CN 43-1258/TP
ISSN 1007-130X

计算机工程与科学
Computer Engineering & Science

第 42 卷第 10 期 2020 年 10 月
Vol. 42, No. 10, Oct. 2020

文章编号:1007-130X(2020)10-1774-10

# 百亿亿次级系统进程管理接口综述*

张　昆,张　伟,卢　凯,董　勇,戴屹钦
(国防科技大学计算机学院,湖南 长沙 410073)

**摘　要**:高性能计算机不断发展,系统规模日益增加,系统内包含的计算结点数、处理器核数扩展到新的水平。在超大规模系统中,并行应用程序的启动时间成为限制系统运行效率、降低系统易用性的一个重要因素。在并行应用启动阶段,利用进程管理接口为进程部署通信通道,供进程后续通信使用。在百亿亿次级规模系统中,传统进程管理接口无法在启动时快速获得通信信息,导致启动时间过长,系统性能下降。首先介绍进程管理接口在并行程序启动过程中的作用,着重介绍面向百亿亿次级系统的进程管理接口 PMIx,而后对比论述 PMIx 对于改进大规模并行程序启动的作用,分析 PMIx 在提升系统性能上做出的优化,以及未来发展方向。

**关键词**:高性能计算;进程管理接口;进程通信
**中图分类号**:TP301　　**文献标志码**:A
**doi**:10.3969/j.issn.1007-130X.2020.10.009

# A survey on the process management interface for exascale computing systems

ZHANG Kun, ZHANG Wei, LU Kai, DONG Yong, DAI Yi-qin
(School of Computer, National University of Defense Technology, Changsha 410073, China)

**Abstract**: With the continuous development of high-performance computing, the scale of the system increases constantly, and the number of nodes and processor cores in the system has expanded to a new level. Under the condition of hyperscale systems, the startup time of parallel applications becomes an important factor, which limits the system's operating efficiency and reduces the ease of use. The process management interface is used to deploy a communication channel for the process during the parallel application startup phase for subsequent communication of the process. In exascale systems, the traditional process management interface cannot quickly obtain communication information at the startup phase, resulting in long startup time and the reduced system performance. We first introduce the role of the process management interface in the parallel program startup process, focus on the process management interface PMIx for exascale systems, compare and discuss the role of PMIx in improving the startup of large-scale parallel programs, analyze the optimization of PMIx in improving system performance, and discuss future development directions.

**Key words**: high-performance computing; process management interface; process communication

## 1　引言

目前,高性能计算 HPC(High Performance Computing)系统性能已经进入百亿亿次时代,各国都向着实现百亿亿次计算性能目标不断行进,为提高经济竞争力、推进工业生产、改善人类社会生活创造更多切实效益。

* 收稿日期:2020-06-10;修回日期:2020-07-23
基金项目:国家重点研发计划(2018YFB0204301);国家数值风洞项目(NNW2018-ZT6B13)
通信地址:410073 湖南省长沙市国防科技大学计算机学院
Address: School of Computer, National University of Defense Technology, Changsha 410073, Hunan, P. R. China

进程管理是高性能计算的一个组成部分,在结点数量较少的计算系统中,进程管理的可扩展性对应用程序运行没有太大影响。但是,随着系统规模扩大,传统进程管理已经不能满足海量处理器核的需求,迫切需要一种可扩展的管理方式,且可以提供进程交换信息的机制。为了解决该问题,在进程管理中引入进程管理接口 PMI(Process Management Interface),用于在启动阶段部署进程的通信方法。引入后的进程管理包括 3 个部分[2]:并行编程库(例如 MPI 库)、PMI 库和进程管理器。进程管理器是一个逻辑上的中心进程,用于管理进程启动与进程间信息传递。PMI 库通过 API 接口提供进程交换信息机制。并行编程库可以通过 PMI API 来访问进程管理器提供的启动进程、初始化进程等各类功能。

IBM 发布的报告[1]指出,实现百亿亿次级计算系统有 5 大瓶颈,其中一项就是通信瓶颈。为了突破这一瓶颈,在进程之间快速建立通信机制成为一项重要的研究课题。百亿亿次系统内包含数万结点与近千万处理器核,以往 PMI 建立通信的方法无法满足进程快速启动的要求,大规模并行应用的启动时间过长,成为限制系统运行效率、降低系统易用性的一个重要因素。

为了解决百亿亿次级系统的启动问题,进程管理接口发展出 PMIx(Process Management Interface-Exascale)的版本,以实现在 30 s 内启动和连接大规模应用程序进程的总体目标[3]。PMIx 定义了新的并行应用程序启动顺序,并针对大规模应用程序提出新的建立通信机制,力求缩短启动时间,提高系统性能。

本文主要探讨百亿亿次级计算性能目标下,进程管理接口启动大规模并行应用程序的有效性,对比 PMI-1、PMI-2 和 PMIx 的工作机制与特点,分析 PMIx 对于作业快速启动、系统高效运行的重要作用,以及在提升性能上做出的优化,并讨论未来的发展方向。

本文第 2 节介绍进程管理接口的发展过程和大规模并行应用程序启动过程,分析进程管理对于提高系统运行效率的重要意义;第 3 节分别介绍传统的并行应用程序启动顺序和 PMIx 模式下的并行应用程序启动顺序,对比分析 PMIx 在改进启动过程中所发挥的作用;第 4 节介绍为了提高系统性能,目前对 PMIx 所做的优化研究;第 5 节从启动并行应用程序期间和运行时控制 2 个方面对未来 PMIx 的发展方向进行了说明。

