成了 20 K 个 CH_4分子（每个分子含 5 个原子，共计 100 K 个原子），这些分子创建于边长为 $292.4 \times 292.4 \times 292.4$Å 的盒子中，并分布在一组 14

沿x轴交错排列的稠密区域和稀疏区域中，其中稠密区域包含90% 的原子，而稀疏区域包含其余 10%。三维模拟盒沿x轴在各秩之间划分，从而形成不平衡的环境。我们将温度保持在较低的 5 开尔文，以使原子速度保持较低。每次模拟运行 100 步。

图 5a 展示了所有配置的评估结果，图 5b 比较了评估节点在执行每种配置期间观测到的总内存带宽（读写）。在独占模式下，单个系综的性能最高可达106K原子步每秒，而在其他配置中，最多只能达到60K原子步每秒。这是因为不存在与其他运行时的干扰，每个排名都可以使用全部内存带宽。然而，如图 5b 所示，单个系综在任何时刻都无法利用全部带宽。此外，每个系综的初始化是顺序进行的（表现为开始阶段和中间阶段的带宽谷值），其开销必须由每个系综分别承担；在其他场景中，系综初始化可以并行完成，可利用更多并行度来发挥机器资源的作用，并且所有系综的聚合原子步每秒性能超过了独占模式。

不过，这两种共置场景都存在严重的负载不均衡，导致一半的缓存和核心未得到充分利用。此外，与独占场景相比，需要保留在缓存中的数据量翻了一倍，因为两个系综是并行运行的。套接字方案的性能最差，因为它还需要承担处理器间通信的开销。

协同执行场景能够充分利用所有核心和缓存，因为：1）每个系综总共提供的线程数与核心数相同；2）线程可以在核心之间迁移。然而，与此同时，超额订阅以及 MPICH 和 OpenBLAS 的忙等屏障也限制了它们的优势。

最后，SCHED_COOP 方案不存在前述限制，却具备所有优点：并行运行两个系综可提供更多并行工作负载，以弥补负载不均衡；线程可以迁移，从而利用所有资源；忙等待也不再使核心停滞。因此，SCHED_COOP 的 Katoms-setp/s 和内存带宽利用率均达到最高；与协同执行的套接字场景相比，加速比最高可达 4%。不过，由于 USF 不支持 I/O 系统调用，大多数阻塞式 MPI 通信会使核心停滞，直至通信完成。未来，我们将研究如何集成这些操作，以进一步改善这些场景。

5.7 要点

根据我们的评估，我们总结出以下经验：（1）组合使用多个运行时的应用程序通常难以调优和调试。在超额订阅情况下，这些应用程序只需稍作代码调整，避免使用忙等待屏障，就能获得显著收益。（2）当参与共同执行的应用程序具有不同程度的并行性时，共同执行有助于最大限度地利用资源；但在超额订阅情况下，抢占式调度器的负面影响很快就会削弱这一优势。（3）SCHED_COOP 可以管理这两种场景；与类似方案不同，它能够透明运行，无需提升权限，并且完全兼容线程局部存储。（4）USF 策略兼容任何依赖 glibc 提供的标准同步 API 的应用程序，但在粒度设置合理的超额订阅场景中，SCHED_COOP 尤为有用。（5）协作式用户空间调度器能够发挥现代应用程序在超额订阅场景下的潜力；此前，由于性能下降，这类场景通常会被规避。（6）其主要局限在于自定义忙等待屏障。不过，我们已经证明，这类屏障很容易识别和修复。

6 相关工作

已有多项研究探讨了如何协调运行时与内核调度器所面临的挑战 [32, 33]。当多个不同的运行时在单个进程中协同运行并导致系统超额订阅时，这一挑战会更加严峻 [65], , 例如 Python 及其高度模块化特性 [35]。这类方法通常会限制线程数量、强制使用单一运行时，或扩展运行时以便彼此协调。

可以通过在同一节点上共同执行多个进程，利用应用程序负载不均衡的特点，用一个进程填补另一个进程的空闲间隙。这种方法已应用于云弹性 [25]„ 以解决操作系统可扩展性和忙等待等问题，但未考虑关键区域中的抢占——这对并行运行时尤其成问题。在高性能计算中，超额订阅 MPI 进程有助于实现通信与计算重叠 [3], , 但管理多个运行时会使这一做法更难实现。另一项研究 [28] 表明，有限度的超额订阅有利于粗粒度工作负载，但在现代众核系统中，系统级抢占式调度器导致同步中断的风险更高 [12, 56]。一种常见的替代方案是资源分区：Arachne [47] 和 DLB [18] 可在运行时动态地为进程分配核心，效果良好，但要使用其应用程序接口，需要对应用程序或运行时进行大量修改。

