计算机科学
COMPUTER SCIENCE

高性能计算检查点技术发展与应用综述

闫晓婷, 王小宁, 董盛, 赵一宁, 肖海力

引用本文

闫晓婷, 王小宁, 董盛, 赵一宁, 肖海力. 高性能计算检查点技术发展与应用综述[J]. 计算机科学, 2024, 51(9): 1-14.

YAN Xiaoting, WANG Xiaoning, DONG Sheng, ZHAO Yining, XIAO Haili. Review on the Development and Application of Checkpointing Technology in High-performanceComputing [J]. Computer Science, 2024, 51(9): 1-14.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231000220

高性能计算检查点技术发展与应用综述

闫晓婷[1,2]　王小宁[1]　董　盛[1,2]　赵一宁[1]　肖海力[1]
1 中国科学院计算机网络信息中心　北京 100190
2 中国科学院大学计算机科学与技术学院　北京 100049
(xtyan@cnic.cn)

摘　要　随着高性能计算系统的规模不断扩大,复杂度不断提升,应用的容错能力成为E级计算面临的重要挑战之一。检查点技术是实现应用程序的容错能力的主要手段之一,通过定期保存应用的执行状态来实现故障恢复。文中针对高性能计算检查点技术的发展和应用情况展开综述。首先,整理了高性能计算领域中检查点技术的发展;其次,根据运行层次的不同,分别阐述了系统层检查点和应用层检查点的工作,包括主流的工具软件、可用的检查点技术、使用的应用场景等;然后,讨论了检查点技术在并行计算的容错与弹性、HPC的调度与迁移、FPGA的调试、深度学习中的容错与忠实重放这4个方面的应用;最后,对检查点技术在高性能计算领域的下一步研究方向进行了展望。

关键词:检查点;高性能计算;容错;调度;作业迁移
中图分类号　TP311.1

Review on the Development and Application of Checkpointing Technology in High-performance Computing

YAN Xiaoting[1,2],WANG Xiaoning[1],DONG Sheng[1,2],ZHAO Yining[1] and XIAO Haili[1]
1 Computer Network Information Center,Chinese Academy of Sciences,Beijing 100190,China
2 College of Computer Science and Technology,University of Chinese Academy of Sciences,Beijing 100049,China

Abstract　As high-performance computers grow in size and complexity,the fault tolerance of applications becomes one of the key challenges facing exascale computing. Checkpointing technology is one of the main means used to achieve fault-tolerance of applications,enabling fault recovery by periodically saving the execution state of applications. This paper conducts a review study on the development and application of checkpointing techniques for high performance computing. First,the development of checkpointing technology in the field of high performance computing is compiled. Then,the system-level checkpointing and application-level checkpointing work are described according to the different operation levels,including the mainstream tool software,available checkpointing techniques,and the application scenarios used. The application of checkpoint technology in four aspects:fault tolerance and resilience in parallel computing,scheduling and migration of HPC,FPGA debugging,and fault tole-rance and faithful replay in deep learning,is discussed. Finally,further research directions of checkpointing technology in the field of high-performance computing are proposed.

Keywords　Checkpointing,High performance computing,Fault tolerance,Scheduling,Job migration

检查点技术是高性能计算(HPC)中广泛采用的一项关键技术。随着E级计算时代的到来,高性能计算系统正在变得越来越复杂[1],可靠性与应用容错能力成为影响高性能计算系统和应用设计的重要因素[2],大量的处理器核心和加速器组件增大了故障发生的概率[3-8]。当前PB级系统的平均无故障时间(Mean Time Between Failures,MTBF)以天为单位,但EB级系统的MTBF将以分钟为单位[9]。检查点技术是高性能计算中的主要容错机制,它可以保存进程上下文和系统状态,在系统失效后恢复进程到检查点状态,实现应用程序的容错,提高可靠性。

近年来,人工智能、HPC和数据科学的融合对检查点的发展提出了新的要求[10-11]。人工智能应用大多依赖GPU的加速[12],在有效调度多个GPU的同时提供严格的实时性的保证是当下面临的挑战[13]。人工智能模型在训练过程中往往需要分布式运行才能完成,在运行期间的关键时刻定期运行高效的检查点是大量案例中反复出现的基本模式,包括

到稿日期:2023-10-31　返修日期:2024-03-28
基金项目:中国科学院战略性先导科技专项项目(B类)(XDB0500103)
This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences(XDB0500103).
通信作者:王小宁(wxn@sccas.cn)

原地或后期分析、可重现性、邻接计算等[14]，其可以有效地保障程序运行的稳定性和有效性。

综上，检查点技术是 HPC 中广泛采用的关键技术，它通过在应用程序运行时捕获全局状态来实现容错和弹性。检查点技术虽起源于 HPC 容错，但已广泛影响到机器学习与大数据分析等领域。它不仅能够提高 HPC 计算环境的可靠性，同时也具有持续优化与发展的潜力。

本文整理了高性能检查点技术相关的文献，以期对后续工作形成指导。第 1 章梳理了检查点技术的发展；第 2 章和第 3 章分别从系统层检查点与应用层检查点两个方面对相关技术进行了总结和归纳；第 4 章对检查点的应用进行了整理；最后对检查点技术的发展进行了总结和展望，并提出了下一步研究的方向。

1 检查点技术的发展

随着应用规模的扩大，HPC 系统通过增加更多硬件组件来支持更大规模的应用部署。对于大规模应用而言，容错越来越重要，而检查点技术是 HPC 的常用容错机制之一。

检查点的发展由来已久，可以根据运行级别的不同分为系统层和应用层。系统层检查点可以保存进程的状态，备份进程及进程的上下文环境到检查点文件；应用层检查点的主要功能是保存指定应用程序的状态与上下文信息，往往需要结合特定的编程语言或编程模型来提供一定的检查点功能支持。发展较为成熟的检查点工具如图 1 所示。较早的系统层检查点技术是由威斯康星大学麦迪逊分校于上世纪 80 年代开发的异构分布式系统 Condor[15]，它支持进程级任务迁移，利用检查点在集群系统中实现进程级任务迁移，实现负载均衡，提高资源利用率。

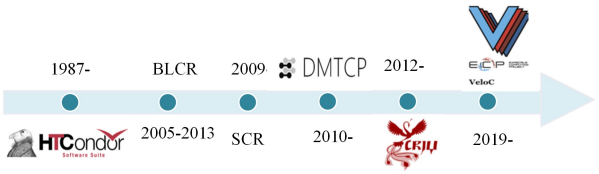

图 1 检查点技术的发展时间轴

Fig. 1 Development timeline of checkpoint technology

大约在 2005 — 2012 年期间，BLCR（Berkeley Linux Checkpoint/Restart，BLCR）[16]，DMTCP（Distributed Multi-Threaded CheckPointing，DMTCP）[17]，CRIU（Checkpoint/Restore-In-Userpace，CRIU）[18]等相继出现，并快速发展为较成熟的系统层检查点工具。BLCR 已经在 2013 年停止了维护，Open MPI 和 SLURM Workload Manager 也废弃了对早期框架 BLCR 的支持。目前最流行的检查点工具是 CRIU，其因活跃的社区支持和丰富的功能而逐渐成为检查点工具的最佳选择。与此同时，Scalable Checkpoint/Restart（SCR）[19]和 VeloC[20]是发展较为成熟的适用于 HPC 系统的应用层检查点，在并行计算中有着持续的应用。

HPC 环境中的应用以 MPI 为主流并行模式。对于 MPI 应用而言，MPI 3.1 标准并没有解决 MPI 进程故障的问题。正在开发中的 ULFM（User-Level Fault Mitigation）[21]可以使 MPI 程序从 MPI 进程故障中恢复，从而规避应用程序重启的大部分开销，包括重新安排时间、启动、初始化和读取检查点数据。每个库都使用不同的方法从 MPI 进程故障中恢复[22]。尽管 ULFM 已经被用于许多 MPI 应用的容错，但是直接将低级别的 ULFM 容错应用到一些应用中还是比较困难的，这种困难刺激了其他 MPI 进程容错扩展的发展，如 ComPiler for Portable Checkpointing（CPPC）[19]，Fenix[23]，FenixLR[24]。除了 MPI 应用外，HPC 环境中还有很多应用是基于 OpenCL 和 CUDA 这些并行编程的，它们都有相关的检查点机制来实现容错。

检查点使大规模并行作业对多节点故障具备弹性，但通常需要大量时间和存储空间。然而，检查点的存储会影响使用消息传递的并行应用的性能和可扩展性[25]。因此，降低容错技术开销成为其中的关键研究热点。随着多功能存储系统的出现，多级检查点（Multi-Level Checkpoint，MLC）[26]已经成为一种提升效率的常见方法。然而，多级检查点/重启会给 HPC 系统带来巨大的 I/O 流量。为了有效地使用多级检查点，需要对多级检查点进行优化、研究最佳检查点策略[27-28]、优化存储系统以减少检查点的 I/O 开销[29]，以及优化检查点配置的性能[30-32]。

近年来，随着人工智能和 HPC 的融合，超算中心部署了大规模的 GPU 集群。随着 GPU 计算资源的大规模应用，如何高效管理这些异构的计算资源成为一个重要的研究问题。异步迭代计算在机器学习和数据挖掘系统中很常见，对于异步迭代计算的容错框架而言[33]，减少检查点和检查点初始化的开销是一个非常重要的问题[34]。

检查点技术的研究可以满足现代 HPC 环境中的诸多需求，如提高资源利用、应对动态变化、开放共享以及长期高可用等，这也是该领域研究活跃的重要动力。检查点技术虽起步较早，但发展之路任重而道远，需要社区与相关企业的持续贡献，不断扩展工具功能与应用场景，提高性能与稳定性，从而真正产生实际应用价值。