## 2　背景

### 2.1　进程管理接口介绍

在现代高性能计算系统中,一个并行计算任务包含多个并行作业,每一个并行作业又包含一个或多个并行应用程序,而每一个应用程序则是一组二进制可执行文件的实例化。在系统中启动一个并行应用程序主要包含以下几步[4]:

(1)部署运行时环境。

该阶段是在分配有作业的计算结点集上部署分布式的运行时环境 RTE(Running Time Environment)。目前在大多数软件堆栈中,该阶段通常在分配的每一个计算结点上都创建一个 RTE 实例,用于后续生成和管理进程。

(2)启动应用程序进程。

在分配有作业的计算结点上,RTE 实例生成相应进程,并执行进程启动。

(3)获取预定义环境信息。

在此阶段,RTE 将自身已知的作业级信息提供给应用程序进程,如进程的 rank 信息、进程总数信息、共享同一计算结点的进程列表等。

(4)获取动态环境信息。

在该阶段,进程需要交换彼此特定于应用程序的数据信息。这些是不包含在阶段(3)中但进程必须知道的信息,例如通信寻址信息,通过定义的全局操作,进程获得此类信息。

目前,大多数的高性能计算软件堆栈都采用进程管理接口来完成阶段(3)与阶段(4)的工作。并行应用程序进程通过 PMI 定义的 API 与 RTE 通信。PMI 将信息组织成键值数据库的形式,通过 Put、Get 和同步原语访问数据库。

在并行应用程序启动过程中,最耗时的阶段为阶段(4)。当面对百亿亿次级系统时,进程数目庞大,所有进程彼此之间交换信息会花费大量时间,这对于实现快速启动应用程序的目标十分不利。而 PMIx 则集中解决此阶段问题,力求寻找获取动态信息的新方法,尽可能将阶段(4)中所需信息转移到阶段(3)中。通过减少进程交换信息操作的时间消耗,即使进程规模庞大,PMIx 也可以达到快速启动应用程序的目标,促进系统高效运行。

### 2.2　进程管理接口发展过程

随着“异构 MPI”项目的研究,各国致力于开发可应用于多台不同类型计算机上的 MPI,这就

需要MPI程序能够处理异构的通信协议、安全性机制和调度程序等[5],这激发工作人员研究一类定义在MPI中使用的通用API,用来执行各类进程管理系统中的启动、监视和控制进程的功能[6]。

第1代PMI(PMI-1)被广泛应用在MPICH2[7]及其派生的MPI实现中[8],其最初设计目标是利用一个接口减少进程管理器与MPI实现之间的耦合。PMI-1主要包含2个部分,一部分旨在协助MPI初始化过程中的引导活动,另一部分旨在支持MPI中的动态进程管理功能。PMI-1设计人员使进程管理器只提供MPI进程交换信息的通用机制。该机制由一个键值存储KVS(Key-Value Store)组成。经过适当的同步操作,一个进程放入KVS的内容可以由另一个进程读出。在PMI-1中,一个简单的集体屏障功能保证了该类同步。一旦所有进程经过屏障操作,则认为所有数据都已写入;并且一旦经过屏障操作释放进程,进程就可以读取所有数据。

但是,将PMI-1应用于现代高性能计算系统上时,却暴露出PMI-1的局限性。虽然在PMI-1中有一个键值数据库用于存取,但进程管理器无法向数据库中添加系统信息,导致MPI进程不能从进程管理器中查询此类信息。另外,PMI-1不能保证线程安全,在给定的PMI连接上,同时只能执行一个请求。并且,PMI-1中缺乏故障处理机制,无法在进程发生故障时重新生成进程。为了解决以上问题,开发人员设计出第2代进程管理接口(PMI-2)。

在PMI-2中,针对PMI-1的以下几个方面进行了改进[2]:

(1)增加查询功能。

PMI-2中加入了进程管理系统预定义的键。通过该类键,进程管理器将系统信息直接添加到键值库中,MPI程序可以获取与MPI进程相关的布局信息。

(2)扩大数据库信息范围。

PMI-1采用的是扁平化的数据库,一个MPI进程不能限制它添加到数据库中的键值的范围。在此情况下,加入的信息都归纳为全局信息,进程无法单独获取仅与本地计算结点相关的子集信息。为了解决这个问题,PMI-2中引入了结点级别与全局级别的数据库信息范围,进程管理器可以更高效率地提取信息。

(3)增加线程安全。

PMI-1面临多线程情况时,MPI实现调用PMI-1需使用适当锁机制保护。但是,PMI-1中的锁机制是粗粒度的,当PMI库的一个查询请求得到了进程管理器的响应后,其他线程就无法在该套接字上通信。PMI-2则对PMI-1中线程安全的限制进行了修改,MPI实现不需要确保一次仅有一个线程调用PMI。

(4)改善容错机制。

当故障发生时,PMI-1没有提供任何的进程重生机制进行处理。PMI-2提供了故障处理方法,利用重生的新进程取代同一进程组中的故障进程。

随着超级计算机的发展,高性能计算系统面临多层次的大规模并行处理,为了满足百亿亿次级系统的要求,PMI社区开发出PMIx接口[3,9]。PMIx是专门设计用于支持百亿亿次级系统的进程管理接口,该接口以编程模型不可知的方式抽象出管理大规模应用程序和监视进程所必需的功能。