近期，随着高性能计算工作负载开始使用作为独立进程运行的外部人工智能（AI）服务，共同执行应用程序再次受到关注 [38]（AI-out-HPC）。然而，要找到一种统一的方案来协调所有涉及的组件和领域颇具挑战。相较之下，在系统级调度器层面解决这一问题，能够提供透明的方案，并有望大幅简化两个领域的集成。

在这方面，Linux 社区通过可扩展调度器类（sched_ext）推进了用户空间调度 [11], , 该机制允许将调度器实现为 eBPF [19] 程序。然而，调度决策仅限于预定义的内核钩子，eBPF 代码无法使用

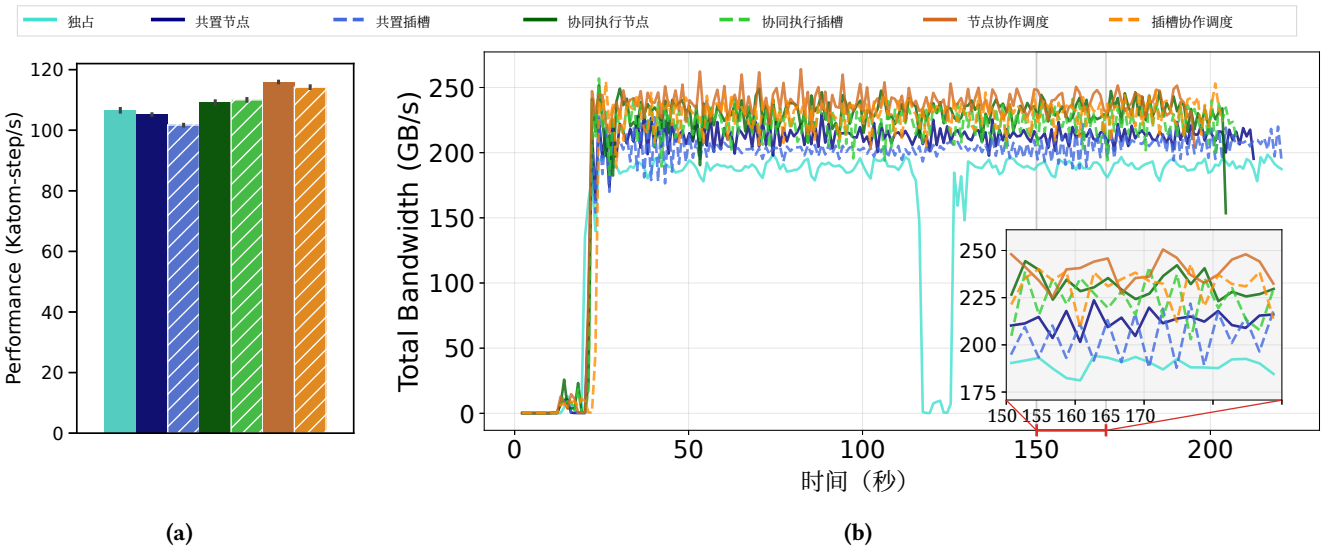

图 5. 在多种配置下，使用两个系综对 LAMMPS 和 DeePMD-Kit 进行评估。除共置版本将 MPI 秩限制为 28 以避免超额订阅外，所有系综均各使用 56 个 MPI 和 2 个 OpenMP。图 a) 展示性能。图 b) 展示测得的主内存读写总带宽。图 b) 中的内嵌图展示了从第 150 秒开始、时长为 20 秒的放大区域。各场景的平均带宽（GB/s）为：独占 165.36，节点协同部署_ 193.09，套接字协同部署_ 185.80，节点协同执行_ 208.17，套接字协同执行_ 201.99，节点协作调度_ 214.78，套接字协作调度_ 205.64。

浮点运算，且要求循环边界在编译期确定；加载程序还需要用户权限。因此，谷歌的 ghOSt [27] 和 F4框架 [73] 提出了不受 eBPF 限制的用户态集成调度器，但它们依赖大量内核修改，这阻碍了其在 Linux 中的普及。Syrup [30] 也有类似的局限，因为它依赖 ghOSt 或 eBPF。纯协作式用户空间调度器通常用于管理用户级线程（ULT） [8, 29, 52], ，但由于缺乏线程局部存储支持，它们无法直接适用于未经预先改造的通用工作负载。

据我们所知，USF 是首个无缝的用户态调度器框架，支持多个进程，并通过 glibc 扩展在现有 Linux 环境中运行。其 SCHED_COOP 策略在保留 TLS 支持的同时，解决了超额订阅环境中 LHP 和 LWP 的根本问题。

