doi:10.11887/j.cn.202002004 http://journal.nudt.edu.cn

高精度CFD程序的内外子区域划分异构并行算法*

王 巍,徐传福,车永刚
(国防科技大学 计算机学院 量子信息研究所兼高性能计算国家重点实验室,湖南 长沙 410073)

摘 要:对计算流体力学(Computational Fluid Dynamics, CFD)程序CNS提出一种Offload模式下对任务内外子区域划分的异构并行算法,结合结构化网格下有限差分计算和四阶龙格-库塔方法的特点,引入ghost网格点区域,设计了一种ghost区域收缩计算策略,显著降低了异构计算资源之间的数据传输开销,负载均衡时CPU端的计算与MPI通信完全和加速器端的计算重叠,提高了异构协同并行性。推导了保证计算正确性的ghost区域的参数,分析了负载均衡的条件。在"CPU(Intel Haswell Xeon E5-2670 12 cores ×2)+加速器(Xeon Phi 7120A ×2)"的服务器上测得该算法较直接将任务子块整体迁至加速器端计算的异构算法性能平均提升至5.9倍,较MPI/OpenMP两级并行算法使用24个纯CPU核的性能,该算法使用单加速器时加速至1.27倍,使用双加速器加速至1.45倍。讨论和分析了性能瓶颈与存在的问题。

关键词:高精度CFD程序;四阶龙格-库塔法;异构并行算法;内外子区域划分;性能分析
中图分类号:TN95 **文献标志码:**A **文章编号:**1001-2486(2020)02-031-10

Inner-out subdomain dividing heterogeneous parallel algorithm for high order CFD solver

WANG Wei, XU Chuanfu, CHE Yonggang
(Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

Abstract: An Offload-mode heterogeneous parallel algorithm via inner-out subdomain dividing was proposed for CFD (computational fluid dynamics) program CNS. Combined with the characteristics of finite difference computing and fourth order Runge-Kutta method in structure mesh, the scheme of ghost region was introduced, based on which a Ghost-Region-Shrinking computing scheme was designed, significantly reducing the overhead of data movement between heterogeneous computing resources, making the computing and MPI communication on CPU absolutely overlap with the accelerator computing under load balance condition, bringing better heterogeneous synergetic parallelism. Parameter of the ghost region for the computing validity was given and load balance tuning was demonstrated. On a server with CPU (Intel Haswell Xeon E5-2670 12 cores ×2) + MIC (Xeon Phi 7120A ×2), an averaged performance improvement of 5.9× was gained over the algorithm of using accelerator with task blocks integrally. Compared with MPI/OpenMP two-level parallel algorithm running on 24 Intel Haswell CPU cores, the proposed method achieved an accelerating of 1.27× with one MIC and 1.45× with two MICs. Finally the bottleneck and disadvantage were discussed.

Keywords: high order CFD program; fourth order Runge-Kutta method; heterogeneous parallel algorithm; inner-out subdomain division; performance analysis

计算流体力学(Computational Fluid Dynamics, CFD)采用数值方法求解流体运动相关微分方程,从而揭示流动现象和规律,是涉及流体力学、数学和计算机科学等多个学科领域的交叉学科。20世纪60年代以来,随着高性能计算技术的发展,CFD也获得了迅速的发展,目前已被广泛应用于航空航天、航海、汽车制造、生物医疗、环境工程、核工业等众多领域。然而,在研究和设计中要使CFD手段达到高保真度水平将引入巨大的计算量,这对现代高性能计算平台的计算和存储能力等提出了极高的要求[1-2]。

另一方面,自2008年的P级计算机问世,超级计算机的发展向E级计算能力水平推进,以满足应用领域的迫切需求。这也使得一系列相关技术,如网络通信、存储结构等,都面临着新的挑战,然而最重要和最根本的是解决能耗问题[3]。为此,异构超级计算机的概念作为解决方案被提出来。概念上,异构超级计算机主要是指计算单元

* **收稿日期:**2019-10-10
基金项目:国家重点研发计划资助项目(2017YFB0202403);国家自然科学基金资助项目(61561146395, 61772542)
作者简介:王巍(1988—),男,湖南长沙人,工程师,硕士, E-mail:wangw111@icloud.com