在PMI-2的基础上,PMIx在功能上进行了进一步完善。相对于PMI-2中的数据库信息范围,PMIx将信息作用域增加到进程本地范围、结点本地范围、远程范围和全局范围。另外,在PMI-1与PMI-2中,键值的数据类型仅定义为字符串,PMIx则扩展了数据表示,可对更多类型数据进行存储。PMIx针对同步原语Fence和Get原语都加入了非阻塞版本,以降低操作延迟。同时,PMIx针对稀疏通信情况进行优化,提供按需数据交换模式,缩短通信时间。更为关键的是,针对启动大规模并行应用程序,PMIx定义了新的启动顺序,力求降低传统应用程序启动过程时间成本,加快程序启动。

## 3 大规模并行应用程序启动过程优化

随着高性能计算处理问题规模不断增大,传统并行程序启动过程显露出弊端。大规模应用程序的启动时间成为一个不可忽视的问题,PMIx为了满足百亿亿次级系统的要求,对传统启动顺序进行调整,力求快速启动应用程序。表1对比分析了传统应用程序启动顺序与PMIx模式下的启动顺序。下面将对PMIx优化并行应用程序的启动过程进行详细介绍。

### 3.1 传统并行应用程序启动顺序

传统并行应用程序的启动顺序大致分为以下6个不同的阶段,如图1[10]所示。

第1阶段:用户将作业组织成脚本或交互会话请求的形式,其中包含完成作业所需的资源类型与

Table 1　Analysis of start sequence of parallel application

表 1　并行应用程序启动顺序分析

| | 传统并行应用程序启动顺序(6 个阶段) | PMIx 模式下并行应用程序启动顺序 |
|---|---|---|
| 第 1 阶段 | 用户提交作业请求,WLM 将用户请求加入处理队列 | 用户提交作业请求,WLM 除了将请求加入队列之外,还利用 PMIx 接口从文件系统中查询用户所有可执行文件与支持库文件 |
| 第 2 阶段 | WLM 为作业分配资源,计算结点生成用户进程(存在问题:大量进程同时检索文件系统,引发"库检索风暴") | WLM 为作业分配资源<br>新调度开始之前,利用 2 个 PMIx 接口进行预处理:<br>(1)重定位检索文件,缓解文件系统压力;<br>(2)将作业描述传递到子系统,获取各类子系统的支持信息(其中包含网络系统对进程通信地址提供的支持) |
| 第 3 阶段 | 进程发现本地信息提供给结点代理 | 计算结点守护进程获得子系统支持信息,并注册到本地 PMIx 服务器中,生成用户进程 |
| 第 4 阶段 | 全局结点代理间信息交换(启动过程最耗时阶段) | 用户进程连接 PMIx 服务器,获得所有本地信息,即进程直接获得网络系统的通信配置信息,可以开始通信 |
| 第 5 阶段 | 结点根据获取的信息构建本地通信配置 | |
| 第 6 阶段 | 全局屏障,保证进程可通信(第 2 个耗时阶段) | |
| 对比总结 | PMIx 模式通过将网络系统的通信信息注册在计算结点 PMIx 服务哭器中,进程与 PMIx 服务器连接时可直接获取此类信息,从而消除全局交换与屏障阶段,加快进程启动 | |

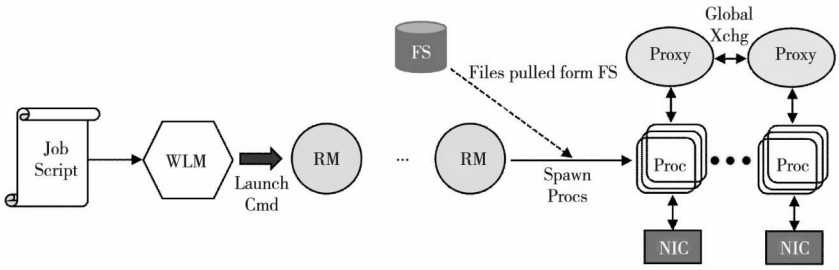


Figure 1　Traditional start sequence

图 1　传统启动顺序

数量描述。而后,用户将作业提交给工作负载管理器 WLM(WorkLoad Manager),等待作业执行。

第 2 阶段:WLM 根据作业优先级、可支配资源数目和资源调度算法,将资源分配给作业,并将分配结果通知控制资源分配的资源管理器 RM(Resource Manager)守护进程[11,12]。RM 守护进程接收通知后,一方面在文件系统 FS(File System)里获取应用进程二进制可执行文件,实例化作业进程,图 1 中"Spawn Procs"代表生成进程过程;另一方面从文件系统中加载实例化进程所需的动态依赖库。

可是,在进程数目庞大的情况下,由于大量结点的多个进程会同时进行请求库操作,极易引发"库检索风暴"问题,导致文件系统压力过大。

第 3 阶段:进程实例化完成后,需要构建进程之间的通信机制。每个进程都要提供交互所需的本地通信信息,例如进程 rank 和通信端点地址。当进程获取自身通信信息后,需要将该类信息发布到计算结点的本地代理上。

第 4 阶段:结点的本地代理发现所有进程信息后,利用网络上定义的全局操作[13],完成结点与结点之间进程通信信息的全局交换过程,这一过程通常称为名片交换或 BCX(Business Card Exchange),图 1 中"Global Xchg"代表该阶段。这个阶段是整个并行应用程序启动阶段中最耗时的部分。大部分的研究都关注如何开发更高效方法,以减少此过程的时间开销。