7 结论与未来工作

本文介绍了 USF，这是一个用于开发无缝用户空间调度策略的框架；同时介绍了 SCHED_COOP，即其默认的协作式策略。两者均通过扩展 glibc 完全在用户空间实现，无需扩展 Linux 内核，并支持线程局部存储。我们的框架支持开发专用调度策略，并提供定义标准阻塞 API 行为的方式。它支持任何依赖 glibc 标准同步原语的应用程序和运行时，不过自定义的忙等屏障需要进行适配。我们在多进程和多运行时场景下对 SCHED_COOP 进行了广泛测试，并证明它能有效管理超额订阅，性能最高可达 2.4 倍。

未来工作包括：通过 io_uring 添加对阻塞式 Linux 输入/输出的支持（如 TASIO [50]）；扩展受支持的阻塞调用范围，将 futex 纳入其中；开放更多 glibc API，以便与 USF 核心进行交互；探索 SCHED_COOP 的内核实现，并将其与 glibcv 版本进行比较；评估 USF 中基于信号的抢占；以及评估面向不同工作负载的新型自定义调度策略，例如混合关键性系统或交互式工作负载。

致谢

本研究部分获得加泰罗尼亚自治区政府（合同编号 2021-SGR-01007）以及西班牙政府通过塞韦罗·奥乔亚计划（CEX2021-001148-S/MCIN/AEI/10.13039/501100011033）提供的资助；同时，本研究也是 ST4HPC（PID2023-147979NB-C21）研究项目的一部分，该项目由 MICIU/AEI/10.13039/501100011033 和欧洲区域发展基金（欧盟）资助。本文还获得巴塞罗那泽塔级实验室的共同资助。该实验室由数字化转型与公务员事务部推动，属于复苏、转型与韧性计划的一部分，由欧盟下一代欧盟基金资助。我们感谢 BSC 的 Jungle 集群管理员提供技术支持。