集成了处理器和所谓的加速器，或称协处理器。目前一种被广泛使用的加速器是通用图形处理器（GPGPU），目前世界超级计算机排行榜（Top500）中最快的超级计算机——Summit 就使用了27 648 块 NVIDIA Volta V100 GPU 作为加速器。另一种重要的加速器是 Intel Xeon Phi 系列协处理器（MIC），这是一种基于片上系统架构的众核加速器，在“天河二号”、Stamped 等超级计算机中得到使用[4-5]。现今，异构系统正成为在能耗允许的范围内建造更大规模和更强计算能力的超级计算机的有力选择[6-7]。

然而，要使 CFD 应用在异构平台上获得良好的性能，将面临新的问题，如异构计算资源之间的数据传输开销、计算能力的差别，以及负载均衡等。针对异构超级计算机系统的特点，将 CFD 应用以合适的方式面向异构平台移植，是使 CFD 应用充分发挥异构超级计算机性能的关键技术，也是领域内的热点课题，已有大量相关研究工作。美国橡树岭国家实验室与 CRAY、NVIDIA 等[8]合作实现了燃烧模拟软件 S3D 的 MPI/OpenMP 混合并行，并基于 OpenACC 实现了其 GPU 并行计算，使用 Fermi GPU 相对于 CPU 的加速为 1.2 倍，在 Titan 超级计算机上的测试规模最大达 8192 节点。俄罗斯凯尔迪什应用数学研究所 Gorobets 等[9]基于 MPI + OpenMP + OpenCL 并行编程，实现了一个采用有限体积法求解可压缩 Navier-Stoke 方程的 CFD 软件的大规模并行，并在多核 CPU、MIC、GPU 等体系结构上进行了性能测试与对比，其大规模异构并行计算扩展到了 320 个 GPU，相对于 48 个 GPU 的并行效率超过 90%。西班牙加泰罗尼亚技术大学 Álvarez 等[10]基于 MPI + OpenMP + OpenCL 编程实现了 HPC2 框架，在此基础上实现了一个不均匀加热充气腔中的湍流流动算例的并行计算，在一个每结点含 2 块 NVIDIA K80 GPU 的集群上，从 1 个节点扩展到 64 节点时的并行效率达到 94%。国内 Cai 等[11]在神威太湖之光超级计算机上，针对极大规模三维爆震波模拟中高阶 WENO 计算问题设计了高效的并行算法和优化技术，测试表明可扩展到 998 万核，获得 23.1Pflops 的性能。

对此，本文针对一个基于有限差分和 4 阶龙格－库塔（Runge-Kutta）方法显式时间推进的高阶精度 CFD 求解器——CNS，提出了一种 Offload 模式下的，对任务内外子区域划分的异构混合算法，从而提升程序在异构平台上的单节点性能。工作和贡献如下：

第一，对 CNS 求解器在 Offload 模式下的异构程序引入了一种新的内外子区域的任务划分策略。这种内外子区域的任务划分策略把 CNS 求解器已有的任务子块再次划分为内部区域和外环区域两个子区域，将程序在内部区域的计算放在加速器端来执行，外环区域的计算和 MPI 通信放在 CPU 端执行。这种方式让加速器端的任务子块成为被外环区域包裹的彼此独立的内部区域块，这样使加速器的计算之间相互独立，没有数据关联；并且可以通过调整内部子区域和外部子区域的大小比例，来灵活地控制 CPU 端和加速器端的计算负载，获得良好的负载均衡。

第二，在内外子区域的任务划分策略基础上针对显式算法的特点进一步设计 ghost 网格点区域，并通过引入 ghost 区域提出了一种 ghost 网格点缩减计算模式，这种模式仅在一个时间推进步开始和结束的时候进行 CPU 和加速器之间的数据传输，而在 4 阶 Runge-Kutta 法计算的内部则完全不进行 CPU 和加速器之间的数据通信。这极大地减少了 Offload 模式下异构程序异构计算资源之间的数据传输开销，并且使得 CPU 端的任务和加速器端的任务在一个时间推进循环内部，完全独立地各自分别进行，仅随着循环开始和结束同步，这样，在负载均衡的情况下，CPU 端的计算和 MPI 通信，与加速器端的计算完全地重叠起来。给出了可以保证计算正确性的 ghost 网格点区域的宽度。