2 系统层检查点

系统层检查点可以保存进程的状态，备份进程及进程的上下文环境到检查点文件。根据检查点运行在系统的用户空间还是内核空间，可以将系统层检查点分为用户级检查点库和内核级检查点库。现有许多关于检查点工具的研究，但大多数都偏向研究的性质，只有少数能用于实际生产。部分工具虽然较为成熟，但是也难以应对 HPC 日趋复杂的环境和规模越来越大的异构负载和并行负载，离 HPC 终端用户在实际生产工作负载中使用还有一定距离。本章从系统层检查点工具的实用与研究的视角出发，将检查点划分为发展较为成熟的主流的检查点工具和其他检查点技术，详细介绍了系统层检查点工具的原理、功能和适用场景，最后进行了总结。

2.1 主流的检查点工具

2.1.1 BLCR

BLCR[16]是一款由伯克利实验室于 2005 年开发的用户透明的系统级内核检查点工具，截至 2013 年最新版本发布，

目前已经停止维护。

如图 2 所示,BLCR 提供了用户态的 libcr 库和 kernel module 来完成相关的检查点/重启工作。它直接在内核实现了执行检查点和重启,这也意味着它可以访问所有内核资源,从而允许 BLCR 检查点/重新启动进程组(如 shell 脚本及其子进程)以及连接它们的管道。

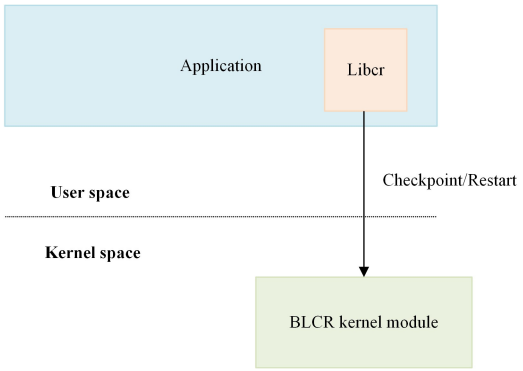

图 2 BLCR 架构

Fig. 2 BLCR architecture

BLCR 在 Linux 内核中进行检查点与恢复操作,可以访问所有内核资源,恢复用户级检查点无法恢复的资源,如进程 ID。BLCR 通过汇编代码访问底层硬件信息实现状态保存,这也意味着其内核模块难以跨 CPU 架构移植。BLCR 长期支持 x86 和 x86_64 架构,0.6.0 版本首次包含 PowerPC64 和 ARM 实验支持,0.7.0 版本增加 32 位 PowerPC 实验支持。目前,x86 和 x86_64 系统是 BLCR 测试最全面的系统架构,其他架构仅在发布时进行大量测试。将 BLCR 移植到不同 CPU 需要丰富的 Linux 内核经验和对目标 CPU ABI 与指令的深入了解。

BLCR 基于 VMADump 扩展而来,VMADump 是一个用于保存进程状态的内核模块。在配备 BLCR 的系统中,BLCR 封装了检查点与恢复操作的实现细节,提供 3 个命令:cr_run,cr_checkpoint 和 cr_restart。用户通过 cr_run 启动应用程序,该命令会在运行时动态链接用户级检查点库 libcr.so,该库提供应用程序检查点与恢复请求的接口。在正常运行中,用户根据检查点策略执行 cr_checkpoint,该命令向目标进程发送检查点请求,执行检查点操作,保存进程状态并生成检查点文件。进程出现故障时,用户通过 cr_restart 读取检查点文件,从检查点位置恢复进程执行。用户无需关注实现细节,只需执行命令就能向目标进程发送检查点或恢复请求。链接检查点库的目标进程收到请求后会触发处理函数,实现检查点与恢复功能。

BLCR 主要用于典型 HPC 应用,特别是 CPU 与内存密集型 MPI 任务。其可以在单机上对计算任务进行检查点与恢复操作,也可以在多机上通过 MPI 或其他并行系统集成执行并行任务的检查点与恢复操作。BLCR 支持保存的进程信息非常广泛,包括但不限于进程 ID、文件访问权限、CPU 寄存器状态、虚拟内存映射、打开文件信息、等待处理信号及相应处理函数等。BLCR 对应用源码限制少,无需修改源码,应用增加检查点功能后完成时间不会显著增加。BLCR 支持通过用户级回调函数处理无法检查的源,如 Cray MPI,Intel MPI 等,使用这些 MPI 实现时无需修改应用代码即可完成检查点操作。

BLCR 不支持某些资源保存,如打开网络套接字、设备文件等。对使用这些资源的应用,BLCR 提供用户级回调接口,允许用户在检查点与恢复时自行处理这些资源。

综上,BLCR 是一款功能强大的检查点工具,尤其适用于 HPC 领域。它实现了跨进程的“相当完整”的状态保存与恢复,具有较高的透明性,对应用源码的入侵性较小,成本也较低。然而,其内核依赖性较强,导致跨平台移植难度大,且开发与维护成本高。随着虚拟化技术的发展,BLCR 逐渐退出历史舞台,但其思路与技术在一定程度上影响并推动了后续的检查点工具发展。

2.1.2 DMTCP

DMTCP[17] 是一款由东北大学于 2010 年开发并维护至今的分布式多线程用户级检查点库,可以实现对用户透明的集群计算与桌面应用程序的检查点。应用程序在启动时必须链接 DMTCP 库,才能使用其功能。

DMTCP 支持跨节点计算任务迁移,但最好在相同架构与操作系统的环境下进行迁移。当从较新 CPU 迁移至不支持全部 CPU 指令集扩展的较旧的 CPU 时,可能遇到“非法指令”;当涉及 GPU 优化时也会出现此问题,可以使用 gcc-mtune=generic 编译 DMTCP、目标应用及所有库来解决;从较旧内核迁移至较新内核通常有效,新内核通常保持向后兼容。

DMTCP 的工作机制如图 3 所示。DMTCP 的协调器是一个特殊的进程,它在 DMTCP 检查点/恢复过程中起着关键的作用。在执行检查点时,必须包括 DMTCP 协调器进程。DMTCP 协调器用于协调计算组中所有进程的检查点操作。当启动用户进程时,DMTCP 首先会启动协调器进程,然后通过动态库注入(LD_PRELOAD) 来强制用户进程程序加载 libdmtcp.so 以及其他插件。在每个进程中产生一个检查点管理器线程,并通过 libdmtcp.so 和其他插件来截获应用程序中的库调用,相当于进行了一层代理,构建一个有关进程内部信息的影子数据库(例如在创建时记录有关打开的套接字的信息等),然后将请求转发给 libc 以及内核的系统调用。Libdmtcp.so 收集的信息、proc 文件系统内的进程状态等,都将用于生成应用程序的检查点文件。实验表明,DMTCP 具有出色的可扩展性,在中等规模集群上,随节点数增加,检查点时间几乎不变。

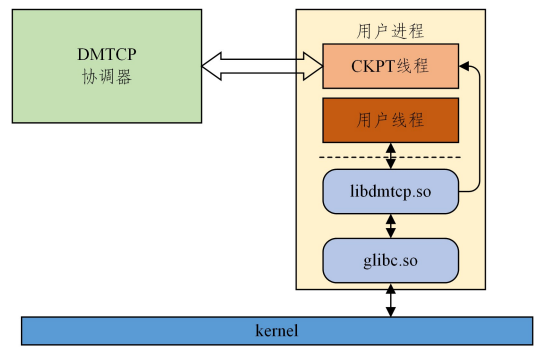

图 3 DMTCP 工作机制

Fig. 3 Working mechanism of DMTCP

DMTCP 采用协调器管理分布式应用程序的检查点，协调器的工作原理如图 4 所示。每个进程在启动时通过 LD_PRELOAD 环境变量预加载 DMTCP 库，该库在 main()前运行，创建 DMTCP 检查点线程。检查点线程与 DMTCP 协调器建立 socket 连接并注册。它还创建一个信号处理程序，默认检查点信号为 SIGUSR2，然后返回至用户线程执行。DMTCP 协调器通过 socket 向检查点线程发送消息请求检查点。检查点线程向每个用户线程发送 SIGUSR2 信号。SIGUSR2 信号仅 DMTCP 内部使用，非 DMTCP 程序不应使用。

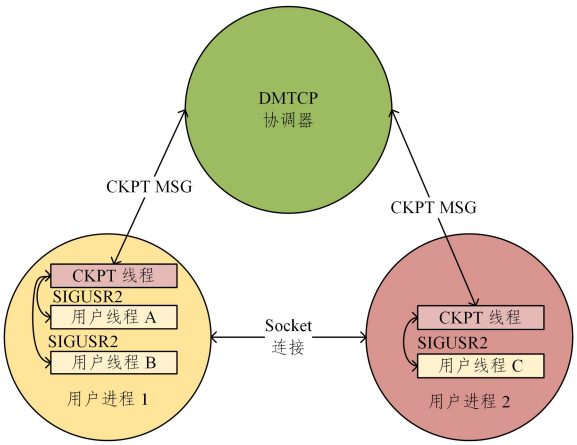

图 4 DMTCP 协调器

Fig. 4 DMTCP coordinator

DMTCP 利用协调器有效管理分布式应用的检查点，实现跨进程与跨节点的状态保存与恢复[35]。程序入侵性小，具有较高的透明性，支持定时自动检查点与手动触发，可按需配置检查点间隔，具有较高的可扩展性。但是，DMTCP 目前仅支持 Linux 操作系统，且最好在相同架构下工作。此外，DMTCP 检测到的资源范围有限，某些资源的恢复可能不完全。

综上，DMTCP 是一款功能强大的分布式检查点工具，实现了用户级的较完整的进程状态保存与恢复，尤其适用于集群计算环境。它具有较高的自动化程度、可配置性与可扩展性，对应用透明性也较高，使得用户可以低成本地对分布式应用进行检查点操作。然而，DMTCP 受限于操作系统与 CPU 架构，且资源覆盖范围有限，但这也是其进一步改进的方向。