第 5 阶段:结点完成信息全局交换后,需要根据所获得的通信信息构建通信通道。在这一阶段,程序库会基于本地代理上的信息进行基础结构的组装,例如将通信通道映射到其他进程。这个阶段的时间花费是从本地结点的代理服务器上检索连接和其他所需信息所花的时间,以及每个进程配置其本地网络接口卡 NIC(Network Interface Cards)所需的时间。

第 6 阶段:执行全局的屏障操作,用来确认结点上的所有进程都已经构建完成通信结构,可以进行通信。虽然这一过程并没有对数据进行处理操作,但此阶段是除全局交换外,另一个耗时最多的阶段。在该阶段,需要观察计算结点上的每一个进

程是否都成功完成通信设置,另外还需要确认本地高速网络也处于接收信息的状态。只有经过该阶段的屏障,才可以认为所有进程能够进行交互。

随着并行应用程序进程数目的大幅度增加,传统的启动过程中第4阶段进程交换信息和第6阶段确认进程状态过程都会花费过多时间。为了实现在短时间内启动大规模应用程序,PMIx对传统应用程序启动顺序进行改进,力求消除全局交换的时间成本。

### 3.2 PMIx模式下并行应用程序启动顺序

传统启动过程中耗时最大的部分是全局交换信息与执行全局屏障阶段,PMIx通过提供新的接口,调整并行应用程序的启动顺序,力求去除以上2个阶段。修改后的并行应用程序理想启动顺序如下所示:

第0阶段:该阶段是针对系统资源进行的汇总过程,技术上来讲并不是启动过程的一部分,但资源目录(包括交换机、网络和结点拓扑等)对于应用程序调度以及有效完成启动过程至关重要。在该过程中,PMIx以动态方式处理资源,并对资源目录实时更新,从而保证启动过程的顺利完成。

第1阶段:用户将作业组织成脚本或交互会话请求的形式,其中包含完成作业所需要的资源类型与数量描述。而后,用户将作业提交给WLM,等待作业执行。在该阶段,WLM除了将请求记录到适当的队列中,还使用PMIx接口将请求传递到文件系统,以获取所有用户标识的可执行文件与支持库。

第2阶段:WLM根据作业优先级、可支配资源数目和资源调度算法,将资源分配给作业。分配通常发生在当前处理作业通知完成,启动末尾操作之前。因此,在新作业调度开始之前,有一定的时间对资源分配进行预处理。

PMIx利用这段时间完成2个重要步骤。首先,它提供一个接口,WLM可以通过该接口通知FS分配情况,并将在初始阶段确定的所需文件列表传递给FS。若此时存在可用缓存,文件系统将启动进程用到的文件缓存到与分配结点相关内存位置,通过定位所需文件,缓解文件系统检索压力。

其次,WLM利用另一个PMIx接口将分配情况和作业描述都传递给子系统,并接收一个"blob",其中包含每个子系统对作业的支持信息(例如网络系统对进程通信的支持信息)。标记返回值来标识哪些值直接提供给结点上的本地资源,以及哪些作为环境变量传递给应用程序进程,环境变量在由特定进程初始化时会被资源识别。

第3阶段:缓存所需文件并获得网络配置后,WLM将"blob"以及分配信息和资源目录信息传递给已分配结点上的RM守护进程。在派生本地进程之前,每个RM守护进程都可以利用PMIx接口将配置信息传递给本地资源。RM守护进程向本地PMIx服务器注册应用程序和本地进程,从而为进程提供与应用程序和本地环境有关的信息。完成操作后,RM守护程序将派生应用程序进程。

第4阶段:在进程初始化期间,每个进程调用PMIx_Init函数连接到本地PMIx服务器。PMIx服务器通知RM进程连接,并返回本地RM守护进程在注册期间提供的所有信息。

至此,启动序列完成,进程已经获取网络系统的通信支持,可以自由通信。

但是,以上过程只是PMIx提出的一个理想启动顺序,目前还没有在进程管理系统中完全实现。要完全消除传统启动过程中全局交换和全局屏障阶段,需要对系统管理堆栈中的网络库与相应的事件处理机制做进一步改进[14,15]。

目前,PMIx提供了2个可用于提高启动性能的方法[3,9]。首先,在BCX操作中删除本地RM守护进程提供给每个进程的数据,从而大大减少BCX操作中包含的数据量,减少全局交换信息的时间。另外,PMIx提出一种直接名片交换DBCX(Direct Business Card Exchange)的替代连接方法,该方法根据需要获取所需进程的名片来代替BCX操作。每个进程发布的数据都缓存在本地PMIx服务器上,当进程向特定进程请求名片时,进程首先检查本地PMIx共享内存段,以查看数据是否已可用。如果不可用,则将请求传递到本地服务器,服务器随后请求主机RM守护进程从指定进程的PMIx服务器中检索数据。然后,本地PMIx服务器将数据提供给请求的客户端进程。虽然DBCX通过消除BCX操作允许更快的启动速度,但检索特定进程名片信息的效率较低。因而,DBCX更适合稀疏通信的环境,对于密集通信,还存在需要改进的地方。

## 4 进程管理接口性能优化分析

PMIx启动过程中耗时最长的部分是全局交换信息、建立通信机制的过程。为了快速启动大规模并行应用程序,目前对于PMIx的研究,大部分都集中在改进全局交换、建立大规模进程通信通道

方面,研究者们针对 PMIx 启动过程中的各个阶段做出不同方面的优化。本节从下列几个方面介绍目前对于 PMIx 所进行的优化研究:(1)优化 PMIx 数据库伸缩性;(2)优化数据表示形式;(3)优化 PMIx 运行时环境。