第三，基于 OpenMP 4.5 实现了该异构算法，并分析了负载均衡的条件，最后对程序在 CPU + MIC 结构的平台上进行了性能测试。测试表明，基于内外子区域划分的异构算法显著提升了 CNS 程序在异构平台上的性能，相较于已有的直接以网格子块为单位进行任务分配的异构并行算法，在本文测试的三种数据规模下，平均有 5.9 倍的性能提升。在单节点上，相对于不使用加速器时，基于内外子区域划分的异构算法在使用单块 MIC 卡和两块 MIC 卡时，分别获得了最大 1.27 倍和 1.45 倍的性能加速。

1 高精度 CFD 求解器 CNS

CNS（compressible Navier Stokes）程序是一款用于求解空气动力学问题的高精度 CFD 求解器，是一款 in-house 代码。该程序提出的稳定有限差分格式，满足能量估计和分部求和规则[12]，求解全三维可压缩 Navier-Stokes 方程组。CNS 在航空

航天科研领域具有良好的应用背景,同时,程序包含复杂的计算。

1.1 CNS 中的控制方程组

CNS 求解的控制方程组包括方程(1)~(4):

$$\frac{\partial \boldsymbol{U}}{\partial t}=\boldsymbol{\Phi} \tag{1}$$

$$\boldsymbol{\Phi}=\boldsymbol{J}-\left(\frac{\partial \boldsymbol{F}}{\partial x}+\frac{\partial \boldsymbol{G}}{\partial y}+\frac{\partial \boldsymbol{H}}{\partial z}\right) \tag{2}$$

$$p=P(\gamma,\rho,e) \tag{3}$$

$$T=Temp(\gamma,Ma,\rho,p) \tag{4}$$

方程(1)和(2)是守恒形式的 Navier-Stokes 方程。方程(1)中 $\boldsymbol{U}=\left[\rho\ \rho u_1\ \rho u_2\ \rho u_3\left(e+\frac{\boldsymbol{V}^2}{2}\right)\right]^{\mathrm{T}}$ 称为解向量[13],里面包含了控制方程的基本未知量(气体密度 ρ;坐标轴上的速度分量 u_1、u_2 和 u_3;单位质量气体的内能 e;$\boldsymbol{V}$ 为速度矢量)及基本未知量的代数组合,其各个元素也被称作守恒变量。方程(2)中 $\boldsymbol{F}$、$\boldsymbol{G}$ 和 $\boldsymbol{H}$ 称为通量向量,并且通常表示为两项的和(黏性项和非黏性项):$\boldsymbol{F}=\boldsymbol{F}_I+\boldsymbol{F}_\nu$、$\boldsymbol{G}=\boldsymbol{G}_I+\boldsymbol{G}_\nu$、$\boldsymbol{H}=\boldsymbol{H}_I+\boldsymbol{H}_\nu$。同时,$\boldsymbol{J}$ 是源项,通常外界对流动的贡献如重力、热源被写入源项当中。方程(3)是内能方程,p 为气体的压力。方程(4)是状态方程,式中 T 为气体的温度,Ma 为马赫数。方程(3)和(4)中 γ 为常数,通常,对于标准条件下的空气取 $\gamma=1.4$。

1.2 基本的数值方法和程序

CNS 程序基于结构化网格求解控制方程组,网格点沿着 xyz 方向均匀分布,如图 1,针对网格点 i,j,k,图中用灰色表示,在计算某一个坐标方向的数值偏导数的时候,将使用前后各 6 个相邻的网格点的数值来构建差分形式(xyz 方向分别为图 1 中绿色、蓝色、橙色)。同时,时间推进采用显式 4 阶 Runge-Kutta 方法,并且所有的计算矩阵都是稠密的。这些不仅带来巨大的数据量和计算量,也增加了 MPI 通信开销和使用异构平台时异构计算资源之间的数据传输开销。

CNS 程序的原始版本基于 MPI 的并行程序,它将一个正六面体空间计算区域均匀地分解成 $nprocs_x \times nprocs_y \times nprocs_z$ 个子块,采用笛卡尔进程通信拓扑,每一个进程承担一个子块的计算,每一个任务子块有相同数量的网格,这样负载是均衡的。初始版本的 CNS 程序本身具有良好的并行可扩展性。