参考文献

[1] Cédric Augonnet、Samuel Thibault、Raymond Namyst 和 Pierre- André Wacrenier。2009年。StarPU：一种用于异构多核架构上任务调度的统一平台。载于 Euro-Par 2009 并行处理，Henk Sips、Dick Epema 和 Hai-Xiang Lin（编）。施普林格，柏林、海德堡，863 874 10 1007 978 3 642 03869 3_80–。doi: 。/ - - [2] - - Michael Bauer 、 Sean Treichler 、 Elliott Slaughter 和 Alex Aiken。2012年。Legion：通过逻辑区域表达局部性与独立性。载于 SC '12：国际高性能计算、网络、存储与分析会议论文集。1–11。doi:10.1109/SC. 2012.71 ISSN：2167-4337。[3] Jan Bierbaum、Maksym Planeta 和 Hermann Härtig。2022年。迈向高效的超额订阅：MPI 中基于事件的通信的成本与收益。载于 2022年 IEEE/ACM 超级计算机运行时与操作系统国际研讨会（ROSS）。1–10。doi:10.1109/ROSS56639.2022.00007[4] Robert D. Blumofe、Christopher F. Joerg、Bradley C. Kuszmaul、Charles E. Leiserson、Keith H. Randall 和 Yuli Zhou。1995年。Cilk：一种高效的多线程运行时系统。SIGPLAN Not. 30, 8（1995年8月），207–216。doi:10.1145/209937.209958[5] Alex D. Breslow、Leo Porter、Ananta Tiwari、Michael Laurenzano、Laura Carrington、Dean M. Tullsen 和 Allan E. Snavely。2016年。高性能计算工作负载协同部署的理由。Concurr. Comput. : Pract. Exper. 28, 2 2016 232 251 10 1002 3187（2016 [6] 2024 6 年2月），–。doi: 。/cpe。BSC 编程模型组。。Nanos 运行时主代码库。https://github.com/bsc-pm/nanos6 访问日期：2024-04-22。[7] Qinglei Cao、Yu Pei、Kadir Akbudak、George Bosilca、Hatem Ltaief、David Keyes 和 Jack Dongarra。2021年。利用 PaRSEC 运行时支持应对具有挑战性的三维数据稀疏矩阵问题。载于 2021年 IEEE 国际并行与分布式处理研讨会（IPDPS）。79–89。doi:10.1109/IPDPS49936.2021.00017 ISSN：1530-2075。[8] Arnau Cinca、Aleix Roca、Kevin Sala、Raúl Peñacoba、David Álvarez 和 Vicenç Beltran。2025年。结合操作系统线程、用户级线程与 C++ 协程，增强 OmpSs-2 可挂起任务。载于 2025年 IEEE 国际并行与分布式处理研讨会（IPDPS）。67–80。doi:10.1109/IPDPS64566.2025.00015 ISSN：1530-2075。[9] Linux 社区。 [无日期。]。EEVDF 调度器 — Linux 内核文档。https://docs.kernel.org/scheduler/sched-eevdf.html [10] Linux 社区。2025年。CFS 调度器 — Linux 内核文档。https://docs.kernel.org/scheduler/sched-design-CFS.html [11] Linux 社区。2025年。可扩展调度器类 — Linux 内核文档。https://docs.kernel.org/scheduler/sched-ext.html[12] Dave Dice 和 Alex Kogan。2019年。通过限制并发避免可扩展性崩溃。arXiv: 1905 10818 [ ] 。cs.OS Dave Dice 和 [13] 2021 Alex Kogan。。进程内缓存与线程复用。arXiv 预印本 arXiv:2105.07497（2021年）。[14] Tu Mai Anh Do、Loïc Pottier、Rafael Ferreira da Silva、Silvina Caíno-Lores、Michela Taufer 和 Ewa Deelman。2023年。资源约束下原位工作流系综的性能评估。Concurrency and Co mputation: Practice and Experience 35, 20（2023年），e7111。[15] Eelco Dolstra。2006年。纯函数式软件部署模型。博士学位论文。乌得勒支大学，乌得勒支，荷兰。导师：Eelco Visser 和 S. Doaitse Swierstra。[16] Alejandro Duran、Marc Gonzàlez 和 Julita Corbalán。2005年。OpenMP 中嵌套并行的自动线程分配。载于第19届年度国际超级计算会议论文集（ICS '05）。计算机协会，纽约州纽约，美国，121–130。doi:10.1145/1088149.1088166[17] Jorge Ejarque、Rosa M. Badia、Loïc Albertin、Giovanni Aloisio、Enrico Baglione、Yolanda Becerra、Stefan Boschert、Julian R. Berlin、Alessan-dro D'Anca、Donatello Elia、François Exertier、Sandro Fiore、José Flich、Arnau Folch、Steven J. Gibbons、Nikolay Koldunov、Francesc Lordan、Stefano Lorito、Finn Løvholt、Jorge Macías 和 Manuela Volpe。2022年。为高性能计算、数据分析和人工智能融合启用动态智能工作流。（2022年9月）。doi:10.13039/501100002347 接收时间：2023年4月18日T11:30:03Z 出版商：爱思唯尔。[18] Marta Garcia Gasulla。2017年。混合应用程序的动态负载均衡。博士论文。加泰罗尼亚理工大学。doi:10.5821/dissertation-2117-108227 接收时间：2017-10-02T00:31:03Z 出版物名称：TDX（在线博士论文）。[19] Bolaji Gbadamosi、Luigi Leonardi、Tobias Pulls、Toke Høiland-Jørgensen、Simone Ferlin-Reiter、Simo Sorce 和 Anna Brunström。2024年。Linux 内核中的 eBPF 运行时。doi:10.48550/arXiv.2410. 00026 arXiv:2410.00026 [cs]. [20] Pawel Gepner。2021年。机器学习与高性能计算混合系统：工程与科学应用中实现性能加速的新途径。载于：2021年第16届计算机科学与智能系统会议（FedCSIS）。27–36。doi:10.15439/2021F004[21] GNU。 [未注明日期]。《GNU C 库 - GNU 项目 - 自由软件基金会》。https://www.gnu.org/software/libc/ [22] Kazushige Goto 和 Robert A van de Geijn。2008年。高性能矩阵乘法剖析。《ACM 数学软件汇刊》（TOMS）34卷3期（2008年），1–25。[23] William Gropp、Ewing (Rusty) Lusk、Rajeev Thakur、Pavan Balaji、Thomas Gillis、Yanfei Guo、Rob Latham、Ken Raffenetti 和 Hui Zhou。2023年。MPICH。 [计算机软件]https://doi.org/10.11578/ dc.20200514.13。doi:10.11578/dc.20200514.13[24] Jochen Hartmann、Mark Heitmann、Christian Siebert 和 Christina Schamp。2023年。超越感觉：情感分析的准确性与应用。《国际营销研究杂志》40卷1期，2023年，75–87。doi: . / j.ijresmar. . .[25] Hang Huang、Jia Rao、Song Wu、Hai Jin、Hong Jiang、Hao Che 和 Xiaofeng Wu。2020年。通过高效线程超额订阅来挖掘 CPU 弹性。载于《第30届高性能并行与分布式计算国际研讨会论文集》（线上活动，瑞典）（HPDC '21）。计算机协会，纽约州纽约，美国，215–226。doi:10.1145/3431379.3460641[26] E. A. Huerta、Asad Khan、Edward Davis、Colleen Bushell、William D. Gropp、Daniel S. Katz、Volodymyr Kindratenko、Seid Koric、William T. C. Kramer、Brendan McGinty、Kenton McHenry 和 Aaron Saxton。2020年。美国国家科学基金会支持的网络基础设施中人工智能与高性能计算的融合。《大数据杂志》7卷1期（2020年10月），88。doi:10.1186/s40537-020-00361-2 [27] Jack Tigar Humphries、Neel Natu、Ashwin Chaugule、Ofir Weisse、Barret Rhoden、Josh Don、Luigi Rizzo、Oleg Rombakh、Paul Turner 和 Christos Kozyrakis。2021年。ghOSt：快速灵活的 Linux 调度用户空间委派。载于计算机协会 SIGOPS 第28届操作系统原理研讨会论文集（SOSP '21）。计算机协会，纽约州纽约，美国，588 604 10 1145 3477132 3483542–。[28] 2010 doi: . / . Costin Iancu、Steven Hofmeyr、Filip Blagojević 和 Yili Zheng。。多核处理器上的超额订阅。载于 2010 年电气电子工程师学会并行与分布式处理国际研讨会（IPDPS）。1–11。doi:10. 1109/IPDPS.2010.5470434 ISSN：1530-2075。[29] Shintaro Iwasaki、Abdelhalim Amer、Kenjiro Taura、Sangmin Seo 和 Pavan Balaji。2019年。BOLT：利用用户级线程优化 OpenMP 并行区域。载于2019年第28届并行体系结构与编译技术国际会议（PACT）。29–42。doi:10.1109/ PACT.2019.00011 ISSN：2641-7936。[30] Kostis Kaffes、Jack Tigar Humphries、David Mazières 和 Christos Kozyrakis。2021年。Syrup：跨软件栈的用户定义调度。载于计算机协会 SIGOPS 第28届操作系统原理研讨会论文集（SOSP '21）。计算机协会，纽约州纽约，美国，605–620。doi:10.1145/3477132.3483548[31] Hyungyo Kim、Nachuan Wang、Qirong Xia、Jinghan Huang、Amir Yazdanbakhsh 和 Nam Sung Kim。2025年。LIA：单 GPU 大语言模型

