计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.181202409

并行程序设计语言中局部性机制的研究

袁 良[1] 张云泉[1] 白雪瑞[2] 张广婷[1]
1 中国科学院计算机体系结构国家重点实验室 北京 100190
2 中国科学院前沿科学与教育局重点实验室处 北京 100864
(yuanliang@ict.ac.cn)

摘 要 大规模并行应用程序的性能优化和并行化的关键瓶颈之一在于多核 CPU 中越来越深和越来越复杂的存储层次。文中系统地分析和总结了当前主要多核 CPU 和并行程序设计语言中的局部性设计方法,提出了两种局部性,即横向局部性和纵向局部性,从这两种局部性的视角深入分析了当前的主要并行程序设计语言的局部性设计机制,进一步总结对比了其优缺点,并指出了新一代并行程序设计语言应具有的特点,重点提出了新语言应同时综合考虑两种局部性支持的设计机制的研究观点。

关键词:并行程序设计语言;并行程序设计模型;局部性;并行性;多核
中图法分类号 TP312

Research on Locality-aware Design Mechanism of State-of-the-art Parallel Programming Languages

YUAN Liang[1], ZHANG Yun-quan[1], BAI Xue-rui[2] and ZHANG Guang-ting[1]
1 State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
2 Bureau of Frontier Sciences and Education, Chinese Academy of Sciences, Beijing 100864, China

Abstract The memory access locality of a parallel program becomes a more and more important factor for exploiting more performance from the more and more complex memory hierarchy of current multi-core processors. In this paper, two different kinds of locality concept, horizontal locality and vertical locality, were proposed and defined. The state-of-the-art of parallel programming languages were investigated and analyzed, while the methods and mechanisms on how these parallel programming languages describe and control the memory access locality were analyzed in detail based on these two kinds view of horizontal locality and vertical locality. Finally, some future research directions on parallel programming languages were summarized, especially on the importance of integrating and support both horizontal locality and vertical locality in the future parallel programming language research.

Keywords Parallel programming language, Parallel programming model, Locality, Parallelism, Multi-core

1 引言

随着处理器速度与访问延迟之间的差距越来越大,在硬件上体系架构设计者引入了名为存储层次的设计概念。同样,在程序设计语言方面,为了获取更高的性能,并行程序设计语言需要提供良好的局部性描述和表示机制,才能将存储层次的硬件设计优点充分挖掘出来。由此可见,在当前的并行算法研究领域中,大量的文献都是对某种算法的局部性改进研究;与此同时,在并行编程模型和并行语言研究领域中,科研人员也将不同的局部性描述和表示机制逐步加入,用于增强语言对局部性挖掘的支持力度。

由于多核处理器的出现,使得当今并行计算机存储系统日益复杂和加深,编译器和运行时系统更难以充分开发局部性。科研人员若要写出性能高的代码,需要考虑在并行程序设计语言和编程模型的设计中加入局部性的描述和表示机制,以方便科研人员利用这些机制来描述和挖掘算法及程序中的局部性。本文深入分析了当今主流并行程序设计语言的局部性设计机制。

本文的工作主要分为 3 个部分:1)对多核并行处理器和并行程序设计语言的发展趋势做了阐述,依照并行计算系统

到稿日期:2018-12-15 返修日期:2019-06-04 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划(2017YFB0202001);中国科学院战略性先导科技专项(C 类)(XDC01040100);国家自然科学基金(61432018,61521092,61602443);北京自然科学基金(L182053);中科院高效空间天气预报模式科技创新交叉与合作团队
This work was supported by the National Key R&D Program of China (2017YFB0202001), Strategic Priority Research Program of Chinese Academy of Sciences (XDC01040100), National Natural Science Foundation of China (61432018, 61521092, 61602443), Science Foundation of Beijing (L182053) and CAS Interdisciplinary Innovation Team of Efficient Space Weather Forecast Models.
通信作者:张云泉(zuq@ict.ac.en)

存储层次的发展规律，以主流并行编程语言所提供的局部性描述和表示机制为基础，总结了将并行程序设计语言中支持的局部性分为横向局部性和纵向局部性的观点，并给出了横向局部性和纵向局部性的定义；2）以两类局部性定义为基础，分类论述和深入分析目前主流的并行程序设计语言的局部性描述和实现机制，并利用两类局部性对这些主流并行程序设计语言进行了综合分类和对比分析；3）指出了新一代并行编程语言所应具有的若干主要语言特点，重点提出了应综合考虑对横向局部性和纵向局部性的描述和表述机制进行研究的观点。

本文第 2 节概括了关于多核处理器和并行程序设计语言的发展趋势，通过总结，提出了横向和纵向局部性的定义；第 3 节介绍了目前主要的并行程序设计语言，从两类局部性的角度分析每一种语言的局部性机制；第 4 节对两种不同的局部性概念进行阐述，理清并行性与局部性之间的关联，通过这两种局部性分类的方式分析语言和模型的优缺点；最后，在此基础上总结并行编程语言，指明下一代并行编程模型和语言所应具有的特点以及未来的研究方向。

2 并行处理器体系结构和程序设计语言的发展趋势

2.1 并行处理器体系结构的发展趋势

观察总结并行计算系统体系结构和处理器近几年的发展可以发现，主要有 3 个明显的现象：1）多核架构的处理器已成为高性能计算机的基本计算单元；2）以异构多核或异构众核构成的并行计算系统越来越多；3）并行计算系统的节点内部和处理器内的存储层次将日趋复杂。

受限于功耗墙、存储墙、发热墙以及指令集并行（Instruction-Level Parallelism，ILP）墙等因素，以通过提高时钟频率或者改进 CPU 流水线的传统方式来提升 CPU 性能的做法没有延续性。自 2005 年 AMD 推出了双核处理器至今，大部分 Top500 中的系统使用了多核甚至异构多核处理器，使多核必将成为并行计算系统的基本计算单元。

文献[1]从理论上印证了若想获得更好的性能加速，则处理器必须支持异构多核。从应用层面，由于计算模式和存储访问模式不同，不同应用需要不同并行粒度和架构的核进行计算。通过对 2005 年出现的 GPGPU 计算系统展开大量的应用研究发现，其比较适用于细粒度计算密集和数据并行程序，并能获得很好的加速性能。国际上对面向计算密集和数据并行应用的新型体系结构也展开了研究，很多新型体系结构相继出现，如 Imagine，RAW，Viram，TRIPS，CELL，Score 和 Larrabee 等。最近，AMD 公司也在极力宣传融聚概念，即将 CPU 和 GPU 集成在一个芯片内部。由此可见，处理器内集成异构的多核或不同类型的处理器集成在同一节点内部进行加速计算的方式必将成为主流发展趋势。

随着集成电路（Very-Large-Scale Integration，VLSI）技术的发展，计算变得相对廉价，传送数据则相对耗时，由此得出一个结论，处理器设计的瓶颈在于通信。例如，在 Itanium2 处理器中，12 个整数计算单元（Arithmetic Logic Unit，ALU）、2 个浮点 ALU 及其寄存器只占表面积的 6.5%，其他面积则用于通信和控制。访存延迟与 CPU 计算速度之间存在巨大的速度差距，在硬件层次上，为了缓解带宽和延迟对性能的影响，引入了同时多线程（Simultaneous Multi-Threading，SMT）技术和设计更加复杂的存储层次。图 1 显示了基于多核的 SMP 系统的存储层次模型、片内 L1 和 L2 Cache，甚至片外 L3 Cache 的设计，有效地缓解了内存墙问题。科研人员只有有效地利用这些硬件特征，才能通过引入的复杂存储层次的方式缓解存储访问对程序执行性能的影响。此外，不同处理器的核内共享方式不同，有的直接共享 L1 Cache，并不存在独立的缓存，有的处理器的缓存地址是独立的，因此为了获得高性能，须针对不同的处理器体系结构存储层次模型设计不同的算法以及调度策略，这使得编程的难度大大增加，也让程序在不同体系结构下保持局部性的可移植变得非常困难。因此，若要求硬件存储层次的复杂化继续加深，软件开发和存储层次的局部性优化也将面临更大的挑战。