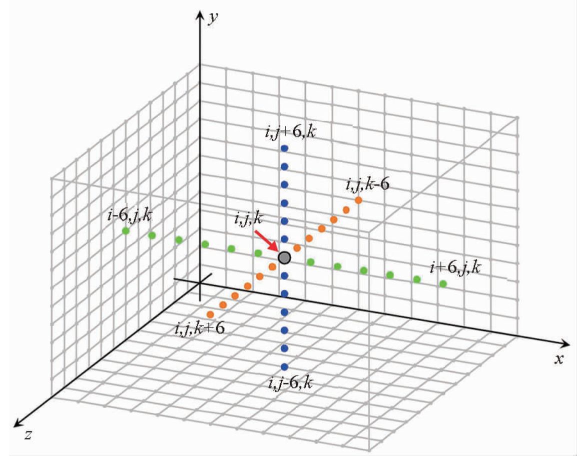


图 1　CNS 程序的网格和差分示意

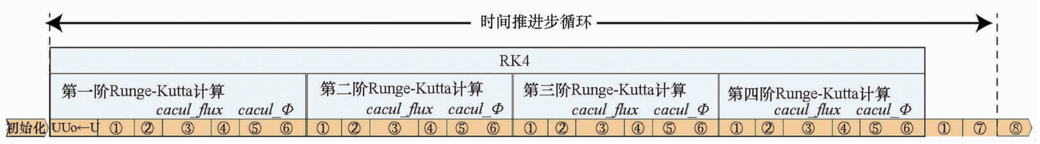
Fig. 1　Finite difference stencil in CNS

CNS 程序主循环的计算内核 RK4 包含了 4 阶 Runge-Kutta 法的 4 个计算阶段,如图 2(a)所示。控制方程组所有项和结果的计算都在 RK4 函数中执行,其中包括两大主要计算模块:通量项的计算,即函数 *cacul_flux*,如图 2(a)中步骤③所示;$\boldsymbol{\Phi}$ 项的计算,即函数 *cacul_*$\boldsymbol{\Phi}$,如图 2(a)中步骤⑤所示。*cacul_flux* 和 *cacul_*$\boldsymbol{\Phi}$ 包括了 CNS 程序全部的差分操作,为了保障 *cacul_flux* 和 *cacul_*$\boldsymbol{\Phi}$ 中的差分计算在边界附近可以顺利执行,程序通过 MPI 通信对子块外围的数据缓冲区进行了填充,如图 2(a)步骤②和④所示。

2　Offload 异构模式和已有的工作

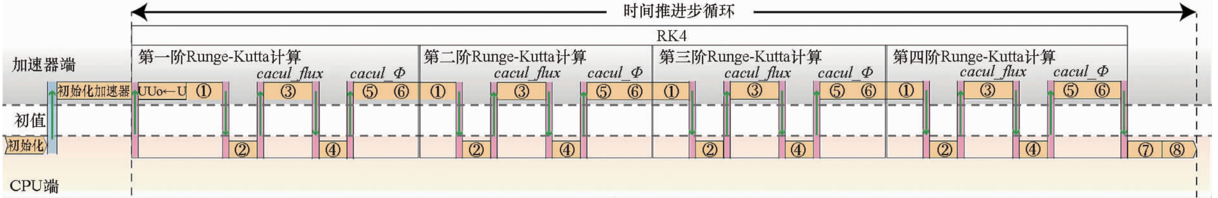
Offload 模式是以 CPU 为主,加速器为辅的一种异构执行模式。相对于加速器和 CPU 对等模式、加速器原生模式等来说,Offload 模式更有利于异构平台计算资源的充分利用和 CPU 端与加速器端的负载均衡。所以,一般使用 Offload 模式来构建和执行异构程序。

由于 CNS 程序原始代码采用一个进程计算一个任务子块的模式,要将其重构成 Offload 模式的异构程序,最直接的做法是将一部分子块(进程)的计算迁移到加速器上去,将一部分子块的计算保留在 CPU 端进行,后面我们将在加速器上计算的子块称为 Offload 子块,在加速器上计算的子块对应的进程称为 Offload 进程。这样,加速器和 CPU 上的计算资源都得到了利用,并可以通过调整在加速器上计算的子块和在 CPU 上计算的子块的数量比例,调整 CPU 端和加速器端的负载。