推理加速：采用协作式 AMX 的 CPU-GPU 计算与 CXL 卸载。载于第52届年度计算机体系结构国际研讨会论文集（ISCA '25）。计算机协会，纽约州纽约，美国，544–558。doi:10.1145/3695053.3731092[32] Jonas H. Müller Korndörfer、Ahmed Eleliemy、Ali Mohammed 和 Florina M. Ciorba。2022年。LB4OMP：面向多线程应用程序的动态负载均衡库。IEEE 并行与分布式系统汇刊33卷4期（2022年4月），830–841。doi:10.1109/TPDS. 2021.3107775[33] Jonas H. Müller Korndörfer、Ahmed Eleliemy、Osman Seckin Simsek、Thomas Ilsche、Robert Schöne 和 Florina M. Ciorba。2023年。操作系统调度器与应用程序调度器如何交互？对多线程应用程序的研究。载于《Euro-Par 2023：并行处理》，José Cano、Marios D. Dikaiakos、George A. Papadopoulos、Miquel Pericàs 和 Rizos Sakellariou（编）。Springer Nature Switzerland，Cham，214–228。doi:10.1007/978-3-031-39698-4_15[34] Kin Long Kelvin Lee 和 Nalini Kumar。2023年。面向高性能计算规模科学发现的人工智能。《Computer》

56, 4 2023 116 122 10 1109 2023 3241692（4月），–。[35] doi: . /MC. . Anton Malakhov、David Liu、Anton Gorshkov 和 Terry Wilmarth。2018年。数值计算库的可组合多线程与多进程。SciPy（6月