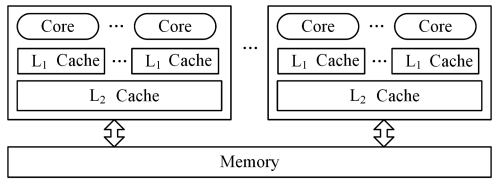


图 1 并行计算系统单节点内的存储层次

Fig. 1 Memory hierarchies in one single node of parallel systems

2.2 并行程序设计语言的发展趋势

随着多路多核处理器的发展，最受欢迎的基于分布式存储的 MPI（Message Passing Interface）[2] 编程方式在多路多核层次的并行算法和性能优化设计上遇到了瓶颈。虽然在节点间运行 MPI 而在节点内运行 OpenMP[3] 可实现两个层次的混合并行，但其要求科研人员在设计程序时要考虑两种不同的并行语言，使得用户设计并行程序的难度增加。因此，新一代并行编程语言必须将多层次并行的支持考虑在内。

对访存的延迟和带宽而言，在硬件层次的设计上，都需要对软件进行充分利用和优化，使数据局部性成为制约程序运行性能的一个关键因素，因此未来的编程语言必须提供给科研人员定义数据局部性的机制[4-5] 才可以减少访存和通信次数，从而提高性能。

观察处理器和编程语言的发展可以发现，由于多核的出现和更加复杂的存储层次，在并行编程语言中局部性机制的设计更加重要。目前，对于使算法针对多核具有良好的存储访问局部性的方法的研究已经成为热点。因此，我们认为：在并行编程语言中，如何设计易于表达算法局部性的机制，例如流编程模型中通信和计算分离以及 Sequoia[6-8] 中的显式存储控制等方式，将是语言设计者未来的研究重点。

2.3 横向局部性和纵向局部性

通过分析上述并行体系结构和并行程序设计语言的发展趋势，并结合第 3 节分析的已有并行程序设计语言局部性机制，本文提出将并行程序设计中的局部性分为横向局部性和纵向局部性。本节对两种局部性做了简要定义，第 3 节和第 4 节将基于两类局部性的分类对具体的并行程序设计语言做深入的对比分析。

定义 1 横向局部性是指数据在不同地址空间的同一层存储之间分布的局部性。

定义 2　纵向局部性是指数据在同一地址空间下的不同存储层次之间访问的局部性。

对于分布式存储的并行程序设计，对局部性的考虑更多的是在横向局部性上，即如何减小消息传递开销。这里所指的地址空间可以是物理的，也可以是逻辑的。纵向局部性以时间局部性和空间局部性为主。对串行程序而言，对局部性的考虑只能在纵向局部性上，对此目前已经有大量的文献研究，例如分块是数值算法中最常用的优化纵向局部性的方法。本文将分析在语言层次是否能够提供统一的框架来同时考虑这两种局部性。

3　并行程序设计语言及其局部性

本节首先简单介绍 Pthreads[9]，OpenMP[3]，MPI[9]，HPF[10]，PGAS[11-13]，HPCS[14-16]等并行程序设计语言。由于这些语言都没有显式的管理纵向局部性的机制，因此只能在横向局部性的设计方法上对这些语言进行简单评价。然后，针对以 GPU 为代表的各种加速器的计算速度的发展，例如 Nvidia 的 CUDA[17]通用计算平台、AMD[18]的流计算编程模型、OpenCL（Open Computing Language）[19]、CBE（Cell Broadband Engine）[20]架构和各种流处理器这类非 CPU 通用计算开发平台，本节还将对 GPU 和 CELL 上的编程语言和模型做阐述，包括显式控制存储层次间数据流动的语言 Sequoia，并对这些语言和模型的优缺点进行分析，对其横向和纵向局部性设计方法做阐述。已有的并行程序设计语言更多通过非阻塞通信和块通信等技术来隐藏延迟，没有考虑提供机制来帮助程序员提高并行程序的横向局部性。最新的并行语言提供了不同的机制来辅助科研人员设计数据的分布，本文将重点介绍 UPC[13]和 Chapel[15]。

3.1　基于线程模型的 Pthreads

Linux 操作系统中的多线程接口标准为 Pthreads（POSIX thread）[9]，是一个可移植的多线程库。其为科研人员提供了创建、管理线程和线程同步等接口，可以随意控制线程流；同时导致了正确性和性能的问题。线程由于共享地址空间的问题会产生数据竞争，这又需要使用显式的同步和互斥机制来控制线程，而线程间的等待极易引起性能的下降甚至死锁等问题，在多核平台上运行的 Pthreads 还可能会带来 Cache 伪共享问题。

使用 Pthread 在共享存储系统中进行多线程程序设计与使用 MPI 编程在分布式存储系统中进行多进程程序设计类似，需要科研人员对数据进行分配和同步，但是利用 Pthread 编程是基于共享存储模型的，其数据访问不用显式的通讯。科研人员可以自由设计数据并行、任务并行和流水线等多种编程模式。虽然 Pthreads 是基于共享存储模型的，但是由于多核系统在内存层次上是共享的，而在高速缓存层次上是单个核私有的，因此也会带来横向和纵向局部性的问题。通过对算法进行改进和调优，可以使一个程序充分利用多核的存储层次，但是 Pthreads 并没有在语言级上给科研人员提供这种便利。

3.2　基于共享存储模型的 OpenMP

OpenMP[3]标准形成于 1997 年，是一种在共享内存的多核计算平台上进行多线程编程的编程模型。整体而言，OpenMP 可分为 3 个重要组件：编译制导指令、运行时库和环境变量。需要注意的是，OpenMP 不是编程语言，而是对 C，C++和 FORTRAN 等语言的扩展。OpenMP 中所有的并行化都是通过编译制导语句来实现的，可以通过 omp for 制导完成数据并行，并通过 omp sections 制导完成任务并行。相比基于线程的编程模型（如 Pthread），OpenMP 程序非常容易编写，但是它对并行的方式有固定的限制，不如 Pthread 灵活。

与 Pthreads 编程模型开发人员可更多关注并行细节不同，OpenMP 具有 global-view 特性，采用编译制导指令的方式可使开发人员不用关注底层并行实现细节，数据划分、任务管理和调度都由编译器或者运行时系统负责，这虽然大大降低了应用开发人员的负担，但也造成了 OpenMP 可应用的并行模式非常有限，同时也缺乏对局部性进行管理的方法[21-25]。