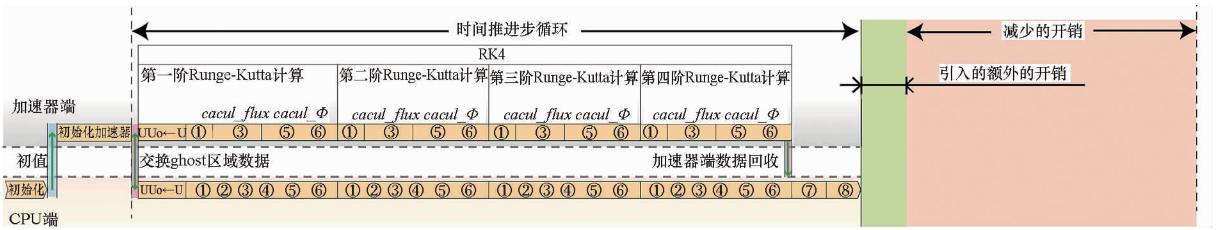
（a）原始 MPI 并行算法过程示意

（a）Original algorithm procedure



（b）Direct-Offload 算法过程示意

（b）Direct-Offload algorithm procedure



（c）Subdomain-Offload 算法过程示意

（c）Subdomain-Offload algorithm procedure

注：①还原基本未知量，边界条件加载；②MPI 通信，填充基本未知量边界；③计算通量项 $\boldsymbol{F}$、$\boldsymbol{G}$、$\boldsymbol{H}$；④MPI 通信，填充通量项边界；⑤计算 $\boldsymbol{\Phi}$ 项：*cacul_Φ*；⑥更新 U 数组；⑦收敛检验、文件输出等；⑧结束工作

图 2　三种算法的执行过程

Fig. 2　Procedure of three involved algorithm

本课题组的王铜铜[14]等，在 CNS 程序原始代码的基础上实现 MPI/OpenMP 的两级并行后，在一个以 MIC 协处理器作为加速器的平台上，基于 OpenMP 4.5 尝试了上述方式来构建 Offload 模式下的异构程序。王铜铜等的工作显著提高了 CNS 程序在 CPU 和 MIC 上的单核计算性能，OpenMP 并行效率高，在“天河二号”超级计算机上的大规模并行测试展示了良好的并行可扩展性，然而其 CPU + MIC 异构并行算法相对于纯 CPU 上 MPI + OpenMP 并行算法尚无性能加速。

通过分析，并参考同类研究，认为造成这种情况的重要原因之一是异构计算资源之间频繁的数据传输带来的开销影响了程序整体的性能。在上述的异构算法中，Offload 进程的执行过程可以概括为图 2(b)，可以看出，对于 Offload 进程，除了在时间推进步开始和结束时 CPU 和加速器端的数据传输和回收以外，每次当计算内核 RK4 执行到步骤②和④都需要将加速器端的中间计算结果传回 CPU 端进行 MPI 通信，再次传至加速器端继续计算。如图 2(b)所示，计算内核 RK4 的内部，由步骤②和④引起的异构计算资源的 PCIe 数据传输共有 16 次，这种数据传输开销很大。另外，这也破坏了 RK4 在加速器端执行的连续性，使得 RK4 内部的 OpenMP 并行区无法尽可能地合并，增加了线程管理开销。为了便于描述，本文后面将这种 Offload 模式下，把部分任务子块整个地迁移到加速器上计算的异构算法称为 direct-Offload 算法(代码)。

3　基于内外子区域划分的异构算法

为此，不得不寻求新的方式来构建异构程序，提升程序在异构平台上执行的性能。结合前述 direct-Offload 程序的经验，从如何降低异构计算资源之间的数据传输开销出发，设计了新的异构算法。Yang 等[6]在将基于有限差分的 HPCG (high performance conjugate gradient benchmark)程序向“CPU + 加速器(MIC)”的异构平台移植时，引入了一种对任务子块的内外子区域划分策略，使降低 PCIe 数据传输开销成为可能，进而使异构程序获得了较好的性能和加速。受此启发，

得益于 CNS 程序本身规则的任务子块和结构化网格的便利，本文借鉴了这种内外子区域再划分的任务再划分策略，并结合 4 阶 Runge-Kutta 法这种显式时间推进算法的特点，成功设计了一种适合 CNS 求解器的基于内外子区域划分的 ghost 区域收缩式计算异构算法。

该算法极大地降低了 Offload 模式下异构程序 CPU 端和加速器端的数据传输开销，提升了 CNS 程序在异构平台上的运行性能，我们从四个方面来介绍和讨论该算法：