2018 10 25080 4 1 417 003）。doi: . /[36] [ ] Majora-af f - shm 概述手册页。未注明日期。shm 概述。https://man7.org/ linux/man-pages/man7/shm_overview.7.html。访问时间：2021-08-07。[37] Raidel Martin-Barrios、Edisel Navas-Conyedo、Xuyi Zhang、Yunwei Chen 和 Jorge Gulín-González。2024年。分子动力学模拟中神经网络势概述。《国际量子化学杂志》124卷11期（2024年），e27389。[38] Andre Merzky、Mikhail Titov、Matteo Turilli、Ozgur Kilic、Tianle Wang 和 Shantenu Jha。2025年。面向数据驱动的混合型高性能计算与机器学习工作流应用的可扩展运行时架构。doi:10.48550/arXiv.2503. 13343 arXiv:2503.13343 [cs].[39] Antoni Navarro。2025年。bsc-pm/nodes。https://github.com/bsc-pm/ nodes 原始日期：2023-04-18T09:43:48Z。[40] Poornima Nookala、Kyle Chard 和 Ioan Raicu。2023年。X-Openmp——利用无锁工作窃取实现极细粒度任务处理。doi:10. 2139/ssrn.4639600[41] John Ossyra、Ada Sedova、Arnold Tharrington、Frank Noé、Cecilia Clementi 和 Jeremy C Smith。2019年。将自适应集合分子动力学工作流移植到 Summit 超级计算机。载于《高性能计算国际会议》。施普林格，397–417。[42] Daon Park 和 Bernhard Egger。2024年。利用 CPU 计算提升面向吞吐量的大语言模型推理性能。载于《2024年并行体系结构与编译技术国际会议论文集》（PACT '24）。计算机协会，纽约州纽约，美国，233–245。doi:10.1145/3656019.3676949[43]Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga 等。2019年。PyTorch：一种命令式风格的高性能深度学习库。《神经信息处理系统进展》32卷（2019年）。[44] Josep M. Perez、Vicenç Beltran、Jesus Labarta 和 Eduard Ayguadé。2017年。改进 OpenMP 中任务嵌套与依赖关系的整合。载于2017年 IEEE 国际并行与分布式处理研讨会（IPDPS）。

809 818 10 1109 2017 69 –。doi: . /IPDPS. . [45] James C. Phillips、Rosemary Braun、Wei Wang、James Gumbart、Emad Tajkhorshid、Elizabeth Villa、Christophe Chipot、Robert D. Skeel、Laxmikant Kalé 和 Klaus Schulten。2005年。利用 NAMD 实现可扩展分子动力学。《计算化学杂志》26卷16期（2005年），1781–1802。doi:10.1002/jcc.20289 _电子版：https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20289。[46] Steven J Plimpton、Axel Kohlmeyer、Aidan P Thompson、Stan G Moore 和 Richard Berger。2007年。LAMMPS：大规模原子/分子大规模并行模拟器。Zenodo（2007年）。[47] Henry Qin、Qian Li、Jacqueline Speiser、Peter Kraft 和 John Ouster- hout。2018年。Arachne：核心感知线程管理。载于第 13 届 USENIX 操作系统设计与实现会议论文集（OSDI'18）。USENIX 协会，美国，145–160。[48] Alec Radford、Jeff Wu、Rewon Child、David Luan、Dario Amodei 和 Ilya Sutskever。2019年。《语言模型是无监督的多任务学习器》。（2019年）。[49] James Reinders。2007年。Intel 线程构建模块：为 C++ 配备多核处理器并行性。"O'Reilly Media, Inc."。[50] Aleix Roca Nonell、Vicenç Beltran Querol 和 Sergi Mateo Bellido。2019年。介绍任务感知存储 I/O（TASIO）库。载于《OpenMP：征服完整硬件谱系》，Xing Fan、Bronis R. de Supinski、Oliver Sinnen 和 Nasser Giacaman（编）。Springer In-ternational Publishing，Cham，274–288。doi:10.1007/978-3-030-28596- 8 19_[51] Kevin Sala、Alejandro Rico 和 Vicenç Beltran。2020年。面向自适应网格细化应用程序的数据流并行化。载于2020年IEEE集群计算国际会议（CLUSTER）。314–325。doi:10.1109/CLUSTER49012.2020.00042 ISSN：2168-9253。[52] Sangmin Seo、Abdelhalim Amer、Pavan Balaji、Cyril Bordage、George Bosilca、Alex Brooks、Philip Carns、Adrián Castelló、Damien Genet、Thomas Herault、Shintaro Iwasaki、Prateek Jindal、Laxmikant V. Kalé、Sriram Krishnamoorthy、Jonathan Lifflander、Huiwei Lu、Esteban Mene-ses、Marc Snir、Yanhua Sun、Kenjiro Taura 和 Pete Beckman。2018年。Argobots：轻量级底层线程与任务框架。《IEEE 并行与分布式系统汇刊》29卷3期（3月