2.1.3 CRIU

CRIU 是一款在 2012 年开源的用户级检查点工具，由开源社区开发与维护，活跃度较高。它实现了对用户透明的应用程序与容器状态的保存与恢复。CRIU 可以捕获进程或进程层次结构的 CPU、内存、磁盘与网络状态，冻结运行中的容器或应用，并生成检查点文件保存状态，用于恢复应用与容器至冻结状态运行。CRIU 可用于应用或容器的实时迁移、快照与远程调试。

CRIU 检查点过程严重依赖/proc 文件系统，/proc 文件系统是 CRIU 获取所需信息的位置。CRIU 从/proc 获取文件描述符、内存映射等信息，实现进程树的转储、清理、恢复、共享资源解析、进程树恢复等功能。

CRIU 支持通过 amdgpu 插件实现对 AMD GPU 上的计算任务状态的检查点与恢复。amdgpu 插件利用 AMD GPU 驱动的最新功能实现对 GPU 资源的状态保存与恢复。它通过 SDMA 引擎拷贝 VRAM 内容，支持多 GPU 与 ROCm 应用，不支持 vulkan compute、OpenGL 与视频加速相关功能。

CRIU 的 amdgpu 插件要考虑 3 个安全问题：远程进程读取与访问、远程进程执行控制，以及对硬件特权状态的写入访问。对于远程进程读取与访问，插件在调用时通过 ptrace 连接至调用者；对于远程进程执行控制，插件在检查点时通过 ptrace 连接至调用者，需要 CAP_SYS_ADMIN 权限恢复转储进程执行；对硬件特权状态写入访问也需要 CAP_SYS_ADMIN 权限。

CRIU 作为一款开源工具，具有较好的可扩展性，目前已集成到 OpenVZ，LXC/LXD，Docker 与 Podman 等软件，并打包到多款 Linux 发行版中。它实现的检查点范围较广，覆盖 CPU、内存、磁盘与网络资源，恢复效果较好，对应用程序透明性也较高。不过，CRIU 目前仅支持 Linux 操作系统，且依赖较新的内核版本，这也制约了其应用范围。此外，CRIU 检测到的资源范围有限，某些资源的恢复可能不完全。

综上，CRIU 是一款功能强大的开源检查点工具，实现了较为完整的进程与容器状态保存与恢复，尤其适用于云计算与容器场景。它具有较高的自动化程度、可配置性与可扩展性，对应用透明性较好，使得用户可以低成本地实现应用或容器的检查点、快照、迁移与调试等功能。不过，CRIU 的跨平台与向后兼容性有待提高，资源覆盖范围也有待扩大，这需要社区的不断贡献与发展。

2.1.4 小结

表 1 中对 BLCR，DMTCP，CRIU 这 3 个发展较为成熟、应用较为广泛的检查点工具进行了总结。

综上，这 3 款开源检查点工具各有优势：

BLCR 是较早的检查点工具，实现较为完整，但已不再维护，不支持容器与某些关键功能。

DMTCP 实现较为自动与透明，支持分布式应用，但容器支持有限，只适用于 Linux。

CRIU 支持广泛，有着较为完整的实现，有活跃的开发与维护，支持容器与云应用，但仅支持 Linux 与较新的内核版本，资源覆盖范围有限。

表 1 BLCR，CRIU 和 DMTCP 总结

Table 1 Summary of BLCR，CRIU and DMTCP

区别	BLCR	CRIU	DMTCP
进程收集机制	cr_module	Ptrace 机制	请求代理转发
Unix/TCP sockets 支持	否	是	是
Infiniband 支持	否	计划中	支持 InfiniBand 常用的 OFED API
并行/分布式计算库	支持 Cray MPI，Intel MPI，LAM/MPI，MPICH-V，MPICH2，MVAPICH，Open MPI，SGI MPT	计划中	已支持 OpenMPI，MPICH2，OpenMP，Cilk
仍在维护	否	是	是
容器支持	否	是	否
多线程/多进程	支持	支持	支持

这 3 款工具覆盖了进程级检查点的主要实现方式与应用场景。用户可根据需要选择适当的工具,或根据项目需求选择多个工具的功能来加以利用。这也体现了开源软件的优势——丰富的选择与灵活的组合。

BLCR,DMTCP 与 CRIU 的发展与功能完善,也反映了进程级迁移工具不断演进的轨迹:从单机到分布式,从裸机到云与容器,从单进程到丰富的第三方库与环境支持,功能不断扩充与改进。这预示着进程级检查点工具仍有较大的提高与改进空间,需要社区与相关企业的持续投入与发展。

2.2 其他检查点技术

Libckpt[36]是 Plank 等提出的一款基于 UNIX 操作系统的用户级检查点工具,以库函数形式实现进程状态的保存与恢复。它提供容错机制,实现大多数检查点优化,在用户空间与内核空间创建中间层,交互获取信息。正常运行时,它将进程上下文与内核参数以特定格式存储在可靠介质中,进程出现故障时使用原可执行文件启动新进程恢复状态。

Libckpt 可以在对程序员几乎完全透明的模式下使用,但也支持将用户指令合并到检查点的创建中,实现用户导向检查点。这种用户导向的检查点是其创新之处。

Kckpt[37]是 Hong 等提出的一款基于 UnixWare 内核的检查点工具。与用户级工具 Libckpt 相比,Kckpt 提供完全透明检查点与用户导向检查点。完全透明检查点不需要源码修改。与 Libckpt 相比,Kckpt 检查点的开销明显更低。

Libckpt 作为用户级工具,对应用入侵性小,具有较高的灵活性,可透明使用或结合用户指令使用。但其实现较为复杂,检查点性能较差。Kckpt 基于内核实现,入侵性较高但更加高效,具有较好的性能与自动化。两者都支持用户导向检查点,具有较好的可配置性。

这两款早期基于 UNIX 的检查点工具,实现了进程状态保存与恢复的基本功能。Libckpt 更加灵活与通用,Kckpt 性能更加优秀。两者的比较也反映了用户级与内核级检查点实现的性能与灵活性之间的权衡取舍。这为后续检查点工具的设计与改进提供了参考。

尽管这两款工具的功能都较为基础,且仅支持 UNIX 系统,但它们在检查点实现机制与方法上进行了尝试与创新,为分布式与异构环境下的检查点工具发展奠定了基础。

GPU 具有极高的计算能力和内存带宽,这在 HPC 中非常重要。许多研究者开始将 GPU 用于通用计算,称为 GPGPU。

系统层 GPU 检查点库有内核内和内核外两种实现方式,核外 GPU 检查点库无法在内核中恢复线程的计算状态,失败时必须重新加载 GPU 状态并重新启动内核,这是其主要缺点。

CheCuda[38]是 Takizawa 等提出的首款 GPU-CR 框架,通过 BLCR 获取系统状态。但 BLCR 不支持 CUDA 环境,检查点前会销毁 CUDA 的上下文,检查点后重新分配。它只支持低级 CUDA 驱动 API,需要重编译应用源码。NVCR[39]是 Nukada 等提出的用于 CUDA 的检查点,它在重新分配 CUDA 上下文后不更改其地址,支持内存相关 API 调用重放,支持 CUDA 运行时与驱动 API,无需重编译,适用于 MPI 应用,如 HPL。

HKC[40]是 Shi 等提出的 CPU-GPU 混合内核内检查点,由 CPU 处理检查点与恢复。它通过 debug API 恢复 GPU 状态,但会增大开销。HKC 以合理的成本提高 CPU-GPU 系统可靠性,弹性优于其他方案。

这几种 GPU 检查点方案各具优势:核外方案对应用入侵性小但无法在内核内恢复,CheCuda 首创但需重新编译,HKC 性能好但开销大。这些方案覆盖了 GPU 的主要检查点实现方式,为异构环境下的检查点发展奠定了基础。

尽管 GPU 检查点发展较晚,但发展速度较快,各方案不断改进与完善,功能趋于自动与高效。随着 GPU 通用计算应用的增多,对高性能与稳定的 GPU 检查点方案的需求将不断增加。这需要社区与相关企业的不断投入,扩展检查点范围,改进实现方式,提升性能与稳定性,真正将 GPU 的计算能力与检查点技术有机结合,发挥 GPU 在 HPC 中的重要作用。

2.3 小结

系统层检查点根据运行级别分为用户级与内核级。内核级直接访问目标进程与资源,用户级通过其他方式获取信息。内核级作为内核模块存在,需要重新编译内核,用户级则无此需求。内核级性能优异但移植性差,用户级则相反。两种类型各有优势,需要根据具体应用场景进行选择。

检查点透明性体现在其无需修改或重新编译应用程序即可使用,大幅减少了故障概率,无需关注实现细节,访问内核数据结构保存全部进程状态,但依赖内核,不可移植。不透明检查点需在应用程序中实现检查点与恢复,增加了设计复杂度,需要针对应用定制,但不需要访问内核,更适用于异构迁移,可以通过用户指定内容减少开销。

表 2 对本文提到的系统层检查点进行了总结。目前,主流的系统层检查点工具包括 BLCR,DMTCP,CRIU 等。这些工具可以有效保存进程状态与上下文,实现进程容错与恢复。但大多数工具仅支持特定 CPU 架构,尚不能很好地支持 GPU 等异构设备。随着 GPU 计算和其他异构设备的广泛应用,对跨平台和异构支持良好的检查点工具的需求将日益增加。

表 2 系统层检查点总结

Table 2 Summary of system layer checkpoints

检查点库	是否对用户透明	适用的处理器	支持平台	运行级别
BLCR	透明	x86 和 x86_64 架构、PowerPC64、PowerPC32	Linux 的许多主要供应商发行版。SuSE,RHEL,Fedora,Debian 和 Ubuntu,vanilla (2.6.0−3.7.1)	内核级
DMTCP	透明	x86 and x86_64,32-bit ARM CPU (armv7/armv7a),64-bit ARM(armv8)	Linux	用户级
CRIU	透明	CPU,AMD GPU	Linux	用户级
CheCuda	不透明	NVIDIA GPU	Linux	内核级
NVCR	透明	NVIDIA GPU	Linux	内核级
HKC	透明	CPU-GPU	Linux	内核级
Libckpt	不完全透明	CPU	Unix	用户级
Kckpt	透明	CPU	UnixWare	用户级