3.3　基于消息传递模型的 MPI

消息传递接口[2]（Message-Passing Interface，MPI）诞生于 1994 年，是目前在并行分布式内存计算平台上最常用的编程方法，通常采用单程序多数据（Single Program Multiple Data，SPMD）模型进行并行程序设计。MPI 通过定义一系列的消息传递接口实现在不同计算节点间的数据通信和消息传递。值得注意的是，这些消息传递接口的实现是相对独立的，与具体计算平台特性（如厂商、操作系统或者硬件特征）无关，因此 MPI 计算程序可实现在不同计算平台上的平滑移植。在实际实现中，MPI 定义了 4 个基本概念：消息数据类型、通信子、通信操作和虚拟拓扑；同时，MPI 标准共定义了 200 多个接口，包括 MPI 子集、MPI 消息机制、点对点通信、集合通信等。

MPI 程序在进行节点间通信时，会带来大量的通信开销。因此，采用粗粒度并行、尽可能减小通信开销是 MPI 编程的重要方面。在 MPI 程序的具体实现过程中，程序员需要对数据和任务进行显式划分，然后通过消息传递接口将其发送到具体的计算节点上。同时，MPI-2 也提供了针对文件不同部分的读写函数，以实现对文件数据的划分。因此，MPI 可使程序员精确控制并行程序，最大程度地提升程序性能，但这也大大增加了开发人员的工作负担；同时，由于通信和共享造成的死锁问题，也是 MPI 编程的难点之一[24-50]。

3.4　基于数据并行模型的 HPF

HPF[10]是 1993 年发布的一种数据并行语言，并在 1997 年更新为 2.0 版本。HPF 对 FORTRAN90 编程语言进行了扩展，包括重新定义的新指令和新语法、重新定义和实现的新算法等。HPF 采用统一的全局地址空间，并没有定义显式的任务划分和数据通信接口，因此，开发人员可以编写高层次的并行代码，这不仅降低了开发人员的工作负担，而且使得并行程序具备良好的可移植性。数据分布和数据并行结构是 HPF 的两个最重要的特征。HPF 数据分布的目标是最小化计算节点间的通信开销，使程序具有良好的横向局部性，当然，这项工作主要通过指令前缀的方式告知编译器执行该工作。HPF 数据并行结构主要包括数组运算、FORALL 语句、

INDEPENDENT 指示、HPF 库算法和内部算法等。在任务划分上，HPF 编译器将根据主属规则，将不同的计算任务分配到对应的处理器上执行。

HPF 是一种基于数据并行模型的编程语言，为开发人员提供了数据划分、分配和对齐机制。同时，HPF 对任务并行的支持并不充分，只针对规则数据提供了几种数据划分和分布方法，并没有对非规则算法和数据结构的支持。

3.5 基于划分的全局地址空间模型的 PGAS

虽然由于访存延迟和访存竞争等问题，多处理器并行计算系统的存储系统逐步从共享存储转变到分布式存储，但是基于共享存储编程依然是非常重要的编程模式之一。共享存储编程模型对底层硬件平台进行了统一抽象，从而对应用开发人员隐藏了具体的存储系统细节，应用开发人员无须关心底层的存储模型，只需要像编写本地程序一样，直接调用非本地数据。这种编程方式比基于消息传递编程简单得多，从而大大降低了开发人员的工作负担。

PGAS[11-13]是结合消息传递和共享存储两个编程模型的优点的典型代表：一方面采用全局地址和统一地址空间大大提升了可编程性，另一方面采用类 MPI 的 SPMD 编程模式可有效提升程序性能。PGAS 设计和实现的核心是基于逻辑划分的全局地址空间内存模型，可根据处理器的数量和分布对统一地址空间进行逻辑划分，处理器间的数据划分和通信由开发人员负责。在分布式存储计算平台上，PGAS 往往采用单边通信算法库实现统一的虚拟地址空间。总体而言，相对于其他并行编程语言，PGAS 的最大优势在于对数据分布和通信进行了统一的定义和抽象。

3.5.1 CAF

CAF[11]是 Minnesota 大学的 Robert Numrich 和 John Reid 设计开发的首个 PGAS 语言，实现了对 FORTRAN90/95 编程语言的扩展，从而完成了对任务和数据的分配。CAF 采用 SPMD 编程模式，将计算任务抽象为固定数量，并且其有自己专属数据集的映像(image)，映像间通过 co-array 方式实现数据分配及通信。例如：real，dimension(n)[*]::x，y 语句对每个映像分配了大小为 N 的数组。值得注意的是，CAF 既可以采用线程也可以采用进程实现并行程序的开发，因此其应用领域比 OpenMP FORTRAN 更广。同时在内存访问上，CAF 通过赋值语句进行直接内存访问的方式替代了复杂的消息传递方式[51-52]。

3.5.2 Titanium

Titanium[12]通过引入 regions 概念实现了对 JAVA 编程语言的扩展，采用了类 UPC 内存模型，并采用了单边通信模式。因为 Titanium 是对 JAVA 的扩展，所以可面向对象编程。定义无序迭代命令 foreach 是 Titanium 语言最重要的特性，该命令允许不同迭代的并发执行。

3.5.3 UPC

UPC[13]通过定义同步原语和内存管理原语，实现了对 ANSI C 编程语言的扩展，并通过定义共享变量的方式实现了线程间的通信。UPC 基于 SPMD 并行模式，针对共享存储和分布存储两种不同的并行计算平台定义了统一编程模型。UPC 执行模型定义了一组可独立运行的线程，并通过定义 Barriers 和 Locks 命令实现了线程间的同步。UPC 内存模型定义了包含共享内存和私有内存的统一全局地址空间，其中共享存储空间可以被所有线程访问，而私有存储空间只能被其对应的线程访问。

UPC 能够定义共享存储空间和线程之间的亲缘性，这使得应用程序可充分开发数据局部性以提升程序性能。UPC 通过定义 shared 关键字来标记数据的属性是私有还是共享。当 0 号线程可访问 shared 变量时，可以说该 shared 变量与 0 号线程具有亲缘性。数组可以通过定义循环分布和块分布两种不同方式来实现数据在线程间的分配。例如：upc_forall 语句可将没有依赖关系的不同迭代进行分布，即每个线程都会分配一组循环迭代及其对应的数据，并可独立执行。

3.5.4 PGAS 语言的局部性

与 HPF 语言类似，PGAS 也定义了一系列数据和任务分布函数，能够根据线程和数据的对应关系进行计算，但在数据局部性方面并没有做太多优化。PGAS 语言的优势是为开发人员提供统一全局地址空间，开发人员可定义共享的全局数据结构。然而，PGAS 仍然保留了 Local View 设计，应用程序需要传输到对应的物理处理器上才能执行，并需要对数据和任务边界进行统一处理。因此，PGAS 仍然需要开发人员对底层的通信细节进行显式管理。为此，ZPL 等提出了 Global View 语言，对上述问题进行了彻底优化。

3.6 以高生产率为目标的 HPCS

DARPA 于 2001 年提出了 HPCS 计划[9-11]，其目标是将并行程序的生产效率提升 10 倍，其发展可分为 3 个阶段。虽然 Sun 公司依然致力于开发和优化 Fortress 语言，但是 HPCS 已不再对其提供支持，因此本文不再对其进行分析和讨论。HPCS 并行程序的生产效率可理解为可编程性，也就是提高应用开发人员的编程效率，使并行程序的编写、移植、扩展、维护和优化更加简单，同时使并行程序性能具有在不同计算平台间的可移植性和健壮性。由于 Chapel 的部分设计理念受 ZPL 语言的影响很大，因此本节首先对其进行详细介绍。