1)任务子块的内外划分策略和 ghost 区域；
2)算法的执行过程；
3)ghost 区域的宽度和收缩计算；
4)代码实现与负载均衡。

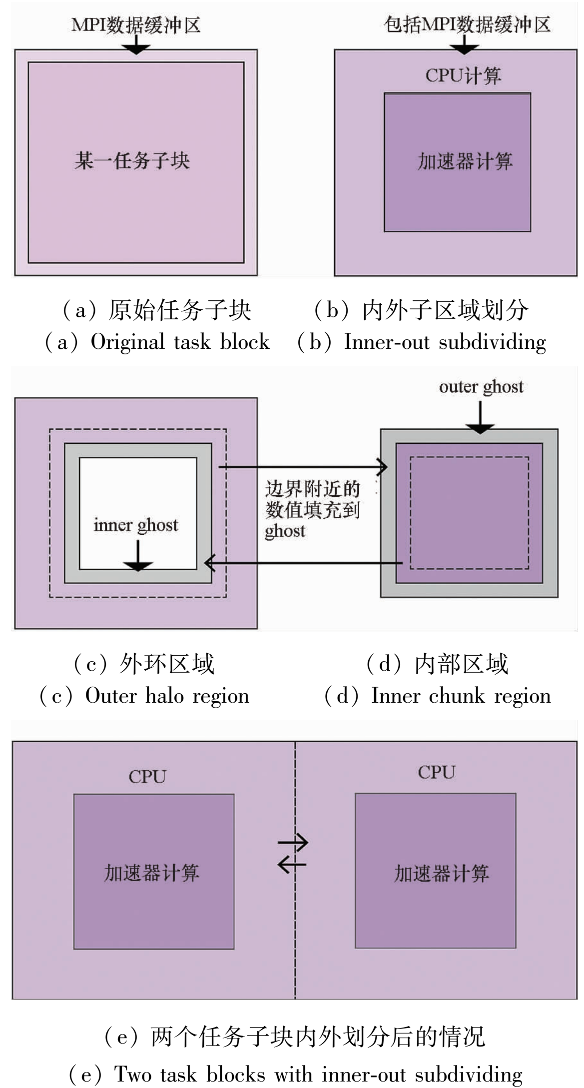
3.1　任务子块的内外划分策略和 ghost 区域

基于二维的情况来描述 CNS 程序的任务子块，图 3(a)是原始 MPI 并行版本的程序的任务子块，本文对其进行如图 3(b)所示的内外子区域再划分，任务子块被再划分成内部区域和外环区域。原来的任务子块是一个整体，包裹在子块外围的是 MPI 数据缓冲区，如图 3(a)所示；进行内外子区域再划分后，原来的任务子块成为两部分：由加速器来计算的内部区域，和由 CPU 端负责的外环区域。值得注意的是，外环区域包含了原来的 MPI 通信数据缓冲区和实际计算的网格点两部分，CPU 端的任务包括了计算和通信两部分。实际上，对于三维的任务子块，本文将立方体区域划分成内部的小立方块和外部具有一定厚度的空心立方壳两部分，其在二维情况下则退化为图中矩形内部区域和框形外环区域，因此，从二维情况来讨论已经足够表达基于内外子区域划分的异构算法的本质，同时，方便描述和理解。

所谓 ghost 区域，是指外环区域内边界内部保留的一定宽度的网格点区域，和内部区域外周界外部保留的一定宽度的网格点区域，如图 3(c)和(d)灰色部分所示，分别称为 inner ghost 区域和 outer ghost 区域。外环区域的 inner ghost 区域将保存内部区域在外周界附近的计算数值，内部区域的 outer ghost 区域将保存外环区域在内边界附近的计算数值。当有两个子块分别使用两块加速器时，任务划分和执行的情况则如图 3(e)所示。

3.2　算法的执行过程

基于上述内外划分策略，原来在整个任务子块执行的计算内核 RK4，也随之划分为内部区域的 RK4 计算和外环区域的 RK4 计算，分别在加速器端和 CPU 端执行。



(a) 原始任务子块　(a) Original task block
(b) 内外子区域划分　(b) Inner-out subdividing
(c) 外环区域　(c) Outer halo region
(d) 内部区域　(d) Inner chunk region
(e) 两个任务子块内外划分后的情况　(e) Two task blocks with inner-out subdividing

图 3　内外子区域任务划分策略
Fig. 3　Inner-out subdomain dividing strategy

对于一个任务子块，重新设计的异构算法过程如图 2(c)所示。

Offload 进程首先初始化加速器端的执行环境，并将初值等传递到加速器端，该操作是在主循环的外部完成的。

随之进入时间推进主循环，在计算内核 RK4 开始之初，通过 PCIe 将内部区域边界附近的值传回 CPU 端，并保存在外环区域的 inner ghost 网格点上，同时，将外环区域内边界附近的值传至加速器端，并保存在内部区域的 outer ghost 网格点上。