总体而言，系统层检查点技术发展时间较长，各工具在性能、自动化与易用性上均在持续优化，但可移植性与通用性还有一定的提高空间。随着异构计算环境的兴起，对高性能的异构检查点方案的需求将不断增长。未来需要扩展现有检查点工具的应用范围或者开发新的检查点工具，加强对新兴异构体系结构的支持。只有深入理解 HPC 系统的特征，设计出高效稳定的检查点机制，检查点技术才能真正发挥其重要作用。

系统层检查点技术的发展离不开操作系统内核的进步和社区的持续贡献。这需要相关研究者与开发者研究 HPC 系统的要求与发展趋势，持续推进理论与技术创新，使检查点工具的覆盖范围不断扩展，提高其稳定性与性能，才能为科学技术发展与产业应用提供持续支撑。只有深入实践，不断改进，体现技术价值，检查点机制才能成为高性能与高可靠计算环境中的关键支撑手段。

3 应用层检查点

应用层检查点的主要功能是保存指定应用程序的状态与上下文信息，往往需要结合特定的编程语言或编程模型来提供一定的检查点功能支持。并行编程是在 HPC 中广泛应用的编程模式，并行应用程序可以通过通信失败而意识到并行系统中的故障。应用层检查点各自对应不同的并行框架与异构环境，需要考虑相应平台与语言的特点进行定制实现，部分机制较为复杂。本章主要介绍了面向 MPI 计算的检查点、面向异构计算的检查点、面向异构存储的多级检查点，并在最后进行了总结。

3.1 面向 MPI 计算的检查点

MPI(Message Passing Interface)是 HPC 领域最流行的消息传递库，广泛用于支持并行程序的开发。MPI 应用程序由多个进程组成，这些进程可以在单机上也可以在机群系统中运行。MPI 的应用层检查点主要保存 MPI 应用程序中所有进程的状态，以实现 MPI 应用容错与恢复。

通常使用协调协议来确保本地创建的检查点的全局一致性。协调协议包括完全协调、半协调或完全不协调。完全协调的主要优点是其与应用程序无关，可以创建全局一致的检查点；主要缺点是进程同步带来的开销较大。不协调的协议在检查点创建期间不需要同步，从而减少了开销并允许应用程序不平衡。然而，在恢复过程中，必须找到一个全局一致的状态。由于这些协议无法保证恢复时的检查点全局一致性，所有进程最终都可能回滚到执行的开始，即多米诺骨牌效应。不协调的检查点协议可以利用消息日志来避免分段检测的多米诺骨牌效应。

ULFM[21]是 MPI 论坛提出的 MPI 规范的扩展，其提出了一个 MPI 容错模型来处理 MPI 进程故障，目标是将进程从故障中恢复。ULFM 提供检测与传播 MPI 进程故障信息、修复 MPI 通信器及支持各种恢复策略的最小接口。ULFM 提供了很高的灵活性，取消了将用户与错误处理过程隔离的做法，不定义特定的恢复机制，用户可以采用任何恢复策略，包括检查点回滚或基于算法的恢复技术，还可以自定义不同的恢复级。ULFM 实现需高效可靠故障检测器，相比现有 MPI 实现能够更可靠地检测故障。截至目前，有两个 ULFM 的原型实现：一个基于 Open-MPI，另一个包含在 MPICH 中。

CPPC[41]是一个开源的检查点工具，最初采用停止并重启的检查点策略用于 MPI 应用。CPPC 进行了扩展以利用 ULFM 提供的新容错能力。该方案能检测一个或多个进程的故障，并从故障中恢复，无需停止应用执行并且可以保留运行应用的 MPI 进程数。使用一个从源头到源头的编译器和静态分析来检测源代码，只对应用重启所需的变量进行非阻塞式多级检查点。检查点被保存为可移植的 HDF5 格式。使用工具化的 GOTO 和 RETURN 语句代替非本地跳转，可以为 Fortran 实现全局非收缩回滚恢复。CPPC 对最终用户显示为编译器工具和运行时库，它自动检测应用程序代码获得等效容错版本，定期保存/恢复计算状态。CPPC 实现了优化来减少检查点开销；使用存活性分析和全零块技术减小检查点文件；多线程转储重叠检查点转储与计算；可移植格式和排除体系结构相关状态使检查点文件可移植，在异构集群完成应用程序。

ULFM 和 CPPC 都是为 MPI 应用提供容错支持的工具，但存在以下主要区别[42]。

1)接口级别：ULFM 提供低级 API，支持各种容错模型，但需要用户设计恢复策略。CPPC 提供编译器工具和运行时库，自动检测应用代码实现等效容错版本，屏蔽用户复杂性。

2)应用广泛性：ULFM 的提议多针对特定应用或应用集，利用应用特征简化恢复过程。CPPC 的方案适用于任何 SPMD 应用，自动检测常规 MPI 代码。

3)恢复策略负责：ULFM 下，用户负责确定恢复策略、保存哪些数据及保存在哪些程序点。CPPC 方案由系统自动完成，屏蔽了该需求。

4)实现机制：ULFM 作为 MPI 标准的扩展，提供容错功能。CPPC 独立实现，检测普通 MPI 代码转换为等效容错版本。

5)支持模式：ULFM 支持各种容错模型和用户选择。CPPC 支持全局非收缩回滚模型，通过检查点恢复。

总之，ULFM 和 CPPC 都为 MPI 应用提供容错支持，但 CPPC 提供更高级别的屏蔽，自动检测常规 MPI 代码获得等效容错版本，通过检查点支持非收缩恢复模型。ULFM 作为 MPI 标准扩展，需要用户设计恢复策略及确定恢复细节。CPPC 适用于任何 SPMD 应用，而 ULFM 的提议常针对特定应用。CPPC 方案自动完成恢复，而 ULFM 下用户需承担更多责任。

Fenix 在 MPI 进程失败时会自动重建在 Fenix 注册的“弹性通信器”。Fenix 同时支持收缩和非收缩的全局恢复。对于非收缩性恢复，“弹性数据”会在替换的 MPI 进程中自动恢复，可以在不中断作业的情况下实现在线全局恢复。在这种方法中，每个故障都会触发全局恢复，这需要所有幸存进程来恢复 MPI 环境。然后，所有幸存的进程，以及新生成的

进程,都必须回滚到最后一个全局一致的检查点。全局恢复的优点是可以以半透明的方式恢复进程,应用程序不必知道故障。

FenixLR 是对 Fenix API 的重新实现,它使用消息记录来支持本地非收缩回滚恢复。FenixLR 不受标准的 Fortran 非本地控制流限制的影响,因为本地恢复意味着 MPI 进程失败不会导致幸存的 MPI 进程的应用控制流被改变。FenixLR 为 C,C++和 Fortran 提供了绑定。本地恢复在故障后不需要全局恢复环境,并且只有新生成的进程才需要回滚到最后一个检查点。实验证明了 FenixLR 能够容忍高频动态注入节点故障,同时在多达 262 144 个核心的规模上保持 S3D 的持续性能。

对比 Fenix 和 FenixLR,两者的区别如下:Fenix 在 MPI 进程失败时自动重建注册的"弹性通信器",支持收缩与非收缩全局恢复。FenixLR 使用消息日志支持本地非收缩回滚恢复,不需要全局恢复环境,只有新进程需要回滚到最近的检查点。FenixLR 可在高频动态节点故障下保持较优的性能,在 26 万核规模 S3D 应用中开销仅 13.75%。FenixLR 编程开销很低,S3D 应用的容错仅需 35 行代码。

表 3 中总结并对比了面向 MPI 计算的检查点。

表 3 面向 MPI 计算的检查点

Table 3 Checkpoints for MPI computing

名称	接口级别	应用广泛性	使用 ULFM	恢复模式
ULFM	提供低级 API	针对特定应用	属于 ULFM	应用指定
CPPC	提供编译器工具和运行时库	常规 MPI 代码	是	全局非收缩回滚模型
Fenix	提供编程接口	基于模板的并行应用程序	是	在线收缩/非收缩全局恢复
FenixLR	提供编程接口	基于模板的并行应用程序	是	本地非收缩性回滚

3.2 面向异构计算的检查点

CUDA 和 OpenCL 都是用于异构计算的并行编程模型,CUDA 是 NVIDIA 提出的通用并行计算架构,被广泛用于 GPU 异构计算。OpenCL 是一种用于异构系统的并行编程语言,其应用程序可以运行在多种处理器上,如 GPU,DSP 和 CPU。其应用层检查点需要同时保存不同处理器(如 GPU,DSP 和 CPU)上运行的应用状态,这就需要考虑不同平台与语言之间的兼容性,并且部分检查点的实现机制较为复杂[43]。

对于 CUDA 而言,Guo 等[44]提出的数据结构与机制用于 C/R 不同 GPU 存储器中的计算状态,是 CudaCR 发展的基础。其应用程序级实现仅限于迭代应用,内核内故障检测会在检测到故障的同时破坏内核,并在该内核的下一次迭代中恢复。其使用内核中断作为检查点时间同步块的方法。

CudaCR[45]是 Pourghassemi 等提出的一个可扩展的应用层 GPU 内核内检查点/重启框架。用户只需确定检查点位置,预编译器自动转换 CPU 与 GPU 代码为能够进行检查点与恢复操作的代码。他们讨论了从不同 GPU 存储器异步收集计算状态的算法与数据结构,提出了显著减少二次数据存储与检查点开销的优化。在最佳情况下,以低于 10%的开销恢复计算任务的运行时状态。

统一虚拟内存(UVM)支持大内存 GPU 程序,最近与 CUDA 8 和 Pascal GPU 一起推出。较老的 CUDA 直接加载/卸载内存段,较新的 CUDA 已经开始利用 UVM。因此,基于 UVM 的检查点越来越重要。Garg 等[46]提出了新的可扩展检查点机制 CRUM,可用于跨计算机节点的混合 CUDA/MPI 计算。CRUM 支持快速分叉检查点,运行时平均开销为 6%,分叉检查点时间约为传统同步检查点的 40 倍。