3.6.1 ZPL

ZPL[14]语言是由 Washington 大学的课题组进行设计和开发的。该语言具备两个典型特征：1)强调数据局部性；2)不明确定义通信网络的拓扑结构。提高语言的抽象层次是使编程语言与底层硬件平台无关的重要方法，但这会使应用程序的性能依赖于编译器，往往不能产生令人满意的性能。因此，编程语言的设计和开发需要在抽象层次和程序性能间找到平衡点。从数据分布的角度，ZPL 的设计者将并行编程语言分为 Local View 和 Global View 两种模式，ZPL 属于后者。

3.6.2 Chapel

Chapel[15]语言由 Cray 公司设计开发，其设计者之一 Bradford 来自 ZPL 语言设计小组。Chapel 语言首先明确定义了开发人员和编程语言的分工，其核心思想是认为编译器和运行时系统并不能很好地解决应用程序并行性开发和数据局部性这两方面的问题，因此需要开发人员来挖掘程序并行性，并对数据分布进行显式控制和分配，只有这样才能有效提升并行程序的性能；并行编程语言的主要任务是给开发人员

提供一种能够非常方便的表达其设计思想的机制,并能够定义一种机制实现对不同层次并行性的表达:既支持粗粒度的任务并行(例如 MPI),又支持细粒度的数据并行(例如 OpenMP)。

通过对 ZPL 编程语言的分析和讨论,Chapel 语言设计和定义了基于 Global View 的编程模式,使应用程序开发人员不必关注底层通信细节,从而能够更加关注算法本身,达到了算法和实现分离的目标。为了支持基于 Global View 编程模式的数据分布,Chapel 语言不仅定义了数据分布的默认方式,而且允许开发人员对数据分布方式进行自定义,并允许重用。

3.6.3　X10

X10[16]是 IBM 公司通过对 JAVA 编程语言的扩展而开发的一种新的并行编程语言。X10 消除了原 JAVA 语言定义的并发和数据,并定义了统一的全局地址空间,设计并重新定义了并发、分布和局部性等概念。新定义的并发概念包括 async,finish,atomic,future,force,foreach,ateach,clocks 等;新定义的分布概念包括点和分布等。X10 定义的数组主要分为两种:分布式数组和数组规约。得益于 JAVA 语言的良好特性,X10 同样支持面向对象,并自带垃圾回收功能。

3.6.4　HPCS 语言的局部性

PL 并行编程语言的最大优势是提高了程序设计和编写的层次,可大大降低开发人员的工作负担。但是,ZPL 程序的并行性挖掘主要来自数据操作,是隐式并行,从而增加了并行性开发的难度。编译器将 ZPL 程序中的 region 转换成循环迭代,并将没有依赖关系的迭代分布到不同处理器上并行执行。当 region 存在 at 操作符时,编译器会自动生成对应的通信语句,并对通信进行优化,例如采用通信避免算法,用计算隐藏通信。因此,ZPL 没有提供可以让开发人员显式表达局部性的机制,同时采用的 CTA 模型也没有很好地对存储层次进行抽象。

HPCS 提出的时间比较早,其主要目标是提高开发人员的工作效率,再加上当时多核处理器还没有出现,存储层次也非常简单。因此,Chapel 在语言设计上更多地采用了类似 ZPL 的 Global View 模式,这就使得开发人员在数据局部性的设计上,只能通过独立分布类的方式来实现。Chapel 对 ZPL 编程语言的最大改进之处是定义了数据的分布和重用。但是,这种方法也存在局限性,即只能面向横向局部性。同时,Chapel 设计了过于复杂的分布类,因此难以被充分利用和重用。

3.7　GPU 和流处理器

本节主要对流编程模型进行分析和讨论[53-55]。

3.7.1　StreamC/KernelC

StreamC/KernelC[53]是对 C 编程语言的扩展,是面向 Imagine 设计的一种新型编程语言,主要用于对流级/核心级程序的开发。StreamC 编程语言基于 C++开发,重新定义了完成流传输和拷贝的类库和函数。KernelC 是一种类 C 的编程语言,一方面 KernelC 只对 C 语言的部分特性提供了支持,但不支持全局变量、指针、函数调用等语句;另一方面 KernelC 也在 C 语言的基础上重新定义了新的数据类型和编程语句。流编程的整体流程是"gather 流—operate 流—scatter 流",即首先将数组中对应的元素组织成流,然后调用计算 Kernel 对流进行处理和计算,最后将计算结果 scatter 回数组。Gather 和 scatter 操作可以是顺序的、strided 和随机的。

3.7.2　Brook 和 AMD Brook+

Brook 设计的初衷是面向 GPU,主要用于图像处理和计算。同 StreamC/KernelC 一样,Brook 是基于 C 语言,并对其进行扩展的一种流编程语言。Brook 允许开发人员定义两种不同作用的同构数据结构:输入流(input stream)和聚集流(gather stream)。两者在使用方式上具有明显不同:输入流按照一定的规则顺序读入数据,同时数据不可重用;聚集流可以对数据进行随机读写,并可被多次重用,同时聚集流还定义了 Reduction 操作,开发人员可以根据需要定义对应的规约函数。

AMD Brook+[18]是 AMD 公司实现的 BrookGPU 规范,Brook+语言非常类似于 Brook 和 StreamC/KernelC,本文不做过多介绍。

3.7.3　CUDA

CUDA[17]是 NVIDIA 为实现 GPU 通用计算而设计和开发的编程框架,在使用 CUDA 编程并行应用程序时,GPU 抽象为可大规模细粒度并行的计算设备(Compute Device)。CUDA 编程模型的基本要素为线程,线程采用层次化的组织模式,多个线程组织为 block,多个 block 又组织为 grid。同时,GPU 内存模型定义了 GPU 不同层次的存储结构。在计算单元的组织上,GPU 由多个流多处理器构成,每个流多处理器又由多个 CUDA core 组成。需要注意的是,流多处理器采用了单指令多数据(SIMD)架构,即每个 CUDA core 在不同的数据上执行相同的指令。

3.7.4　OpenCL

开发计算语言(Open Computing Language,OpenCL)[19]是由包括 NVIDIA 和 AMD 在内的多家公司提出的面向异构计算系统的并行编程模型,其目标是实现"一份代码,到处执行",即相同的程序可高效运行在不同架构的计算平台上。OpenCL 1.0 版本于 2009 年推出,目前已更新至 OpenCL 3.0。OpenCL 可对计算系统中不同架构的处理器进行有效组织,并支持数据并行和任务并行两种不同的编程模式。OpenCL 也定义了 Kernel 以实现对核心计算模块的并行处理,大多采用数据并行的并行模式,同时也引入了任务并行。

3.7.5　流编程模型的局部性

流编程模型设计的最大优势是将通信与计算分离,不同任务间可进行并行调度。这样做的好处是可通过多个任务的并发执行最大化隐藏访存延迟,并最大程度地消除全局通信。在实现方式上,流编程模型可以在 Kernel 内部使用数据并行模式,同时也可在 Kernel 间使用任务并行模式。

对于访存密集型和计算密集型的应用程序来说,流编程模型是最佳选择。流编程模型作为算法和硬件的桥梁,非常适用于 GPU 这种大规模细粒度并行的处理器。然而,这些特征也导致了流编程模型的应用范围比较局限,只能对具有固定形式的应用进行并行开发。

流处理器使流编程模型可以实现对数据操作的显式控

制，可以对通信进行显式管理。然而，PGAS 和 HPCS 等较新的并行编程语言很好地继承了这些优点，只在横向对数据分布和节点间通信进行了重新设计，但是并没有实现对存储层次之间数据操作的显式管理。