### 4.1 优化 PMIx 数据库伸缩性

在进程管理接口提供的交互方式里,资源管理器与应用程序的信息交换是通过键值数据库 KVDb(Key-Value Database)来进行的。应用程序进程通过 PMI 提供的 Put 原语将数据提交到 KVDb 中,利用特定的同步原语使数据库接收所有进程 Put 的数据,而后进程调用 Get 原语从 KVDb 中获取所需信息。

结点内的 KVDb 访问是影响 PMI 性能的主要原因之一,主要体现在 Get 原语的延迟问题。Chakraborty 等人[16] 对资源管理器 SLURM 中 PMI-2 实现的可伸缩性进行分析,将客户进程与服务器的通信定义为影响通信效率的一个关键限制,并且他们也探讨了利用共享内存机制实现 KVDb 的优势,主要包含以下几点:

(1)内存消耗可伸缩:传统方式在应用程序的每一个进程上都复制一次 KVDb 的内容,导致占用过多内存。利用共享内存机制,只需要在计算结点上存储一次 KVDb 内容,可减少存储量。

(2)更优的并行性:共享内存允许客户端在不涉及服务器的情况下独立访问 KVDb。

(3)延迟更低:操作被允许可以按照 CPU 速度进行处理。

通过对 PMI 数据库伸缩性进行优化,从而降低 Get 原语的延迟,加快进程数据访问,提升资源管理器性能。与 PMI-1 和 PMI-2 相比,PMIx 提供了更优的数据访问能力,一方面对作业级信息进行扩展,资源管理器可以直接获得作业环境的信息;另一方面,PMIx 允许按需获取信息,在 PMIx_Get 中可以从 KVDb 中获得指定进程的信息。

PMI 中原有的锁机制是 POSIX 线程[17] 中的 Read/Write locks(RW-locks)锁机制,该类锁的可伸缩性十分有限。为了进一步减少 PMIx_Get 操作延迟,Polyakov 等人[18] 基于文献[19]中的静态算法在 PMIx 的 2.1 版本中提出 2N-lock 锁机制。通过引入信号锁解决静态算法中写入器易饿死的问题,信号锁不易被读取器获取,但可以被写入器获取。写入器需要首先获得所有信号锁,然后获取读取锁进行写入操作。

另外,Polyakov 等人[18] 发现 PMIx_Get 中的线程转换也是影响性能的另一个限制因素。由于在 PMIx 2.1 版本的参考实现里,共享内存组件无法访问 PMIx 实现的全局状态,作者提出了新的线程安全转换方案:在 PMIx_Get 获取所需数据后,再进行线程 shift 操作,从而避免读写数据过程中的线程 shift 操作带来的延迟。对比原有实现,新提出的 PMIx 主机通信的锁定与线程安全方案,在 168 PPN(Processors Per Node)情况下,最高可减少 PMIx_Get 操作 66 倍的延迟。

### 4.2 优化数据表示形式

这一部分的优化是针对 PMIx 启动过程中全局交换信息阶段的数据量大小进行的,通过精简交换信息的数据量,去除不必要的重复信息,从而缓解数据交换的压力,加速信息交换过程。并且,更精简的数据表示形式也可以减少存储空间占用,这对于大规模并行程序来说效果更为明显。

表 2 针对 PMIx 启动过程中涉及到的在数据表示形式方面所做的不同优化方式的效果进行了总结,简要介绍了各类方法的优化对象和优化方式,对比优化前与优化后数据表示形式的字节大小,分析所提出方法的优化效果,本节后续将对各类优化方法的具体内容进行详细介绍。

#### 4.2.1 优化通信地址信息

Polyakov 等人[20] 使用基于 UCX(Unified Communication X)通信的 PMIx 高性能软件栈对数据布局进行优化。UCX 是一个高级通信库,其地址信息中包含进程建立连接过程中用到的所有信息,如通信地址、地址长度、通道特征等。Polyakov 等人[20] 通过去除地址表示中的通道特征和地址长度等通用信息,将通信实体地址信息大小从 51 B 降低到 21 B,优化了 60%。

#### 4.2.2 优化进程交换数据表示

在 PMIx 中,进程交换全局信息时,通过调用运行时环境的 Fence 函数确保 KVDb 的全局信息同步,这是整个并行应用程序启动过程中最耗时的部分[21]。Fence 函数通过交换每个运行结点上的数据块来保证信息同步,数据块由头部信息与进程组件信息组成,进程组件信息按照进程 rank 排序,每一个进程组件由进程标识符与进程键值对组成。为了减少交换的数据量信息,Polyakov 等人[20] 对数据块中各部分进行了如下优化:

(1)优化进程标识符表示。

Table 2 Summary of data representation optimization mechanisms

表 2 数据表示形式优化机制总结

| 优化对象 | 优化方式 | 优化前 | 优化后 | 效果对比 |
|---|---|---|---|---|
| UCX 通信地址 | 去除 UCX 地址表示中的通用信息 | 通信实体地址信息大小为 51 B | 通信实体地址信息大小为 21 B | 地址信息大小优化达 60% |
| 进程标识符 | 利用参与集体操作的进程的序列号标识进程 | 进程标识符由进程 namespace 与 rank 标识 | 进程标识符用一个 4 B 的整型序列号标识 | namespace 字段在不同运行时环境中长度不同:在 Open RTE 中为 10 B,在 SLURM 中至少为 21 B。优化后只需 4 B |
| 键值对表示 | 建立 key 的索引表 | 对于 $N$ PPN 情况,若 key 大小为 $s$ 个字节,另需 4 个字节说明字符串长度,则 key 共需占用 $N*(s+4)$个字节 | 对于 $N$ PPN 情况,若 key 大小为 $s$ 个字节,另需 4 个字节说明字符串长度,则 key 共需占用 $(s+4)+4*N$个字节 | 对比减少占用 $s*(N-1)-4$ 个字节 |
| 数值编码方式 | 采用 Little-Endian Base 128 编码方式 | 以 value 字段为例,不论 value 真实大小,统一采用 uint64 数据类型,内存占用[21;256] B | 仅把需要字节打包到缓冲区 | 减少因数据类型定义引发的内存空间浪费 |