2018 512 526 10 1109 2017 2766062），–。doi：。/[53] TPDS。。Tyler M. Smith、Robert A. van de Geijn、Mikhail Smelyanskiy、Jeff R. Hammond 和 Field G. Van Zee。2014年。高性能多线程矩阵乘法剖析。载于第28届IEEE国际并行与分布式处理研讨会（IPDPS 2014）。https://doi.org/ 10 1109 2014 110。/IPDPS。。[54] Ion Stoica 和 Hussein Abdel-Wahab。1995年。最早合格虚拟截止时间优先：一种灵活且精确的比例共享资源分配机制。Old Dominion University，弗吉尼亚州诺福克，技术报告 TR-95-22（1995年）。[55] Sukeshini、Priyanka Sharma、Mohit Ved、Janaki Chintalapti 和 Supriya N. Pal。2021年。高性能计算应用的大数据分析与机器学习技术。载于《面向计算、通信与智慧世界的演进技术》，Pradeep Kumar Singh、Arti Noor、Ma-heshkumar H. Kolekar、Sudeep Tanwar、Raj K. Bhatnagar 和 Shaweta Khanna（编）。施普林格，新加坡，411–424。doi:10.1007/978-981-15-7804-5 31_[56] Boris Teabe、Vlad Nitu、Alain Tchana 和 Daniel Hagimont。2017年。锁持有者和锁等待者的抢占问题：使用知情自旋锁（I-Spinlock）防患于未然。载于第十二届欧洲计算机系统会议论文集（塞尔维亚贝尔格莱德）（EuroSys '17）。计算机协会，纽约州纽约，美国，286–297。doi:10.1145/3064176.3064180[57] Aidan P Thompson、H Metin Aktulga、Richard Berger、Dan S Bolin-tineanu、W Michael Brown、Paul S Crozier、Pieter J In't Veld、Axel Kohlmeyer、Stan G Moore、Trung Dac Nguyen 等。2022年。LAMMPS——一种灵活的模拟工具，用于原子、介观和连续体尺度上的基于粒子的材料建模。《计算物理通讯》271（2022），108171。[58] Hugo Touvron、Thibaut Lavril、Gautier Izacard、Xavier Martinet、Marie-Anne Lachaux、Timothée Lacroix、Baptiste Rozière、Naman Goyal、Eric Hambro、Faisal Azhar 等。2023年。LLaMA：开放且高效的基础语言模型。arXiv 预印本 arXiv:2302.13971（2023年）。[59] Field G. Van Zee 和 Robert A. van de Geijn。2015年。BLIS：快速实现基础线性代数子程序库功能的框架。《美国计算机协会数学软件汇刊》

41, 3 2015 14 1 14 33 10 1145 2764454（6月），：–：。[60] https://doi.acm.org/ . / Ioannis Vardas、Sascha Hunold、Philippe Swartvagher 和 Jesper Lars-son Träff。2024年。探索共分配的映射策略

高性能计算应用。载于《Euro-Par 2023：并行处理研讨会》，Demetris Zeinalipour、Dora Blanco Heras、George Pallis、Herodotos Herodotou、Demetris Trihinas、Daniel Balouek、Patrick Diehl、Terry Cojean、Karl Fürlinger、Maja Hanne Kirkeby、Matteo Nardelli 和 Pierangelo Di Sanzo（编）。Springer Nature Switzerland, Cham，271–276。doi:10.1007/978-3-031-48803-0_31[61] Raúl Peñacoba Veigas、David Álvarez Robert、Vincent A Arcila Larrea、Antoni Navarro Muñoz 和 Vicenç Beltran Querol。2025年。通过利用 nOS-V 支持 OpenMP 自由代理。载于《异步多任务系统与应用研讨会》。施普林格，95–107。[62] Han Wang、Linfeng Zhang、Jiequn Han 和 Weinan E。2018年。DeePMD-kit：用于多体势能表示和分子动力学的深度学习软件包。《计算物理通讯》228（2018年7月），178–184。doi:10.1016/j.cpc.2018.03.016[63]Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、Rémi Louf、Morgan Funtowicz、Joe Davison、Sam Shleifer、Patrick von Platen、Clara Ma、Yacine Jernite、Julien Plu、Canwen Xu、Teven Le Scao、Syl-vain Gugger、Mariama Drame、Quentin Lhoest 和 Alexander M. Rush。2020年。Transformers：最先进的自然语言处理。载于2020年自然语言处理实证方法会议：系统演示论文集。计算语言学协会，线上，38–45。https://www.aclweb.org/anthology/ 2020 6 .[64] emnlp-demos。Qingqing Xiong、Emre Ates、Martin C. Herbordt 和 Ayse K. Coskun。2018年。Tangram：通过超额订阅实现高性能计算应用共置。载于2018年IEEE高性能极限计算会议（HPEC）。1–7。doi:10.1109/HPEC.2018.8547644 ISSN：2377-6943。[65] Yonghong Yan、Jeff R. Hammond、Chunhua Liao 和 Alexandre E. Eichenberger。2016年。向 OpenMP 提出的应对 CPU 超额订阅挑战的建议。载于《OpenMP：内存、设备与任务》，Naoya Maruyama、Bronis R. de Supinski 和 Mohamed Wahib（编）。Springer International Publishing，Cham，187–202。doi:10.1007/978-3-319-45550-1 14[66] Chenle Yu、Sara Royuela 和 Eduardo Quiñones。2023年。Taskgraph：一种低争用的 OpenMP 任务框架。《IEEE 并行与分布式系统汇刊》34卷8期（2023年8月），2325–2336。doi:10.1109/TPDS.2023.3284219[67] Renchao Yu。2024年。人工智能和机器学习中的中央处理器应用。载于国际决策科学与管理会议论文集（ICDSM '24）。计算机协会，纽约州纽约，美国，221–224。doi:10.1145/3686081.3686118[68] Felippe Vieira Zacarias、Vinicius Petrucci、Rajiv Nishtala、Paul Carpen-ter 和 Daniel Mossé。2021年。高性能计算工作负载的智能共置。《并行与分布式计算杂志》151（2021年5月），125–137。doi:10.1016/j.jpdc.2021.02.010