3.8 CELL

Cell[20]处理器是由 IBM，Sony 和 Toshiba 联合开发的，包括 9 个独立核心：1 个 PowerPC Processing Element（PPE）以及 8 个 Synergistic Processing Element（SPE）。PPE 具有 64 kB L1 Cache 和 512 kB L2 Cache，顺序执行，具有 2 路 SMT，负责运行操作系统并管理 SPE 任务分配。SPE 是一个 128 位的 SIMD-RISC 处理器，拥有 256 kB 局部存储，每时钟周期可发送 2 条不同指令到不同流水线：一条为访存流水线，一条为计算流水线。SPE 的主要作用是执行计算密集型任务。

Cell 处理器的最大创新之处是将局部内存地址与全局内存地址独立开来，图 2 显示了 Cell 处理器的内存模型。这种内存模型要求并行程序开发人员对数据操作进行显式控制，通过消息传递或者 DMA 方式实现核内与内存间数据传输的控制。这样，既可控制数据的纵向局部性，不同 SPE 的计算没有依赖，相互独立，可并行执行；又可控制数据的横向局部性。基于 Cell 处理器，目前已经有多种不同编程模型，下面将详细分析和介绍其中 3 种典型的编程模型。

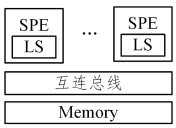


图 2 Cell 的内存模型

Fig. 2 Memory model of Cell processor

3.8.1 IBM Cell SDK

IBM Cell SDK[54]更多地将硬件架构的实现细节暴露给开发人员，如 SPE 的 SIMD 计算单元、本地内存与全局内存的 DMA 操作方式等。IBM Cell SDK 定义了多种不同的编程模型，开发人员可根据实际情况进行选择。值得注意的是，这些模型只提供了并行编程框架，并没有在语言层次给予支持。在众多的编程模型中，Function-Offload 和 Streaming 是最常用的两种编程模型。Function-Offload 也称为 RPC 模型，该模型允许运行在 PPE 上的线程调用运行在 SPE 上的线程，PPE 为 SPE 计算准备相应的数据，并可对运行在 SPE 上的线程进行组织和管理。通过该模型编程的并行程序，既可采用数据并行（所有 SPE 运行相同程序处理不同数据）模式，也可采用任务并行模式（不同 SPE 运行不同程序）。Streaming 编程模型将每个 SPE 看作流水线中的不同处理单元，不同 SPE 之间可进行通信。

3.8.2 Cellgen

Cellgen[55]是一种针对 OpenMP 的源到源转换器，由 Virginia Tech 设计开发。Cellgen 基于 OpenMP，并实现了它的一个子集。Cellgen 的输入为 OpenMP 并行程序，输出为可在 SPE 和 PPE 上并行运行的 C 代码。同 OpenMP 类似，Cellgen 抽象和定义了共享存储模型。同时，与 OpenMP 对数据属性的定义一致，Cellgen 的数据类型也可定义为共享数据或者私有数据。通过对程序上下文的分析，共享变量根据使用情况被编译器标示为 in 流、out 流或者 inout 流 3 种不同类型，Cellgen 通过此种方式管理程序的数据局部性。由于 Cell 的片上本地内存非常小，不可能存储所有的操作数据，片上和片外的通信是并行程序主要的性能瓶颈，因此可通过定义不同流的方式来实现通信和计算的重叠，用计算来隐藏访存延迟。在 Cellgen 设计中，in 流和 out 流都采用了 double-buffering 机制，一个 buffer 专门用来进行数据传输，另一个 buffer 专注于计算；inout 流则采用了 triple-buffering 机制，3 个 buffer 分别用来进行数据的输入、输出和计算。除了通过计算来隐藏访存延迟外，Cellgen 也可通过循环展开应用程序来实现该功能。

3.8.3 Sequoia

Sequoia[6-8]是一种基于 PMH 模型的并行编程语言，由美国 Stanford 大学的 William Dally 研究组设计开发。Sequoia 允许开发人员对内存的组织和分配进行显式管理，并可显式管理数据的分配和通信。Sequioa 编程语言遵循了当前计算机体系结构的发展趋势：内存层次性不断增加，可控制不同内存层次的编程模型存储愈发重要；不同层次内存间的通信方式和协议日益多样化，定义统一的通信接口对开发人员屏蔽通信细节，降低复杂性，使应用程序具有更好的可移植性变得尤为重要；计算机体系结构需要应用程序能够显式操作内存系统，以对数据进行分配和通信。

图 3 给出了 Sequoia 的内存模型。对于集群并行系统，内存层次从外到内依次为 Disk，Memory，L3 Cache，L2 Cache，L1 Cache。图 4 给出了基于 Sequoia 的 Cell 处理器的内存模型，最后一层为纯粹的计算层次，其余层次的内存仅负责数据通信。

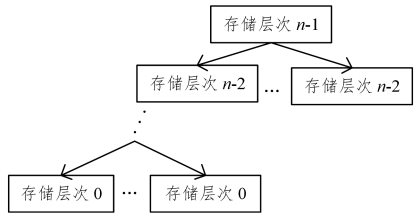


图 3 Sequoia 的存储模型

Fig. 3 Memory model of Sequoia

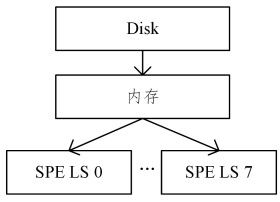


图 4 从 Sequoia 角度观察 Cell 的存储模型

Fig. 4 Cell's memory model from perspective of Sequoia

task 是 Sequoia 的基本元素，task 能够在不同的内存层次上通过传递参数的方式调用 subtask，这也是 Sequoia 表示数据通信和数据流动的唯一方式。也就是说，task 间只能通过参数传递和结果返回实现通信和调度，否则不同 task 是独立的，并拥有独立的内存空间，同时不同 task 也是可以并行执行的。task 可以根据底层硬件平台特性进行参数设定，这就使得应用程序具备了不同硬件平台间的可移植性。Se-

quoia 共定义了两种 task：1）内部节点 task，只负责在不同的内存层次间实现数据分配和映射任务，除非调用预定义的映射原语，内部节点 task 不会进行任何计算；2）叶子 task，不对其他 task 进行调用，主要负责计算任务。同时，Sequoia 还预定义了一组管理数组分块的原语和一组管理数据映射的原语。

3.8.4　Cell 编程模型的局部性

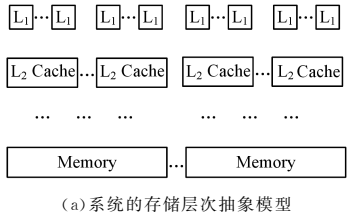
Cell 在数据局部性的管理上与 GPU 类似，但 Cell 可对数据移动和通信进行更加显式的管理，因此更加灵活。在编程方面，Cell 不仅能够使用流式编程模型，而且还能够使用其他类型的编程模型，同时也能更好地利用双缓冲技术对程序性能进行优化。当应用开发人员通过使用 Cell SDK 编写应用程序时，需要对 Cell 的硬件架构特征以及多线程编程具有一定的了解。除此之外，开发人员还需要对数据通信、数据对齐等操作进行显式管理，当需要用计算来隐藏访存延迟时，还需要显示分配双 buffer。