一直以来,进程标识符是由进程 namespace 与 rank 组成。namespace 字段由运行时环境所分配,在不同的运行时环境中,该字段的长度是不同的。在 Open RTE(Run Time Environment)环境中,namespace 字段大小为 10 B,而在 SLURM 中,该字段至少为 21 B。Polyakov 等人对 PMIx_Fence 进行分析发现,参与同步操作的进程必须是同一 namespace 中的,这就表示参与同步操作的进程可组成一个固定序列,只需用一个大小为 4 B 的序列号标识进程即可,减少了进程标识符的内存占用。

(2)优化键值对表示。

在 2.1 版本及以前的 PMIx 参考实现里,一个键值对中 key 的表示形式为(namespace,rank,key-name),并且 PMIx_Get 原语可以识别请求 key 所在的进程的 rank 值,这导致相同的 key 可能在数据块中重复出现。本文通过建立关于 key 的索引表,并只用 key 在表中的整数索引(大小为 4 B)标识 key 的位置,对键值对表示进行优化。对于每个结点上 $N$ 个进程(PPN 为 $N$)的情况,若 key 字段大小为 $s$ 个字节,另外需要 4 个字节来表示 key 的长度,则一个 key 共需占用 $N*(s+4)$个字节;引入索引表后,上述情况只需占用$(s+4)+4*N$个字节,对比减少占用 $s*(N-1)-4$ 个字节,对于高性能计算软件堆栈,优化效果明显。

(3)优化数值编码方式。

在 2.1 版本及以前的 PMIx 参考实现里的数值表示中,为了防止数据溢出,通常采用一个相比数值实际大小要大很多的数据类型来表示数据。例如对于键值对中的 value 字段,不论键值真实大小,统一采用 uint64 数据类型定义键值,在内存中占用[21;256] B。这种方法在一定程度上造成了空间浪费,当进程数目较多时,浪费更为明显。本文采用 Little-Endian Base 128(LEB128)[22] 编码方式,仅把所需要的字节打包到缓冲区中,避免了数据类型造成的多余空间浪费,优化了数据占用空间。

### 4.3 优化 PMIx 运行性时环境

针对分布式计算运行时环境的问题,目前已经有了一系列的研究,每一类方法都集中解决一类特定问题。MPICH 提供了多种运行时环境[23],例如 Hydra[24]、MPD (MultiPrupose Daemon)[25]、Gforker 等。Hydra 可以很好地扩展单个结点上的大量进程,与混合编程模型可有效交互;虽然 Hydra 可以检测和报告 MPI 进程失败,但无法处理故障进程。MPD 通过一个环形拓扑连接结点,然而 MPD 弹性差,如果 2 个结点故障会将所有结点分成 2 个单独的结点组,从而阻止故障结点的通信;另外,MPD 也不具备可伸缩性。

OPEN RTE[26] 是 OPEN MPI[27] 的运行时环境,用于启动、监视和杀死并行作业,通过各种拓扑连接守护进程,但是通信并不可靠。PRETE 运行时环境是 PMIx 规范的参考实现,技术上属于 OPEN RTE 的一个分支,继承了 OPEN RTE 中启动和监视 MPI 作业的大部分功能,能够部署各种各样的并行应用程序和工具。

针对不同的运行时环境,在 PMIx 中可进行不同的优化措施,目前已有的针对运行时环境优化的研究有:

Polyakov 等人[20] 针对 SLURM[28] 运行时环境中 PMIx_Fence 操作进行优化,将 PMIx_Fence 与 MPI Allgatherv[29] 操作对比分析,采用调整后

的 Bruck 级联算法作为 Fence 操作。作者在 108 个结点上进行实验,以 SLURM 18.08 作为运行时环境,利用提出的集体算法与原有的树形 Fence 操作、环形 Fence 操作进行实验对比。结果表明,在 108 个结点上,当消息的规模从 20 字节扩展到 214 字节时,相比树形算法和环形算法,集体算法有效降低了 Fence 操作延迟。

为了在系统上有效运行计算任务,处理系统故障具有重要意义[30]。Zhong 等人[31]针对 PMI-x 的参考运行时环境(PRRTE)中的故障处理进行研究,实现一种高效运行时故障检测,每个参与者只在可恢复的环拓扑之后观察单个对等点,利用多个重叠的拓扑进行优化检测与传播,最小化故障检测开销并保证框架的可伸缩性,加快进程故障处理。作者提出的方法权衡了系统开销、检测效率和检测风险之间的复杂关系,既可以承受结点与故障检测的高频特性,同时具有较低程度的拓扑结构,为在大型系统上有效进行故障管理提供了新的思路。

### 4.4　进程管理接口优化机制效果对比

4.1～4.3 节中介绍了目前 PMIx 在数据库伸缩性、数据表示形式、运行时环境 3 方面所做的优化研究,表 3 对这 3 类优化机制效果进行了综合分析,探讨每一类优化机制的侧重点方向,通过对比为未来优化方向提供研究思路。