综上,CudaCR 是可扩展应用层 GPU 内核检查点/重启框架,支持预编译器自动生成检查点与恢复代码,其中的优化机制减少了开销。CRUM 是首个在 UVM 下演示的检查点机制,支持 CUDA/MPI 混合计算跨节点的迁移,对大内存 GPU 程序与快速分叉检查点更有优势。相比传统同步检查点,CRUM 运行时开销更低,分叉检查点速度更快。

对 OpenCL 而言,CLPKM[47]是 Chiu 等提出的一种框架,它在 OpenCL 应用程序与 OpenCL 运行时底层之间提供抽象层,基于软件检查点机制实现内核执行实例的抢占。CLPKM 包括:

1) 拦截 OpenCL API 调用的 OpenCL 运行时库;

2) 执行允许抢占转换的源到源编译器;

3) 使用基于优先级的抢占调度技术调度 OpenCL 任务的守护进程。

CRState[48]是 Chen 等提出的可以实现 GPU 内核中的检查点/重启操作。底层硬件中的计算状态包括堆、数据段、本地存储器、堆栈和代码段,被 CRState 识别与具体化以建立底层状态与应用层表示之间的关联。通过编译时用预编译器将原语插入 OpenCL 程序,在运行时提取计算状态。由于计算状态在应用程序级别被复制,OpenCL 程序可以被抢占并跨异构设备移植。通过全面的实例和 10 个权威基准程序证明了 CRState 的可行性与有效性。

综上,CLPKM 与 CRState 是实现 OpenCL 应用层检查点的两种有效机制。CLPKM 基于软件实现内核执行实例的抢占与调度,实验表明其可以有效降低高优先级进程的加速比,提高系统的整体公平性。而 CRState 可以在 GPU 内核级别进行检查点与重启操作,通过识别与提取底层硬件的计算状态实现与应用层表示的关联,使 OpenCL 程序具有抢占与跨平台移植能力。两种机制的可行性与效果均已通过实验或实例证明。

总之,针对 CUDA 和 OpenCL,已有相关工作研究应用层检查点机制。这为高性能异构计算的容错与负载均衡提供了一定支持。

3.3 面向异构存储的多级检查点

SCR[49]是一个可扩展检查点/重启方案。随着系统内存大小的增长速度快于并行文件系统的带宽,检查点的成本开始影响应用程序的运行时间。多级检查点通过在一次运行中具有不同成本和不同弹性级别的多种类型的检查点潜在地

解决了这个问题，其使用不同成本和弹性的检查点来应对不同类型的故障。SCR 不仅可以将检查点写入并行文件系统，还可以写入计算节点的 RAM、闪存或磁盘。SCR 支持 C/C++，Fortran 和 Python 应用。SCR 可以利用 Linux 集群的分布式存储获得高 I/O 带宽，用于检查点、重新启动和写入大型数据集。使用 SCR，作业运行更高效，发生故障后重新计算的工作量更少，共享资源如并行文件系统的负载也更轻。SCR 是一个低成本的检查点方案，其速度比并行文件系统快 100～1000 倍，并且可以有效地抵御 85%的系统故障，这使得机器效率提高了 35%，并行文件系统的负载减少了一半[50]。SCR 提供以下功能：可扩展检查点、重启和输出带宽、异步数据传输到并行文件系统、最佳检查点频率指导、自动跟踪并从最近的检查点重新启动、挂起或失败后，在分配中自动重新启动作业。

VeloC[20]是一个用于 HPC 和大规模数据中心的多级检查点/重启运行时。它旨在为复杂的异构存储层次提供高性能和可扩展性，同时不牺牲易用性和灵活性。VeloC 是阿贡国家实验室和劳伦斯利弗莫尔国家实验室之间的合作成果，是美国 Exascale 计算项目的一部分，2018 年在 github 上发布了第一个版本。向外部存储（如并行文件系统）的全局检查点是许多 HPC 应用的常见 I/O 模式，但由于外部存储 I/O 吞吐量有限，全局检查点通常会导致 I/O 瓶颈。为解决这个问题，学者们越来越多地采用从同步检查点到异步检查点（如先写入本地存储，再异步刷新到外部存储）的转变。但是，随着每个节点核心数量的增加和本地与外部存储的异构性，设计高效的异步检查点机制因节点本地与全局级别的高并发和 I/O 性能变化之间的复杂交互作用而变得困难。

VeloC 提出了一系列设计原则来实现异构本地存储的高效异步检查点。

1）隐藏异构存储的复杂性；

2）将异步写入合并到外部存储的活动后端，实现 I/O 并行性的弹性控制；

3）细粒度分块检查点，更好地利用节点本地设备；

4）使用性能建模的自适应块放置。

VeloC 通过一系列算法描述来实现这些设计原则，这些算法描述基于 VeloC 运行时。VeloC 在 ANL Theta 系统上进行评估，使用合成基准和 HPC 应用，结果显示其可以显著提升异步检查点的性能，大大减少应用运行时间的干扰。

综上，SCR 和 VeloC 都是支持异构存储的检查点框架。SCR 实现多级检查点，支持各种本地存储和并行文件系统，对不同故障采取不同策略，可有效提高系统效率和减少并行文件系统负载。VeloC 专注于异构本地存储，提出设计原则实现高效异步检查点，通过算法实现，可显著提升异步检查点性能和减少应用运行干扰。这两种框架均考虑到异构存储的优点，可为异构存储环境下的 HPC 提供更灵活和高效的检查点方案。

3.4 小结

MPI 是 HPC 最广泛使用的消息传递库，其应用检查点主要保存所有进程的状态以实现容错。有关 MPI 应用检查点的研究较多，如 ULFM，CPPC，Fenix 与 FenixLR 等开源工具实现 MPI 应用的容错与恢复。这些工作考虑了 MPI 应用的特点，提供了不同粒度的容错方案。ULFM 是 MPI 论坛提出的 MPI 规范的扩展，提供 MPI 进程故障检测与恢复的最小接口。ULFM 实现需高效可靠的故障检测器，目前有 Open-MPI 和 MPICH 的原型实现。CPPC 是一款开源检查点工具，采用编译器检测源代码并采用非阻塞的方式通过多级检查点来保存关键变量；使用 HDF5 格式保存检查点，支持进程故障检测与恢复。CPPC 实现了存活性分析、全零块技术和多线程转储重叠检查点与计算等优化。如果要简单容易地为 MPI 应用获得容错能力，CPPC 可能更适合。如果需要更细粒度控制或自定义的容错方案，ULFM 作为 MPI 标准扩展可能更灵活。但对于大多数用户来说，CPPC 提供的自动和简单机制可能更实用。Fenix 在 MPI 进程失败时自动重建注册的“弹性通信器”，支持收缩与非收缩全局恢复。FenixLR 使用消息日志支持本地非收缩回滚恢复，不需要全局恢复环境，只有新进程需要回滚到最近检查点。

面向异构计算的检查点研究较少，目前主要集中在对 CUDA 和 OpenCL 两个并行编程模型的研究上。针对 CUDA，已有应用层检查点框架实现容错，并考虑不同 GPU 存储器和 UVM 的特点。针对 OpenCL，同时有应用层软件检查点实现内核执行实例抢占，以及内核级别检查点识别底层硬件状态实现程序抢占与移植。这些工作部分考虑了异构系统的特点，为异构 HPC 带来更强的容错与弹性。但总体上异构计算检查点仍需加强，如对 FPGA 和其他异构加速器的支持。

面向异构存储的检查点研究考虑存储层次的异构性与性能差异，SCR 提供多级检查点对不同故障采用不同成本的检查点。VeloC 提出设计原则和算法实现异构本地存储高效异步检查点，通过在 Theta 上评估显示效果显著。这些工作在一定程度上解决了异构存储环境下 I/O 瓶颈与性能问题，为异构存储 HPC 应用提供更优的容错与性能。但总体来说，异构存储检查点机制仍需进一步研究，目前主要局限在几种存储介质的组合上，对更丰富的存储层次支持不足。

总体而言，应用层检查点可以根据用户需求与应用特征进行更好的定制，有助于减小检查点规模和性能损失，体现更高的可扩展性。未来，需要进一步拓展现有检查点工具的应用范围，支持更丰富的应用类型与异构平台，提高工具的通用性。这需要相关社区的广泛合作与持续贡献。预计未来，强大的 HPC 系统会比目前的系统有更高的故障率。故在这种未来系统上运行的 HPC 应用比在今天的机器上更有可能遇到系统故障。因此，应用程序的容错性变得更加重要。

4 检查点的应用

检查点技术在并行计算中主要用于错误恢复，在其他领域的作用还包括调试与忠实重放。其基本思想是：在程序执行过程中，周期性地捕获程序的状态（检查点）并保存到某种持久化存储中。如果之后程序出现错误或故障，可以从最近

的检查点状态恢复,而不是从头重新执行。检查点技术以其灵活性和通用性,在系统软硬件协同与优化、高可靠性计算等方面有着广泛的应用前景。相关理论和实现技术的进步,将大大提升系统的性能、可靠性与易用性。本章主要从并行计算的容错与弹性、HPC 的调度与迁移、FPGA 的调试、深度学习中的容错与忠实重放这 4 个方面介绍检查点的应用。

4.1 并行计算的容错与弹性

HPC 系统为提升性能不断增加计算节点和核心数量,长期运行和可扩展的 MPI 应用程序容易受到节点故障的影响,如果没有容错机制,已完成计算的工作可能丢失,重新计算的成本很高。为解决此问题,HPC 系统提供了容错技术,如检查点和回滚恢复,允许系统从故障恢复到一致状态。将最新的弹性算法和技术整合到现有代码中是容错的独特挑战,除了实现弹性本身的复杂性之外,集成独立新策略与应用现有容错机制也是一大难点。