在数据局部性管理方面，IBM Cell SDK 给应用开发人员提供了最大的自由度。应用开发人员可最大限度地通过手工优化来提升应用程序的性能。然而，另一方面，IBM Cell 暴露给开发人员细节的方式比较简单，同时也没有提供相应的机制来管理局部性，这又增加了开发人员的工作负担。Cellgen 将 OpenMP 的编程方式引入 Cell 上，其优缺点与 OpenMP 相同。

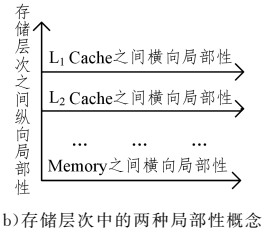
与 IBM Cell SDK 不同，除了与局部性相关的硬件特征外，Sequoia 并没有将 Cell 的过多硬件细节暴露给用户，同时抽象定义了一个可跨平台移植的编程模型。在该模型的定义中，开发人员通过使用预定义的两种 task 在不同的存储层次间进行通信和计算，这就使得纵向局部性可控。Sequoia 的设计思想相对成熟和成功，下一节将对其进行详细介绍。

4　并行程序语言设计中的横向局部性和纵向局部性

并行编程语言的首要设计目标是让开发人员编写出高性能的应用程序。由于处理器的计算性能与存储的访问性能间的差距越来越大，良好的局部性是高性能应用程序的重要指标。随着并行计算系统存储系统的复杂性日趋提高，高性能并行应用程序需要能够开发和描述两种不同的局部性：横向局部性和纵向局部性。图 5 说明了这两种局部性的差别。



(a)系统的存储层次抽象模型



(b)存储层次中的两种局部性概念

图 5　横向局部性和纵向局部性

Fig. 5　Horizontal and vertical localities

4.1　横向局部性

横向局部性的定义为位于同一内存层次上的数据由于存储位置的分布而导致的通信，如在 Jacobi 算法中对数组数据进行分块而导致的不同块间的边界通信。当数据分布具有良好的空间局部性时也会有良好的的横向局部性。但是横向局部性不仅仅是空间局部性的代名词，还包含数据通信开销。

并行编程语言的横向局部性设计可分为两种：1）数据划分和分配由编译器自动进行，如 OpenMP；2）数据划分和分配由开发人员进行管理和控制，如 HPF。本节将详细分析不同编程语言的横向局部性设计机制。

OpenMP 和 MPI 都没有为应用开发人员提供机制来支持并行程序的横向局部性设计，并且两种编程语言对横向局部性的管理处于两个极端。OpenMP 是基于共享内存的编程模型，在存储层次上只考虑了内存，并没有涉及 Cache 对性能的影响。同时，OpenMP 没有定义显式的局部性管理机制，数据的划分和任务的分配都由编译器和运行时系统自动完成。与之相反，MPI 对横向局部性的管理都由开发人员完成。与 OpenMP 和 MPI 都不相同，HPF 编程模型设计和定义了不同的分布类型来管理并行程序的横向局部性，因此 HPF 可以更好地管理横向局部性。并行程序开发人员在编写 HPF 程序时，通常会使用编译制导语句来实现对数据的分块，并使用 forall 语句来完成对不同数据块的并行操作，但是 HPF 对数据局部性的描述比较单一。PGAS 并行编程语言采用划分的内存地址空间，对 OpenMP 的存储模型进行了改进，可显式管理数据分块，数据分块的方法虽然有差异，但思想相同，即数据分块由并行程序开发人员显式管理，线程间的通信由编译器和运行时系统负责。HPCS 语言（如 Chapel）通过定义 distributions 对象来实现对数据分块的管理，并将其应用到 domains 索引集上。这种方法在考虑横向局部性的同时也兼顾了负载均衡，这就使得并行程序开发人员能够设计不同策略的分布方法，更重要的是这些方法可以被重用。图 6 明确地描述了这些语言在地址空间和横向局部性坐标中的位置。

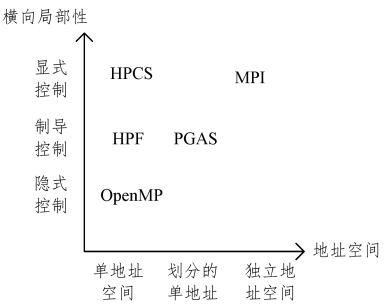


图 6　编程语言在地址空间与横向局部性坐标中的位置

Fig. 6　Programing languages in two localities coordinated system

4.2　纵向局部性

纵向局部性的定义为不同存储层次（如内存和 Cache）之间的局部性。纵向局部性可分为两种：时间局部性和空间局部性。现代并行计算系统中，存储层次的复杂性以及 CPU 性能和访存性能的差距日益扩大，高性能计算程序必须要充分考虑数据的纵向局部性，同时，这种纵向局部性的考虑往往不具备可移植行。因此，重新设计和开发的并行编程模型必须要实现对存储层次的显式管理，特别是在多核处理器出现后，

如何使并行程序能够显式管理存储层次，从而对纵向局部性进行优化，已经成为提升并行应用程序性能的主要方法。

在纵向局部性的显式管理上，目前已经有基于 Cell 处理器的多个不同编程模型。Cell SDK 允许开发人员显示管理片上 Local Store 和内存之间的通信，这实现了对横向局部性和纵向局部性的显式管理。同 MPI 通过消息传递的方式来实现对横向局部性管理一样，Cell SDK 可以认为是在 Cell 核内进行 MPI 编程；同 Pthreads 通过创建 SPE 线程以及 DMA 数据传输实现了显式管理纵向局部性一样，Cell SDK 也可以认为是在独立地址空间进行 Pthread 编程。然而，Cell SDK 的这两种局部性管理都存在明显的缺点：基于线程的横向局部性管理机制，大大增加了并行编程的难度和并行程序的开发效率；基于本地存储和内存的纵向局部性管理机制，也仅仅是对两个不同的存储层次间的通信进行了管理，不仅可扩展性差而且可移植性也差。

针对 Cell SDK 在局部性管理方面的缺点，Sequoia 应运而生。Sequoia 允许并行程序开发人员显式管理存储层次，并基于此设计高性能算法，目前 Sequoia 已经是 Cell 的重要组成部分。Sequoia 能够依据底层硬件平台的存储层次特征设置各存储层次间的数据传递大小，从而实现对纵向局部性的显式管理；同时，Sequoia 定义的存储层次树模型的根节点是并行计算系统的一级共享存储，如 SMP 并行计算系统中的共享存储器。然而，对于计算机集群系统来说，计算节点通过高速网络连接，并没有如上所述的共享机制。为此，Sequoia 增加了一个虚拟层，实现了对物理内存的抽象。如 Sequoia 将计算机集群中所有节点的内存抽象为根节点，子节点为每个计算节点的实际物理地址，虚拟根节点和子节点间会进行数据通信。这样 Sequoia 语言也可管理横向局部性，允许并行程序开发人员优化应用该程序的横向局部性和纵向局部性，从而提升应用程序的性能。因此，可以将 Cell SDK 看作具有两个存储层次的 Sequoia，并可在每个层次上进行显式管理(PPE 和 SPE)。图 7 详细描述了 Cell 编程模型在纵向局部性与横向局部性坐标中的位置。