Table 3　Comparison and analysis of different PMIx optimization mechanisms

表 3　不同 PMIx 优化机制效果对比分析

| 优化机制 | 优化方向 | 优化特点 | 目前已有优化效果 |
|---|---|---|---|
| 优化数据库伸缩性 | 优化结点内的 KVDb 访问,主要操作体现为降低 PMIx_Get 原语的延迟 | 对 PMIx_Get 操作过程进行分析,针对过程中涉及到的可优化部分进行改进 | 优化锁机制以及调整线程转换阶段,降低 PMIx_Get 操作延迟,加快进程对 KVDb 的数据访问 |
| 优化数据表示形式 | 减少数据表示占用的内存空间 | 结合实际环境,尽可能避免数据表示中出现重复信息的冗余,减少已知信息的内存占用 | 优化 UCX 通信实体地址表示,提高内存效率;<br>优化全局交换的数据块信息表示形式,减少交换数据量,加快进程通信 |
| 优化运行时环境 | 结合不同的运行时环境特点,提出针对性优化措施 | 分布式运行时环境的特点各有不同,可针对实际环境中的特定问题,进行有目标的优化 | 在 SLURM 中提出新型集体操作算法,有效降低 Fence 操作延迟;<br>在 PRETE 中提出高效故障检测技术,加快进程故障处理 |

## 5　结束语

为了早日实现百亿亿次计算目标,PMIx 社区与各国研究人员一直致力于改进 PMIx,以实现在更短时间内启动大规模进程的启动目标。本节主要从 PMIx 对于并行应用程序启动期间和运行时控制 2 个方面对 PMIx 未来的发展方向进行介绍。

### 5.1　并行应用程序启动期间

并行应用程序启动期间,在所有计算结点上实例化可执行文件及其依赖库的时间是影响启动过程的重要因素,当大规模进程同时请求加载一个依赖库时,容易引发“库检索风暴”问题。

目前,PMIx 正在定义新的 API,优化可执行文件的实例化。为了缓解文件系统的检索压力,未来 PMIx 正致力于通过预先定义的接口对作业脚本进行分析,提前获得所执行作业的动态依赖库信息,调用脚本级的指令提前发现所需的依赖文件,预定位到存储管理器中,并利用二进制文件执行动态库的加载。

在进程建立通信阶段,探索新的进程通信方式,当前发布的 PMIx 3.0 版本提供了为进程分配静态端口的函数。在未来的资源管理器中,可以利用此类接口取消进程动态分配通信端口的过程,从而消除全局交换信息阶段带来的时间成本问题。

### 5.2　运行时控制

目前 PMIx 对于运行时控制的优化主要集中于对存储数据方面的控制能力,例如对于并行应用程序数据重定位或存储策略的能力,目前这方面的研究还处于起步阶段,在未来还有很大改进空间。当前,PMIx 正致力于发展以下几个方面:

(1)增强并行应用程序进程向资源管理器汇报数据存储情况,并将当前系统可用的资源实时更新通知到资源管理器。

(2)加强并行应用程序的进程向资源管理器查询当前数据的存储情况,快速获取进程所需数据的位置以及当前资源管理器存储方式等信息。

(3)进一步加强 PMIx 对于事件的处理机制,利用触发器及时激发事件进行事务处理;提高故障处理能力,合理应对系统运行过程中可能出现的各类问题,加强对于数据库更新的管控能力。

## 参考文献:

[1] Agerwala T. Challenges on the road to exascale computing [C]// Proc of the 22nd Annual International Conference on Supercomputing(ICS'08), 2008:2.

[2] Balaji P, Buntinas D, Goodell D, et al. PMI:A scalable parallel process-management interface for extreme-scale systems [M]// Recent Advances in the Message Passing Interface.

Berlin, Heidelberg:Springer Berlin Heidelberg, 2010:31-41.

[3] Castain R H, Solt D, Hursey J, et al. PMIx:Process management for exascale environments[C]// Proc of the 24th European MPI Users' Group Meeting (EuroMPI '17), 2017:Article No. 14.

[4] Yu W K, Wu J S, Panda D K. Fast and scalable startup of MPI programs in InfiniBand clusters[M]// Lecture Notes in Computer Science. Berlin, Heidelberg:Springer Berlin Heidelberg, 2004:440-449.

[5] Cheatham T E, Roe D R. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis [J]. Computing in Science & Engineering, 2015, 17(2):30-39.

[6] Toonen B, Ashton D, Gabriel E, et al. Interfacing parallel jobs to process managers[C]// Proc of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001:431-432.

[7] Buntinas D, Mercier G, Gropp W. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem[M]// Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg:Springer Berlin Heidelberg, 2006:86-95.

[8] Huang W, Santhanaraman G, Jin H W, et al. Design of high performance MVAPICH2:MPI2 over InfiniBand[C]// Proc of the 6th IEEE International Symposium on Cluster Computing and the Grid(CCGRID'06), 2006:43-48.

[9] Castain R H, Hursey J, Bouteiller A, et al. PMIx:Process management for exascale environments[J]. Parallel Computing, 2018, 79:9-29.

[10] Los Alamos National Laboratory. Science at the petascale: Roadrunner results unveiled[EB/OL]. [2009-10-26]. https://www.sciencedaily.com/releases/2009/10/091026125535.htm.

[11] Liao X K, Pang Z B, Wang K F, et al. High performance interconnect network for Tianhe system [J]. Journal of Computer Science & Technology, 2015, 30(2):259-272.