为了实现并行应用程序的检查点的全局一致性,需要用到检查点回滚恢复算法。检查点回滚恢复算法包括完全不协调检查点、半协调检查点和协调检查点。

使用非协调检查点技术,当故障发生时,并非所有的进程都会回滚。当使用非协调检查点时,有必要在日志文件中保存消息痕迹,以便在恢复期间能够返回到当前状态(避免多米诺效应)。非协调检查点独立保存每个进程的状态,因此减小了控制消息日志的大小。但是,当不协调检查点实施基于接收器的悲观回滚恢复协议时,为保护 MPI 消息而增加的延迟会急剧增加。这个问题是由节点内和节点间的网络带宽的差异造成的。因此,在多核系统中,这种基于接收器的悲观消息记录的性能甚至更糟。

协调检查点协调所有进程回滚来恢复全局状态,由于 I/O 拥堵,检查点的协调可能会减慢应用程序的执行。在故障发生后,每个应用程序的进程都从最后的检查点重新启动。这种技术的缺点是,当应用程序的进程数量增加时,由于要执行与回滚后的重新执行有关的计算工作,计算成本变得十分高昂。

除了协调检查点和非协调检查点,还有一种方法是半协调的检查点,目的是减少容错机制的强制开销。这些技术在处理规模增长方面很有用,特别是增加每个节点的核心数时。有了半协调检查点,就没有必要再保存一组进程的内部通信。因此,需要记录的消息较少,因为它只需要保存进程组之间的相互作用。

RADIC 架构[51]是一个并行应用程序的容错结构,它提供了保护、监测和恢复功能,以保证 MPI 应用在节点发生故障时也可以成功执行。最初,RADIC 架构是在一个被称作 RADICMPI 的 FT-MPI 库中实现的。该库被开发出来用于单核系统,单核系统就是一个进程通常在一个核心中运行的系统。当一个计算节点发生故障时,只有一个进程需要被恢复。这个库实现了 MPI 调用的一部分,从而限制了用户可以执行的应用程序的数量。之后,在套接字层面和系统层面使用半协调检查点进行了一些研究[52]。此外,还有用户级半协调检查点中间件 ULSC2-RADIC[53],其允许将 MPI 应用程序划分为独立的分区,这些分区之间相互通信。利用现有的工具,如 DMTCP 和 MPI 库,ULSC2-RADIC 中间件控制分区以创建半协调检查点。

Whitlock 等[54]设计、实现并验证了一个综合的运行时系统,通过利用弹性工具 Fenix 更新了 Kokkos 弹性和 VeloC 的使用模式,以支持弹性运行时的应用级整合。通过设计可整合的系统而不是整合的系统,允许用户设计优化和升级复原力技术,同时保持一体化复原力解决方案的简单性和良好性能,可以获得更好的长期灵活性、性能,以及简单性。

Parasyris 等[55]提出了 FRAME,这是一个容错解决方案,通过首次将异步多级检查点库即容错接口 FTI 与 MPI 的在线容错解决方案 Reinit 相结合,大大缩短了应用恢复时间。具体来说,FRAME 利用支持 Reinit 的 MPI 来提取故障的拓扑结构,并优化 FTI 中的检查点检索,以节省识别和获取系统中最新可用检查点的大量开销。与基线 FTI 相比,FRAME 的优化将检索检查点的时间减少到 67%。包括基于 Reinit 的 MPI 恢复的结果显示,在 32768 个 MPI 行列的大规模执行部署中恢复 1.3TB 的检查点数据时,该方法的端到端恢复时间减少了 360%。

综上,容错技术特别是检查点/重启机制在大规模 HPC 系统中的应用至关重要。相关研究的目标在于实现高效的检查点/重启策略,在提供高容错能力的同时降低资源消耗,这需要系统软硬件协同优化,同时也是 HPC 系统可扩展性的重要支撑技术。

4.2 HPC 的调度与迁移

到目前为止,HPC 集群上运行的作业通常与开始运行的节点绑定,这限制了资源利用效率和作业的可调度性。同时,随着人工智能与高性能的融合,GPU 资源被大量应用在高性能环境中,在大规模 GPU 集群上运行的作业往往会遇到各种动态变化,如用户数量变化、作业规模变化、硬件异常等,这会导致作业的运行时间延长和资源利用率下降。针对这种动态变化,如何通过作业迁移和检查点技术提供容错和提高资源利用率成为研究的重点。传统的 GPU 集群资源管理通常采用静态的资源隔离和划分,这降低了资源利用率;而开放的资源管理可以跨用户动态调度资源,最大化集群的整体效率,这需要作业迁移和检查点技术来保证性能。新一代 GPU 的发布会导致大型 GPU 集群最后有多种 GPU 组合,GPU 资源静态划分虽然可以保证可预测性和隔离性,但是 GPU 的利用率会变低,如何在不同用户之间智能分配不同代的 GPU 以在最大限度提高效率的同时确保公平性,对 GPU 的调度提出了更高要求。

到目前为止,在 HPC 集群上运行的作业都是与开始执行的节点绑定在一起的。Rodríguez-Pascual 等[56]将用户级检查点/重启机制集成到资源管理器中,实现了透明的作业迁移,为调度和工具带来了更大的可能性,同时提高了容错性。这在未来超大规模 HPC 集群中尤为重要,因为高效调度的程度和复杂性不断增加,使得获得应用所要求的并行程度

成为挑战。在大规模的计算集群中,由于作业规模和机器服务能力的变化,作业的完成时间可能会延迟。为了解决这个问题,现有的工作已经提出了各种基于冗余的调度算法。Xu等[57]采用严格的分析方法来设计使用冗余和检查点的调度算法,通过广泛的模拟证明了该算法在非多任务和多任务模式下都能显著减少作业流时间。

虚拟化技术可以提高集群的资源利用率,远程 GPU 虚拟化技术可以提供灵活性,而这一灵活性可以通过应用迁移机制来进一步提高。通过迁移,应用程序的 GPU 部分可以在执行期间以透明的方式实时迁移到集群中其他地方的 GPU。Prades 等[58]介绍了 rCUDA 远程 GPU 虚拟化中间件中迁移机制的实现,并在三代不同的英伟达 GPU 上对 rCUDA 中迁移机制的实现进行了全面的性能分析,还使用了两个不同版本的 InfiniBand 网络进行互连测试。通过 GPU-作业迁移机制可以为数据中心带来非凡的好处。

Gandivafair[59]是一个跨用户的公平 GPU 集群调度系统,它在用户之间动态分配不同 GPU,最大化集群利用率,同时保证性能隔离和公平性。关键机制是作业在节点之间的迁移,基于 PyTorch 和 TensorFlow 的检查点/重启实现。

总之,开放的 GPU 资源管理与作业迁移技术在提高大规模 GPU 集群效率与容错性方面有着非常重要的作用。开放的资源管理通过跨用户动态调度最大化资源利用,而作业迁移与检查点技术保证了应用的高可用性。两者结合为构建规模庞大且异构的 GPU 计算环境提供了良好的思路。开放的资源管理与高级作业控制技术是实现弹性与高效的异构计算环境的关键,这也是 HPC 系统下最为重要的技术方向之一。

4.3 FPGA 的调试

FPGA 与 HPC 之间存在密切的相互作用和促进关系。FPGA 的可重配置性和高效性使其成为 HPC 系统优良的协处理器和加速器,也可用于 HPC 系统的建模与评估。FPGA 已经在 HPC 基础设施中发挥着不可替代的关键作用。

对于 FPGA 项目来说,调试的效率是至关重要的,因为目前平均有 51% 的 FPGA 项目时间是用于验证和调试的[60]。此外,随着设计复杂性的提高,这一比例也在不断增大。两个主要的 FPGA 调试流程是仿真和片上调试[61]。然而,它们提供了有限的可观察性和低可控性,很难找到故障的根本原因。因此,开发具有类似于软件的速度和可观察性组合的调试流程尤其有吸引力。

StateMover[62]是一个基于检查点的调试器,可以在 FPGA 和模拟器之间来回移动设计状态。这允许设计以全硬件速度运行,直到达到一个感兴趣的点。StateMover 不完全支持带有外部 I/O 的设计,它不能用于调试具有外部 I/O 流的接口,但是这些接口在 FPGA 系统中非常常见。StateMover 也不适合与大型存储介质(如硬盘)连接的设计。在这种情况下,将片外状态移动到模拟器上是不现实的。

StateLink[63]是一个基于事务的协同仿真框架,它允许系统的一部分在模拟器中运行,并且仍然与在硬件中的其他系统组件进行交互。StateLink 允许任务在其状态被转移到模拟器中后,仍然与整个硬件系统相连并处于活动状态。将基于检查点的调试框架的功能扩展到具有外部 I/O 的设计,大大加快了作为大型系统一部分的任务的仿真速度。

综上,StateLink 等基于检查点和事务的协同仿真技术在 FPGA 设计调试中的应用,实现了软硬件协同,提高了调试效率,这是 FPGA 设计验证中的重要技术进步和创新。相关技术的发展有助于降低 FPGA 设计复杂度带来的验证难度,加速 FPGA 产品研发进度。

4.4 深度学习中的容错与忠实重放

深度学习应用逐渐成为高性能计算和云系统最重要的应用之一,其所使用的海量数据集和深度神经网络给 HPC 带来了许多挑战。检查点在深度学习中的主要作用是容错,除此之外,检查点对深度学习模型训练过程中的忠实重放也非常重要。忠实重放可以提升生产率、模型性能和鲁棒性,并有助于安全审计。因此,对深度学习来说,HPC 检查点/重启工具是一个极具吸引力的选择。

大多数并行深度学习训练工作使用的是一种被称为根检查点(Root Checkpointing)的原始方案,这种方案会受到阻塞语义和前进进度滞后的影响。Anthony 等[64]将多级检查点工具 SCR-Exa 应用于分布式 DL 应用。在领先的 TOP500 系统 Lassen 上大规模测试了两个深度神经网络模型的性能,同时确保深度神经网络在模拟系统故障重启后仍能保持准确性。测试结果表明,多级检查点方案能够在大规模条件下实现几乎不变的开销。这项研究首次评估证明了分布式 DL 的检查点方案具有很强的可扩展性,并且无须对特定框架进行更改。