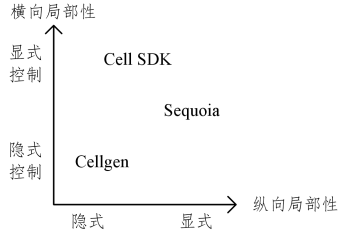


图 7 Cell 编程模型在纵向局部性和横向局部性坐标中的位置

Fig. 7 Cell's programming languages in two localities coordinated system

4.3 两种局部性的综合

若要实现并行应用程序的高性能，必须在算法设计和并行实现中充分考虑横向局部性和纵向局部性。已经设计了横向局部性管理机制的并行编程语言(如 MPI 和 PGAS)，无论采用什么方式的通信机制，其目标都是使并行程序开发人员能够显式管理横向局部性，从而最大程度地降低通信开销。这就意味着当使用这些语言开发并行应用程序时，同时考虑了并行性挖掘、并行模式和横向局部性，从而实现了并行性和横向局部性的完美融合。而在纵向局部性的考虑上，Sequoia 语言设计和定义的 task，既明确了多层次并行模式，又定义了对横向局部性和纵向局部性的管理机制，从而使编程语言能够管理并行和两种局部性。因此，新设计和开发的编程模型要遵循这种方式，使得并行程序开发人员能够设计和编写具有高效并行性和良好局部性的应用程序。

本节对现有的不同并行编程语言的局部性机制进行了对比和总结，表 1 详细列出了这些对比。其中，前 3 列采用的是不同编程模型的并行程序开发语言，这些语言最重要的区分是对横向局部性和数据分布的管理；PGAS 和 ZPL 分别实现了双边通信模式和数据分布对开发人员的隐藏，从而有效降低了并行应用程序的开发难度，但这两种语言并没有在局部性管理方面进行太多的考虑和改进；Sequoia 通过 task 的定义，既实现了对纵向局部性的显式管理，又提高了并行应用程序在不同硬件平台间的可移植性。

表 1 各种并行程序设计语言特点的对比

Table 1 Comparison of characteristics of parallel programing languages

编程模型	MPI[2]	OpenMP[3]	HPF[10]	PGAS[11-13]	ZPL[14]	Sequoia[6]
传统分类	消息传递	共享存储	数据并行	DSM	数据并行	混合型
View	Local	Global	Global	Local	Global	Local
数据分布	显式	隐式	显式	显式	隐式	隐式
生成任务	隐式	隐式	隐式	隐式	隐式	显式
粒度大小	粗	粗/细	细	粗/细	细	粗/细
横向局部性	否	是	是	是	否	是
纵向局部性	否	否	否	否	否	是
同步通信	显式是	显式否	隐式否	显式否	隐式否	隐式否
存储层次管理	否	否	否	否	否	隐式

结束语 本文首先对多核处理器和并行编程语言的现状和发展趋势进行了分析和讨论，然后详细描述了目前主流的并行编程语言的特点，特别详细讨论了局部性机制，最后总结了横向局部性和纵向局部性的定义，从而明确了基于存储层次的并行编程语言和并行编程模型的设计开发方向。最终提出面对目前主流的异构众核计算平台，新设计的并行编程语言及并行编程模型应具备如下要点：

1)具有显式管理横向局部性和纵向局部性的机制。面对目前并行计算系统日益复杂的存储层次，单纯依靠编译器和运行时系统，无法实现对并行程序的并行性开发和局部性问题的最优处理，往往需要开发人员进行显式管理和控制。

2)能够支持不同的并行编程模式，包括数据并行、任务并行和流水线并行等。

3)能够对程序的并行性和局部性进行统一管理和描述，并行程序开发人员能够通过编程语言提供的接口和定义，对并行算法的这两种特性进行整体设计和开发，在降低开发人员工作负担的同时，又可以提升并行程序的性能以及在不同硬件平台间的可移植性。

4)基于 Local View，Global View 虽然能够降低开发人员的工作负担，提升并行程序的开发效率，但是统一数据处理方案的设计思想也制约了对并行算法和数据划分进行更有效的设计，同时也仅能支持有限的并行模式和通信方式。虽然开

发人员能够使用复用和继承分布类管理局部性，但是本文认为这种复用性非常有限，即便设计和定义了大量可管理局部性的分布类，开发人员也难以对其进行高效利用。

根据以上描述和讨论，并行编程语言和编程模型研究可在以下两个方向上同时进行。

1）改进现有并行编程模型和语言（如 MPI[2]），增加能够适应目前多核异构并行计算系统的并行程序设计和开发机制，并行程序开发人员可以将 MPI 管理横向局部性的方法迁移到多核计算平台上，同时引入管理纵向局部性的描述机制，并设计能够支持这些特性的编译器或运行时系统。这种方式一方面可以降低学习成本，另一方面能够使并行程序语言的设计和开发者充分利用已有语言的设计机制。

2）基于本文讨论的 Sequoia[6] 的诸多优势，以及本文确定的新型并行编程模型和语言应该具有的特点，本文认为 Sequoia 能够成为成功的并行编程语言。然而，Sequoia 还缺乏能够支持现有并行开发模式的策略和机制（如同步机制），以及能够支持异构处理器的任务分配机制等。未来我们将深入研究 Sequoia，在总结其优缺点的基础上提出新一代并行编程模型。

参 考 文 献

[1] HILL M D, MARTY M R. Amdahl's Law in the Multicore Era [J]. IEEE Computer, 2008, 41(7): 33-38.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (Version 2.1) [S]. High-Performance Computing Center Stuttgart, 2008.

[3] OpenMP Standards Board. OpenMP Application Program Interface [OL]. http://openmp.org/wp/openmp-specifications/.

[4] ZHANG Y Q, CHEN G L, SUN G Z, et al. Models of parallel computation: a survey and classification [J]. Frontiers of Computer Science in China, 2007, 1(2): 156-165.

[5] ZHANG Y Q. DRAM(h): A Parallel Computation Model for High Performance Numerical Computing [J]. Chinese Journal of Computers, 2003, 26(12): 1660-1670.

[6] FATAHALIAN K, KNIGHT T J, HOUSTON M, et al. Sequoia: programming the memory hierarchy [C] // Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC). IEEE, 2006: 11-17.

[7] KNIGHT T, PARK J, REN M, et al. Compilation for explicitly managed memory hierarchies [C] // Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP). ACM, 2007: 14-17.

[8] HOUSTON M, PARK J, REN M, et al. A portable runtime interface for multi-level memory hierarchies [C] // Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, 2008: 20-23.

[9] IEEE POSIX P1003.4a: Threads Extension for Portable Operating Systems [M]. Piscataway, NJ: IEEE Press, 1994.

[10] High Performance Fortran Forum. High Performance Fortran Language Specification [OL]. http://hpff.rice.edu/versions/hpf2.

[11] NUMRICH R, REID J. Co-array Fortran for parallel programming [J]. ACM SIGPLAN Fortran Forum, 1998, 17(2): 1-31.

[12] YELICK K, SEMENZATO L, PIKE G, et al. Titanium: A high-performance Java dialect [C] // ACM 1998 Workshop on Java for High-Performance Network Computing. ACM, 1998: 1-10.

[13] CARLSON W, DRAPER J M, CULLER D E, et al. Introduction to UPC and language specification [R]. University of California-Berkeley, 1999.

[14] ALPERN B, CARTER L, FERRANTE J. ZPL A Machine Independent Programming [J]. IEEE Transactions on Software Engineering, 2000, 26(3): 197-211.

[15] CALLAHAN D, CHAMBERLAIN B L, ZIMA H P. The Cascade high productivity language [C] // Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments. IEEE, 2004: 52-60.

[16] CHARLES P, GROTHOFF C. X10: an object-oriented approach to non-uniform cluster computing [C] // Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. ACM, 2005: 16-20.