[69] Jinzhe Zeng、Duo Zhang、Denghui Lu、Pinghui Mo、Zeyu Li、Yixiao Chen、Marián Rynik、Li'ang Huang、Ziyao Li、Shaochen Shi、Yingze Wang、Haotian Ye、Ping Tuo、Jiabin Yang、Ye Ding、Yifan Li、Davide Tisi、Qiyu Zeng、Han Bao、Yu Xia、Jiameng Huang、Koki Muraoka、Yibo Wang、Junhan Chang、Fengbo Yuan、Sigbjørn Løland Bore、Chun Cai、Yinnian Lin、Bo Wang、Jiayan Xu、Jia-Xin Zhu、Chenxing Luo、Yuzhi Zhang、Rhys E A Goodall、Wenshuo Liang、Anurag Kumar Singh、Sikai Yao、Jingchao Zhang、Renata Wentzcovitch、Jiequn Han、Jie Liu、Weile Jia、Darrin M York、Weinan E、Roberto Car、Linfeng Zhang 和 Han Wang。2023年。DeePMD-kit v2：深度势模型软件包。《化学物理学杂志》159（2023年），054801。第5期。doi：10.1063/5.0155600[70] Jinzhe Zeng、Duo Zhang、Anyang Peng、Xiangyu Zhang、Sensen He、Yan Wang、Xinzijian Liu、Hangrui Bi、Yifan Li、Chun Cai、Chengqian Zhang、Yiming Du、Jia-Xin Zhu、Pinghui Mo、Zhengtao Huang、Qiyu Zeng、Shaochen Shi、Xuejian Qin、Zhaoxi Yu、Chenxing Luo、Ye Ding、Yun-Pei Liu、Ruosong Shi、Zhenyu Wang、Sigbjørn Løland Bore、Junhan Chang、Zhe Deng、Zhaohan Ding、Siyuan Han、Wan-run Jiang、Guolin Ke、Zhaoqing Liu、Denghui Lu、Koki Muraoka、Hananeh Oliaei、Anurag Kumar Singh、Haohui Que、Weihong Xu、Zhangmancang Xu、Yong-Bin Zhuang、Jiayu Dai、Timothy J. Giese、Weile Jia、Ben Xu、Darrin M. York、Linfeng Zhang 和 Han Wang。2025年。DeePMD-kit v3：机器学习势的多后端框架。《化学理论与计算杂志》21，9（2025年），4375–4385。doi：10 1021 5 00340 ./acs.jctc.c[71]Zhanpeng Zeng、Michael Davies、Pranav Pulijala、Karthikeyan Sankar- alingam 和 Vikas Singh。2023年。LookupFFN：让 Transformers 实现轻量化计算，以进行 CPU推理。载于第*40*届国际机器学习会议论文集（*ICML'23*，第*202*卷）。JMLR.org，美国夏威夷州檀香山，40707–40718。[72] Cheng Zhang、Xingyu Zhu、Longhao Chen、Tingjie Yang、Evens Pan、Guosheng Yu、Yang Zhao、Xiguang Wu、Bo Li、Wei Mao 和 Genquan Han。2025年。通过软硬件协同优化策略提升 ARM CPU 上的大语言模型推理性能。集成电路与系统2，2（2025年5月），49–57。doi：10.23919/ICS. 2025.3568404[73] Xiaobo Zheng、Zihao Duan、Shiyi Li、Haojun Hu 和 Wen Xia。2024年。F4：快速、灵活且稳定增强的用户空间线程调度框架。载于*2024*年国际网络、体系结构与存储会议（*NAS*）。1–8。doi：10.1109/NAS63802.2024.10781349 ISSN：2835-3323。[74] David Álvarez、Kevin Sala 和 Vicenç Beltran。2024年。nOS-V：使用全系统任务调度协同执行高性能计算应用。载于*2024*年*IEEE*国际并行与分布式处理研讨会（*IPDPS*）。312–324。doi：10.1109/IPDPS57955.2024.00035

收稿日期：2025-09-01；录用日期：2025-11-10