随后加速器端独立地在内部区域上进行 RK4 的 4 阶计算，步骤②和④的 MPI 通信被完全交给了在 CPU 端执行的外环区域，在加速器端 RK4 的计算过程内部不再与 CPU 端通信，在 outer ghost 区域已经保存了需要的外环区域的值，从而完整的、连续地执行，直到一次循环结束。

CPU 端同时进行外环区域 RK4 的计算,并且进行步骤②和④来与其他进程通信以填充子块外围 MPI 缓冲区的值,而 inner ghost 网格点提供计算外环区域内边界附近的差分所需的内部区域边界附近的值,这样 CPU 端也连续地执行其任务,直到一次循环结束前都不需要与加速器进行数据传输。

最终,一次循环结束时,将内部区域的计算结果从加速器端回收到 CPU 端用以收敛检验和文件输出等。

重新设计的异构算法,整个主循环的执行过程仅随着循环的开始和结束进行两次 PCIe 数据传输。参照直接将子块整个地迁移到加速器端计算的 direct-Offload 算法的过程,如图 2(b)所示,重新设计的异构算法一方面减少了异构计算资源之间的 PCIe 数据传输次数;另一方面,也比直接将整个子块的数据在 CPU 和加速器端来回转移具有更少的数据传输量。

重要的是,得益于显式算法的便利性,对于 CNS 程序,这样的异构算法可以保证计算结果正确,因为 ghost 区域的宽度,是计算结果正确性的关键。

3.3 ghost 网格点区域的宽度和收缩计算

以内部区域的 outer ghost 为例,来讨论 ghost 区域的宽度。设子块进行内外划分后,内部区域在 x 方向上有 nn_x+1 个网格点,y 方向上有 nn_y+1 个网格点,z 方向上有 nn_z+1 个网格点,二维情况下,nn_z+1 可以为 1(或是任意正整数,设它为 1 即可)。

由于 CNS 程序在 xy 方向的差分计算,在形式上没有任何区别,其 ghost 区域的宽度是一样的。设 xy 方向 outer ghost 区域的宽度都为 W_g 个数据点。从图 1 中可知,CNS 程序在计算一个网格点在某一个方向的数值导数时,将使用该网格点前后各 6 个网格点的数值,xy 坐标方向差分计算形式有如下形式:

$$A_{i,j}=f(B_{i-1,j},B_{i-2,j},B_{i-3,j},B_{i-4,j},B_{i-5,j},B_{i-6,j},B_{i+1,j},B_{i+2,j},B_{i+3,j},B_{i+4,j},B_{i+5,j},B_{i+6,j}) \tag{5}$$

$$C_{i,j}=f(B_{i,j-1},B_{i,j-2},B_{i,j-3},B_{i,j-4},B_{i,j-5},B_{i,j-6},B_{i,j+1},B_{i,j+2},B_{i,j+3},B_{i,j+4},B_{i,j+5},B_{i,j+6}) \tag{6}$$

式中,B 表示泛指程序中某一变量,式(5)和(6)分别计算了变量 B 在 xy 两个方向的数值导数并赋值给了变量 A 和变量 C。

在 *cacul_flux* 中,CNS 程序对物理场变量(如速度、温度等)的空间 x 方向数值偏导数的计算,都具有上述式(5)的形式,y 方向的数值偏导数的计算形式如式(6)所示。而 *cacul_Φ* 的计算,是对 *cacul_flux* 差分计算所得的结果,即通量项($\boldsymbol{F}$,$\boldsymbol{G}$,$\boldsymbol{H}$),进一步差分得到 $\boldsymbol{F}$ 项的,其计算形式同样具有式(5)和(6)的形式。进一步,以 x 正方向的情况为例,规定内部区域(不包括 ghost 点)沿着 x 正方向的第一个网格点的 i 下标为 0,下标沿着 x 正方向递增,那么其 ghost 区域在 x 正方向的最后一个点的 i 下标为 nn_x+W_g。在 *cacul_flux* 中,ghost 区域能保障式(5)形式的差分能够正确计算的网格点的 i 下标的上界需要满足条件:$i+6=nn_x+W_g$,显然可算得循环下标 i 的上界为 nn_x+W_g-6。