[17] CUDA 2.2 Programming Guide [OL]. http://www.nvidia.com/object/cuda_develop.html.

[18] Stream_Computing_User_Guide [OL]. http://developer.amd.com/.

[19] Khronos OpenCL Working Group. The OpenCL Specification Version: 1.0 [OL]. http://www.khronos.org/opencl/.

[20] CHEN T, RAGHAVAN R, DALE J, et al. Cell broadband engine architecture and its first implementation: a performance view [J]. IBM Journal of Research and Development, 2007, 51(5): 559-572.

[21] OREN G, GANAN Y, MALAMUD G. Automp: An automatic openmp parallization generator for variable-oriented high-performance scientific codes [J]. IJCOPI, 2018, 9(1): 46-53.

[22] BERTOLACCI I, STROUT M, SUPINSKI B, et al. Extending openmp to facilitate loop optimization [C] // 14th International Workshop on OpenMP. IWOMP, 2018: 53-65.

[23] KANG S, LEE A, LEE K. Performance comparison of openmp, mpi, and mapreduce in practical problems [J]. Advances in Multimedia, 2015, 24(5): 1-9.

[24] WU X, TAYLOR V. Performance characteristics of hybrid mpi/openmp scientific applications on a largescale multithreaded blue-gene/q supercomputer [J]. IJNDC, 2013, 1(4): 213-225.

[25] BENEDICT S. SCALE-EA: A scalability aware performance tuning framework for openmp applications [J]. Scalable Computing: Practice and Experience, 2018, 19(1): 15-30.

[26] LI H, CHEN Z, GUPTA R. Parastack: effecient hang detection for MPI programs at large scale [C] // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2017: 1-12.

[27] MORRIS B, SKJELLUM A. Mpignite: An mpi-like language and prototype implementation for apache spark [J]. arXiv: 1707.04788.

[28] AUBREY-JONES T, FISCHER B. Synthesizing MPI implementations from functional data-parallel programs [J]. International Journal of Parallel Programming. Springer, 2016, 44(3): 552-573.

[29] KOWALEWSKI T, FÜRLINGER K. Nasty-mpi: Debugging synchronization errors in MPI-3 one-sided applications[M]// European Conference on Parallel Processing. Cham: Springer, 2016: 51-62.

[30] DENIS A, TRAHAY F. MPI overlap: Benchmark and analysis [C]// 45th International Conference on Parallel Processing. ICPP, 2016: 258-267.

[31] KONIGES A, COOK B, DESLIPPE J, et al. MPI usage at NERSC: present and future[C]// Proceedings of the 23rd European MPI Users' Group Meeting. EuroMPI, 2016: 217-227.

[32] IMANI M, KIM Y, ROSING T. MPIM: multi-purpose in-memory processing using con gurable resistive memory[C]// 22nd Asia and South Pacific Design Automation Conference. ASP-DAC, 2017: 757-763.

[33] MÉNDEZ S, REXACHS D, LUQUE E. Analyzing the parallel I/O severity of MPI applications[C]// Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE, 2017: 953-962.

[34] DAO T, CHIBA S. Semem: Deployment of mpi-based in-memory storage for hadoop on supercomputers[M]// European Conference on Parallel Processing. Cham: Springer, 2017: 442-454.

[35] BAYSER M, CERQUEIRA R. Integrating MPI with docker for HPC[C]// 2017 IEEE International Conference on Cloud Engineering. IC2E, 2017: 259-265.

[36] AHMED H, SKJELLUM A, BANGALORE P, et al. Transforming blocking MPI collectives to non-blocking and persistent operations[C]// Proceedings of the 24th European MPI Users' Group Meeting. EuroMPI, 2017: 1-11.

[37] MAGOULÈS F, BENISSAN G. JACK2: an mpi-based communication library with non-blocking synchronization for asynchronous iterations[J]. Advances in Engineering Software, 2018, 23(5): 116-133.

[38] NIELSEN F. Introduction to HPC with MPI for Data Science [M]// Undergraduate Topics in Computer Science. Springer, 2016.

[39] BADER D. Evolving MPI+X toward exascale[J]. IEEE Computer, 2016, 49(8): 10-18.

[40] MOHAMED H, MARCHAND-MAILLET S. MRO-MPI: mapreduce overlapping using MPI and an optimized data exchange policy[J]. Parallel Computing, 2013, 39(12): 851-866.

[41] YIN J, FORAN A, WANG J. DL-MPI: enabling data locality computation for mpi-based data-intensive applications[C]// Proceedings of the 2013 IEEE International Conference on Big Data. IEEE, 2013: 506-511.

[42] SNIR M. Technical perspective: The future of MPI[J]. Communications of the ACM, 2018, 61(10): 105-114.

[43] RAMESH S, MAHÉO A, SHENDE S, et al. MPI performance engineering with the MPI tool interface: The integration of MVAPICH and TAU[J]. Parallel Computing, 2018, 77(1): 19-37.

[44] PIMENTA A, CÉSAR E, SIKORA A. Methodology for MPI applications autotuning[C]// 20th European MPI Users's Group Meeting. EuroMPI, 2013: 145-146.

[45] DEUZEMAN A, REKER S, URBACH C. Lemon: An MPI parallel I/O library for data encapsulation using LIME[J]. Computer Physics Communications, 2012, 183(6): 1321-1335.

[46] LI S, ZHANG Y, HOEFER T. Cache-oblivious MPI all-to-all communications based on morton order[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(3): 542-555.

[47] ALFATAFTA M, ALSADER Z, AL-KISWANY S. COOL: A cloud-optimized structure for MPI collective operations[C]// 11th IEEE International Conference on Cloud Computing. CLOUD, 2018: 746-753.

[48] DONATO D. Simple, effcient allocation of modelling runs on heterogeneous clusters with MPI[J]. Environmental Modelling and Software, 2017, 88(3): 48-57.

[49] SUBRAMONI H, HAMIDOUCHE K, VENKATESH A, et al. Designing MPI library with dynamic connected transport (DCT) of infiniband: Early experiences[C]// 29th International Conference Supercomputing. ISC, 2014: 278-295.

[50] ISLAM T, MOHROR K, SCHULZ M. Exploring the MPI tool information interface: features and capabilities[J]. IJHPCA, 2016, 30(2): 212-222.

[51] SHARMA A, MOULITSAS I. MPI to coarray fortran: Experiences with a CFD solver for unstructured meshes[J]. Scientific Programming, 2017, 55(2): 1-12.

[52] SHTERENLIKHT A, MARGETTS L, CEBAMANOS L. Modelling fracture in heterogeneous materials on HPC systems using a hybrid mpi/fortran coarray multi-scale CAFE framework[J]. Advances in Engineering Software, 2018, 12(5): 155-166.

[53] MATTSON P. A Programming System for the Imagine Media Processor[D]. Stanford: University of Stanford, 2002.

[54] IBM Corporation. Software development kit for multi-core acceleration[OL]. http://www.ibm.com/developerworks/power/cell.

[55] SCHNEIDER S, YEOM S, ROSE B, et al. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies[C]// Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, 2009: 131-140.

YUAN Liang, born in 1984, Ph.D, associate professor, is member of China Computer Federation (CCF). His main research interests include parallel computational models.

ZHANG Yun-quan, born in 1973, Ph.D, professor, Ph. D supervisor, is senior member of China Computer Federation (CCF). His main research interests include high performance computing and parallel processing of big data.