深度学习训练中固有的非确定性给忠实重放带来了突出的挑战。即使使用固定的随机种子,同一训练管道的多次运行也可能产生性能相差 20% 的模型。利用现有的基础架构检查点支持,开发人员无法忠实地重放训练过程。Xu 等[65]提出了 DETrain,一种针对长期运行的深度学习训练程序的检查点和忠实执行/重放的新解决方案;引入了一种新的随机数生成机制,它能在数据并行的情况下生成一致的随机数。此外,还设计了一种新颖的分析方法,可以确定忠实重放所需的一组状态变量。这些变量要么保存在检查点中,要么通过一种选择性执行技术——快进,重新生成。DETrain 在 13 个 PyTorch 模型和 16 个 Tensorflow 模型上进行了评估,它可以确定性地执行这些程序,并以合理的开销从检查点进行重放,还能帮助开发人员诊断训练中的问题。

综上,在高性能分布式环境下为深度学习训练引入检查点机制以实现容错和确定性重放,可以促进深度学习训练的容错计算和确定性分析。这是一项值得继续关注和推进的技术。

5 总结与展望

本章总结了系统层检查点与应用层检查点的区别以及检查点的发展与应用,并对 HPC 环境检查点技术的未来发展方向进行了展望。

5.1 总结

系统层检查点和应用层检查点是两种不同的检查点机制,它们都可以捕获应用程序的运行时状态,并从挂起点通过保存的检查点文件进行恢复。二者的区别如表 4 所列。具体如下:

捕获层面不同:系统层检查点捕获操作系统层面资源的状态,如进程、文件系统、网络连接等;应用层检查点捕获应用程序自身的状态,如变量值、栈信息、堆内存使用等。

成本和性能影响不同:系统层检查点要保存程序的完整的进程信息,而应用层检查点只需要保存指定其保存的内容。因此,相对于应用层检查点,系统层检查点需要保存的内容更多,存储开销更大,对系统性能的影响也更大。应用层检查点的性能开销和资源消耗较小。

恢复时间不同:由于捕获和恢复的粒度不同,系统层检查点的恢复时间一般较长,而应用层检查点的恢复时间较短。系统层检查点需要重启操作系统和恢复各系统资源,应用层检查点只需要重置应用程序状态。

用户透明性不同:应用层检查点需要集成到应用程序中,在使用的过程中对于用户来说是不透明的;系统层检查点根据实现细节的不同,分为透明、不透明以及不完全透明。

复杂性不同:系统层检查点需要深入内核获取进程的信息,实现起来更复杂;应用层检查点主要是对编程语言或编程模型进行修改,或者是针对该编程语言或编程模型开发的库。

适用场景不同:系统层检查点更适用于系统级容错和防护;应用层检查点更适用于应用程序容错与弹性、调试、验证和优化等。

表 4 系统层检查点与应用层检查点的比较

Table 4 Comparison between system layer checkpoints and application layer checkpoints

区别	系统层检查点	应用层检查点
捕获层面	操作系统层面	应用程序层面
成本和性能	较大	较小
恢复时间	较长	较短
用户透明性	透明/不透明	不透明
复杂性	较复杂	更简单
适用场景	系统级容错与防护	应用程序容错/弹性/调试/验证和优化等

总体来说,系统层检查点的粒度更大,范围更广,恢复时间更长,但成本也更高,性能影响更大;应用层检查点的粒度更小,范围更窄,恢复更快,成本更低,性能影响较小。相较于应用层检查点而言,系统层检查点不需要考虑编程语言的差异,它直接和系统内核通信获取进程信息。应用层检查点会针对相应的应用程序的编程语言和编程模式进行设计和优化,在运行时状态的保存上自由度较高。两种检查点机制各有优势,可以根据具体需求和场景选择使用。许多系统会结合使用两种检查点,实现更强大的功能。

检查点技术的发展与应用主要体现在容错与弹性、调度与迁移、FPGA 调试、深度学习中的检查点这 4 个方面。整体而言,检查点技术的应用场景较广,在现代 HPC 环境中发挥着重要作用。而随着云计算、边缘计算等新型计算模式的出现,检查点技术也面临着新的机遇与挑战,未来的发展前景广阔。

5.2 展望

检查点近些年来得到了快速的发展,也取得了一定的成果和进展,然而该领域依然有许多亟待解决的问题。本文总结了以下 3 个未来值得研究的方向:

1)面向融合计算实际生产场景研发灵活易用的的检查点工具

深度学习、大数据和高性能的融合,对 HPC 环境的更精细的容错和调度提出了新的挑战。虽然现有的关于检查点技术的研究有很多,但是真正可以投入到生产运行的面向融合计算的检查点工具还很有限,尤其是国产异构计算系统。

2)检查点工具的优化

检查点工具的优化方向包括检查点可检查范围的扩大以及检查点策略的优化。目前已经有许多关于系统层检查点、应用层检查点的工具,以及基于这些工具开展的应用与优化,然而,现有的检查点工具仍然有许多不足之处。各个检查点工具受限于自身的功能以及系统架构,支持的计算任务类型有限,对一些计算任务无法完全而完整地进行检查点/恢复操作。检查点设置策略也需要优化,包括检查点检查的频率、检查的触发动机以及存储位置等,根据应用场景对检查点设置策略进行优化可以提高检查点的效率、节省系统资源。未来还需要对系统层检查点如 CRIU 增加对并行计算库的支持,增加系统层检查点对 IB 网的支持。

3)检查点的应用拓展

目前,在 HPC 环境中已经有较为丰富的关于检查点的应用的研究,主要集中在并行应用程序的容错与弹性、调度与作业迁移、FPAG 的调试以及深度学习训练的容错计算与忠实重放上。对深度学习模型而言,其训练是一个长期的迭代优化过程,通常需要运行数周甚至数月。在长期的训练过程中,理想的 GPU 资源管理和作业调度机制应能够动态响应集群状态的变化,并能够确保训练过程的高可用性,这也需要作业迁移和检查点技术的支持。基于检查点的并行应用程序的调试工具也是未来的研究方向。在深度学习中引入检查点并控制开销,处理个节点的参数状态一致性的问题,为基于深度学习的 AI 应用提供重要的高性能计算支持,尤其是随着大模型的发展,进一步扩展检查点技术的应用对高性能 AI 应用来说非常重要。

参 考 文 献

[1] KUMAR M,MOLLA A R. On the Message Complexity of Fault-Tolerant Computation: Leader Election and Agreement [C]// Proceedings of the 24th International Conference on Distributed Computing and Networking. Kharagpur India: ACM, 2023:294-295.

[2] LIN L,HUANG Y,LIN Y,et al. FFNLFD: Fault Diagnosis of Multiprocessor Systems at Local Node with Fault-Free Neighbors under PMC Model and MM * Model[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(7): 1739-1751.

[3] YOUNESS H, OMAR A, MONESS M. An Optimized Weighted Average Makespan in Fault-Tolerant Heterogeneous MPSoCs [J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(8): 1933-1946.

[4] PSISTAKIS A, CHRYSOS N, CHAI X F, et al. Optimized Page Fault Handling During RDMA[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 3990-4005.

[5] LEE Y L, LIANG D, WANG W J. Optimal Online Liveness Fault Detection for Multilayer Cloud Computing Systems[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(5): 3464-3477.

[6] PALAZZI L, LI G, FANG B, et al. Improving the Accuracy of IR-Level Fault Injection[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(1): 243-258.

[7] ANSARI M, SAFARI S, KHDR H, et al. Power-Aware Checkpointing for Multicore Embedded Systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 4410-4424.

[8] KHORGUANI A, ROPARS T, DE PALMA N. ResPCT: fast checkpointing in non-volatile memory for multi-threaded applications[C]// Proceedings of the Seventeenth European Conference on Computer Systems. Rennes France: ACM, 2022: 525-540.

[9] BEIGI M V, CAO Y, GURUMURTHI S, et al. A Systematic Study of DDR4 DRAM Faults in the Field[C]// 2023 IEEE International Symposium on High-Performance Computer Architecture(HPCA). Montreal, QC, Canada: IEEE, 2023: 991-1002.

[10] ROJAS E, PEREZ D, CALHOUN J C, et al. Understanding Soft Error Sensitivity of Deep Learning Models and Frameworks through Checkpoint Alteration[C]// 2021 IEEE International Conference on Cluster Computing(CLUSTER). Portland, OR, USA: IEEE, 2021: 492-503.

[11] BORGHESI A, MOLAN M, MILANO M, et al. Anomaly Detection and Anticipation in High Performance Computing Systems [J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(4): 739-750.

[12] ZHAO K, DI S, LI S, et al. FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks[J]. IEEE Transactions on Parallel and Distributed Systems 2021, 32(7): 1677-1689.

[13] ZOU A, LI J, GILL C D, et al. RTGPU: Real-Time GPU Scheduling of Hard Deadline Parallel Tasks With Fine-Grain Utilization[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(5): 1450-1465.

[14] MAURYA A, NICOLAE B, RAFIQUE M M, et al. Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing[C]// 2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOTS). Houston, TX, USA: IEEE, 2021: 1-8.

[15] LITZKOW M, TANNENBAUM T, BASNEY J, et al. Checkpoint and migration of UNIX processes in the Condor distributed processing system[R]. University of Wisconsin-Madison Department of Computer Sciences, 1997.

[16] HARGROVE P H, DUELL J C. Berkeley lab checkpoint/restart (BLCR) for Linux clusters[J]. Journal of Physics: Conference Series, 2006, 46: 494-499.

[17] ANSEL J, ARYA K, COOPERMAN G. DMTCP: Transparent checkpointing for cluster computations and the desktop[C]// 2009 IEEE International Symposium on Parallel & Distributed Processing. Rome: IEEE, 2009: 1-12.