[12] Pang Z B, Xie M, Zhang J, et al. The TH Express high performance interconnect networks[J]. Frontiers of Computer Science, 2014, 8(3):357-366.

[13] Shamis P, Graham R, Venkata M G, et al. Design and implementation of broadcast algorithms for extreme-scale systems[C]// Proc of 2011 IEEE International Conference on Cluster Computing, 2011:74-83.

[14] Tsaregorodtsev A, Garonne V, Stokes-Rees I. DIRAC:A scalable lightweight architecture for high throughput computing[C]// Proc of the 5th IEEE/ACM International Workshop on Grid Computing,2004:19-25.

[15] Broquedis F, Clet-Ortega J, Moreaud S, et al. Hwloc:A generic framework for managing hardware affinities in HPC applications[C]// Proc of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010:180-186.

[16] Chakraborty S, Subramoni H, Perkins J, et al. SHMEMPMI:Shared memory based PMI for improved performance and scalability[C]// Proc of 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016:60-69.

[17] Portable Applications Standards Committee, IEEE Computer Society. Standard for information technology— Portable operating system interface (POSIX) base specifications: IEEE Std 1003.1[S]. New York:IEEE Computer Society, 2004.

[18] Polyakov A Y, Ladd J, Shipunova E, et al. A scalable PMIx database[C]// Proc of EuroMPI 2018 Conference (EuroMPI'18), 2018:10-11.

[19] Hsieh W C, Weihl W E. Scalable reader-writer locks for parallel systems[C]// Proc of the 6th International Parallel Processing Symposium,1992:656-659.

[20] Polyakov A Y, Karasev B I, Hursey J, et al. A performance analysis and optimization of PMIx-based HPC software stacks[C]// Proc of the 26th European MPI Users' Group Meeting(EuroMPI'19), 2019:Article No. 9.

[21] Process management interface for exascale(PMIx) standard (version3.1)[EB/OL]. [2019-02-20]. https://github.com/pmix/pmix-standard/releases/download/v3.1/pmix-standard-3.1.pdf.

[22] The DWARF debugging standard (Version 5)[EB/OL]. [2019-02-04]. http://www.dwarfstd.org/.

[23] Bosilca G, Herault T, Rezmerita A, et al. On scalability for MPI runtime systems[C]// Proc of 2011 IEEE International Conference on Cluster Computing, 2011:187-195.

[24] Mathematics and Computer Science Division Argonne National Laoratory. Hydra pocess management framework [EB/OL]. [2014-06-24]. https://wiki.mpich.org/mpich/index.php?title=HydraProcess_Management_Framework.

[25] Butler R M, Gropp W, Lusk E. A scalable process-management environment for parallel programs[M]// Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg:Springer Berlin Heidelberg, 2000:168-175.

[26] Castain R H, Woodall T S, Daniel D J, et al. The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing[M]// Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg:Springer Berlin Heidelberg, 2005:225-232.

[27] Bosilca G, Herault T, Rezmerita A, et al. On scalability for MPI runtime systems[C]// Proc of 2011 IEEE International Conference on Cluster Computing, 2011:187-195.

[28] Yoo A B, Jette M A, Grondona M. SLURM:Simple linux utility for resource management[M]// Job Scheduling Strategies for Parallel Processing. Berlin, Heidelberg:Springer Berlin Heidelberg, 2003:44-60.

[29] Ahn D H, Garlick J, Grondona M, et al. Flux: A next-generation resource management framework for large HPC centers[C]// Proc of the 2014 43rd International Conference

on Parallel Processing Workshops, 2014:9-17.

[30] Cao C X, Herault T, Bosilca G, et al. Design for a soft error resilient dynamic task-based runtime[C]// Proc of 2015 IEEE International Parallel and Distributed Processing Symposium, 2015:765-774.

[31] Zhong D, Bouteiller A, Luo X, et al. Runtime level failure detection and propagation in HPC systems[C]// Proc of the 26th European MPI Users' Group Meeting on(EuroMPI'19), 2019:1-10.

## 作者简介:

**张昆**(1995-),**女,山东烟台人,硕士生,研究方向为系统软件和高性能计算。E-mail**:zhangkun_102311@163.com

**ZHANG Kun**, born in 1995, MS candidate, her research interests include system software, and high performance computing.

**张伟**(1982-),**男,吉林榆树人,博士,助理研究员,研究方向为高性能计算和并行存储。E-mail**:weizhang@nudt.edu.cn

**ZHANG Wei**, born in 1982, PhD, assistant research fellow, his research interests includes high performance computing, and parallel storage.

**卢凯**(1973-),**男,上海人,博士,研究员,研究方向为并行编程、操作系统和系统安全。E-mail**:kailu@nudt.edu.cn

**LU Kai**, born in 1973, PhD, research fellow, his research interests include parallel programming, operating system, and system security.

**董勇**(1980-),**男,山东宁阳人,博士,副研究员,CCF 会员**(E200034838M),**研究方向为高性能计算、并行 I/O 和网络存储系统。E-mail**:yongdong@nudt.edu.cn

**DONG Yong**, born in 1980, PhD, associate research fellow, CCF member(E200034838M), his research interests include high performance computing, parallel I/O, and network storage system.

**戴屹钦**(1998-),**男,广东中山人,硕士生,研究方向为系统软件和高性能计算。E-mail**:yiqindai98@163.com

**DAI Yi-qin**, born in 1998, MS candidate, his research interests include system software, and high performance computing.