[18] CRIU. CRIU[EB/OL]. [2023-07-16]. https://criu.org/Main_Page.

[19] SCR. Scalable Checkpoint/Restart(SCR) User Guide[EB/OL]. [2023-7-16]. https://scr.readthedocs.io/en/latest/#.

[20] NICOLAE B, MOODY A, GONSIOROWSKI E, et al. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale[C]// 2019 IEEE International Parallel and Distributed Processing Symposium(IPDPS). Rio de Janeiro, Brazil: IEEE, 2019: 911-920.

[21] The Open MPI Community. User-Level Fault Mitigation(ULFM) [EB/OL]. [2023-07-16]. https://docs.open-mpi.org/en/v5.0.x/features/ulfm.html.

[22] WEEKS N, LUECKE G, MARIS P, et al. Challenges in Developing MPI Fault-Tolerant Fortran Applications[C]// 2018 IEEE International Conference on Cluster Computing(CLUSTER). Belfast: IEEE, 2018: 524-531.

[23] GAMELL M, KATZ D S, KOLLA H, et al. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales[C]// SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, LA, USA: IEEE, 2014: 895-906.

[24] GAMELL M, TERANISHI K, HEROUX M A, et al. Local recovery and failure masking for stencil-based applications at extreme scales[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Austin Texas: ACM, 2015: 1-12.

[25] LEÓN B, FRANCO D, REXACHS D, et al. Analysis of parallel application checkpoint storage for system configuration[J]. The Journal of Supercomputing, 2021, 77(5): 4582-4617.

[26] WANG X, LEIDEL J D, WILLIAMS B, et al. xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing[C]// 2021 IEEE International Parallel and Distributed Processing Symposium(IPDPS). Portland, OR, USA: IEEE, 2021: 454-463.

[27] DU Y, MARCHAL L, PALLEZ G, et al. Optimal Checkpointing Strategies for Iterative Applications[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(3): 507-522.

[28] SIGDEL P, YUAN X, TZENG N F. Realizing Best Checkpointing Control in Computing Systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(2): 315-329.

[29] RAYBON G, ADAMIECKI A, CHO J, et al. Single-carrier all-ETDM 1.08-Terabit/s line rate PDM-64-QAM transmitter

using a high-speed 3-bit multiplexing DAC[C]// 2015 IEEE Photonics Conference(IPC). Reston, VA: IEEE, 2015: 1-2.

[30] DEY T, SATO K, NICOLAE B, et al. Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning[C]// 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). New Orleans, LA, USA: IEEE, 2020: 1036-1043.

[31] MO Y, XING L, LIN Y K, et al. Efficient Analysis of Repairable Computing Systems Subject to Scheduled Checkpointing[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(1): 1-14.

[32] LEÓN B, MÉNDEZ S, FRANCO D, et al. A model of checkpoint behavior for applications that have I/O[J]. The Journal of Supercomputing, 2022, 78(13): 15404-15436.

[33] ZHOU T, GAO L, GUAN X. A Fault-Tolerant Distributed Framework for Asynchronous Iterative Computations[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(8): 2062-2073.

[34] MAURYA A, NICOLAE B, RAFIQUE M M, et al. Towards Efficient Cache Allocation for High-Frequency Checkpointing [C]// 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC). Bengaluru, India: IEEE, 2022: 262-271.

[35] ANSEL J, ARYA K, COOPERMAN G. DMTCP: Transparent checkpointing for cluster computations and the desktop[C]// 2009 IEEE International Symposium on Parallel & Distributed Processing. Rome: IEEE, 2009: 1-12.

[36] PLANK J S, BECK M, KINGSLEY G, et al. Libckpt: Transparent checkpointing under unix[M]. Computer Science Department, 1994.

[37] HONG G, AHN S J, HAN S C, et al. Kckpt: checkpoint and recovery facility on unixware kernel[C]// Proceedings of the 15th International Conference on Computers and Their Applications (ISCA). 2000.

[38] TAKIZAWA H, SATO K, KOMATSU K, et al. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications[C]// 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies. Higashi Hiroshima, Japan: IEEE, 2009: 408-413.

[39] NUKADA A, TAKIZAWA H, MATSUOKA S. NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA [C]// 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. Anchorage, AK, USA: IEEE, 2011: 104-113.

[40] SHI L, CHEN H, LI T. Hybrid CPU/GPU Checkpoint for GPU-Based Heterogeneous Systems[M]// Parallel Computational Fluid Dynamics: Vol. 405. Berlin, Heidelberg: Springer, 2014: 470-481.

[41] LOSADA N, CORES I, MARTÍN M J, et al. Resilient MPI applications using an application-level checkpointing framework and ULFM[J]. The Journal of Supercomputing, 2017, 73(1): 100-113.

[42] LOSADA N, MARTÍN M J, GONZÁLEZ P. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications[J]. The Journal of Supercomputing, 2017, 73(1): 316-329.

[43] PARASYRIS K, KELLER K, BAUTISTA-GOMEZ L, et al. Checkpoint Restart Support for Heterogeneous HPC Applications[C]// 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). Melbourne, Australia: IEEE, 2020: 242-251.

[44] ZHANG Y, GUO X, JIANG H, et al. A Checkpoint/Restart Scheme for CUDA Applications with Complex Memory Hierarchy[C]// 2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. Honolulu, HI, USA: IEEE, 2013: 247-252.

[45] POURGHASSEMI B, CHANDRAMOWLISHWARAN A. cudaCR: An In-Kernel Application-Level Checkpoint/Restart Scheme for CUDA-Enabled GPUs[C]// 2017 IEEE International Conference on Cluster Computing (CLUSTER). Honolulu, HI, USA: IEEE, 2017: 725-732.

[46] GARG R, MOHAN A, SULLIVAN M, et al. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory[C]// 2018 IEEE International Conference on Cluster Computing (CLUSTER). Belfast: IEEE, 2018: 302-313.

[47] CHIU M T, YOU Y P. CLPKM: A checkpoint-based preemptive multitasking framework for OpenCL kernels[J]. Journal of Systems Architecture, 2019, 98: 53-62.

[48] CHEN G, ZHANG J, ZHU Z, et al. CRState: checkpoint/restart of OpenCL program for in-kernel applications[J]. The Journal of Supercomputing, 2021, 77(6): 5426-5467.

[49] MOHROR K, MOODY A, BRONEVETSKY G, et al. Detailed Modeling and Evaluation of a Scalable Multilevel Checkpointing System[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(9): 2255-2263.

[50] MOODY A, BRONEVETSKY G, MOHROR K, et al. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System[C]// 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, LA, USA: IEEE, 2010: 1-11.

[51] DUARTE A, REXACHS D, LUQUE E. Increasing the cluster availability using RADIC[C]// 2006 IEEE International Conference on Cluster Computing. Barcelona: IEEE, 2006: 1-8.

[52] CASTRO-LEÓN M, MEYER H, REXACHS D, et al. Fault tolerance at system level based on RADIC architecture[J]. Journal of Parallel and Distributed Computing, 2015, 86: 98-111.

[53] WONG A, HEYMANN E, REXACHS D, et al. Middleware to Manage Fault Tolerance Using Semi-Coordinated Checkpoints [J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(2): 254-268.

[54] WHITLOCK M, MORALES N, BOSILCA G, et al. Integrating process, control-flow, and data resiliency layers using a hybrid Fenix/Kokkos approach[C]// 2022 IEEE International Confe-

rence on Cluster Computing(CLUSTER). Heidelberg, Germany: IEEE, 2022: 418-428.

[55] PARASYRIS K, GEORGAKOUDIS G, BAUTISTA-GOMEZ L, et al. Co-Designing Multi-Level Checkpoint Restart for MPI Applications[C]// 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing(CCGrid). Melbourne, Australia: IEEE, 2021: 103-112.

[56] RODRÍGUEZ-PASCUAL M, CAO J, MORÍÑIGO J A, et al. Job migration in HPC clusters by means of checkpoint/restart [J]. The Journal of Supercomputing, 2019, 75(10): 6517-6541.

[57] XU H, DE VECIANA G, LAU W C, et al. Online Job Scheduling with Redundancy and Opportunistic Checkpointing: A Speedup-Function-Based Analysis [J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(4): 897-909.

[58] PRADES J, SILLA F. GPU-Job Migration: The rCUDA Case [J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(12): 2718-2729.

[59] CHAUDHARY S, RAMJEE R, SIVATHANU M, et al. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning[C]// Proceedings of the Fifteenth European Conference on Computer Systems. Heraklion Greece: ACM, 2020: 1-16.

[60] FOSTER H. Wilson Research Group functional verification study: IC/ASIC functional verification trend report[R]. White Paper. Wilson Research Group and Mentor, A Siemens Business, 2020.

[61] ASAAD S, BELLOFATTO R, BREZZO B, et al. A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation[C]// Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. Monterey California USA: ACM, 2012: 153-162.

[62] ATTIA S, BETZ V. StateMover: Combining simulation and hardware execution for efficient FPGA debugging[C]// Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2020: 175-185.

[63] ATTIA S, BETZ V. Toward Software-like Debugging for FPGAs via Checkpointing and Transaction-based Co-Simulation [J]. ACM Transactions on Reconfigurable Technology and Systems, 2023, 16(2): 1-24.

[64] ANTHONY Q, DAI D. Evaluating Multi-Level Checkpointing for Distributed Deep Neural Network Training[C]// 2021 SC Workshops Supplementary Proceedings(SCWS). St. Louis, MO, USA: IEEE, 2021: 60-67.

[65] XU X, LIU H, TAO G, et al. Checkpointing and deterministic training for deep learning[C]// Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI. Pittsburgh Pennsylvania: ACM, 2022: 65-76.

YAN Xiaoting, born in 1999, postgraduate. Her main research interests include high performance computing and cloud service.

WANG Xiaoning, born in 1981, Ph. D, associate professor, Ph. D supervisor, master supervisor. Her main research interests include high performance computing, grid computing and cloud service.

(责任编辑:杨雪敏)