在计算完 *cacul_flux* 的结果的基础上进一步执行 *cacul_Φ*,显然要保证 *cacul_Φ* 中式(5)形式的差分能够正确计算的网格点的 i 下标的上界需要满足条件:$i+6=nn_x+W_g-6$,因此能够正确计算 *cacul_Φ* 的网格点的循环 i 下标的上界为nn_x+W_g-12。

观察图 2(c)不难发现,在 RK4 的第一阶计算中包含差分计算的函数就是 *cacul_flux* 和 *cacul_Φ* 这两个函数,第二阶计算在第一阶计算的基础上重复与第一阶类似的过程,因此第二阶计算只能在 i 上界为 nn_x+W_g-12 的范围内进行。同理,第三阶计算只能在 i 上界为 nn_x+W_g-24 的网格点范围内进行,第四阶计算只能在 i 上界为 nn_x+W_g-36 的网格点范围内进行,第四阶计算完,计算结果正确的网格点的 i 下标的上界为 nn_x+W_g-48,式(5)和(6)有完全相同的形式,并且是以 i,j 号网格点中心对称的,所以 i 下标的下界和 j 下标的上下界都有上述"收缩"的规律。将这个 ghost 区域以 12 个网格点宽度,逐阶收缩的过程表示为图 4,浅灰色表示最初的 ghost 区域,深灰色表示计算完一阶后向内收缩了一层。

显然,保证 RK4 的四阶计算结束后内部区域的网格点计算结果正确的条件是 $nn_x+W_g-48=nn_x$,则有 $W_g=48$。因此内部区域 ghost 网格点的宽度至少为 48 个网格点,外环区域的计算和内部区域的计算是一样的,可证其 ghost 区域的宽度也需满足这个条件。取 ghost 区域的宽度为 48 个网格点即可,我们没必要取更宽的 ghost 区域,因为更宽的 ghost 区域除了增加主循环开始时 CPU 和加速器之间的数据传输量,以及带来更多不必要的计算以外,没有任何意义。三维情况下,上述收缩计算在 z 方向同样执行,所以对于三维的立方块,ghost 区域在 xyz 方向的宽度都为 $W_g=48$。

值得注意的是,在这种算法中,应该保证外环

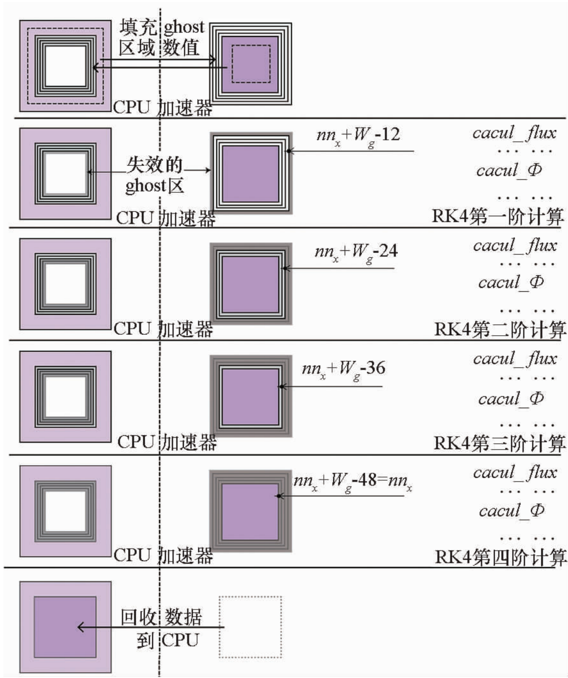


图 4　ghost 区域的收缩计算

Fig. 4　ghost-region-shrinking computing scheme

区域和内部区域本身的大小，要使其能够提取足够的网格点来填充 ghost 区域的数值。

3.4　代码实现与负载均衡

OpenMP 4.5 对 Offload 模式的异构程序开发与移植已经有较好的支持，通过合适的编译指导语句，便可以将代码段放在加速器端执行，并且支持数据异步传输等操作，同时支持多平台，代码具有良好的可移植性。在已有的 CNS 程序的 MPI/OpenMP 两级并行代码的基础上，基于 OpenMP 4.5，实现了上述基于内外子区域划分的 Offload 模式的异构算法。为了叙述的方便，后面将基于内外子区域划分的 Offload 模式的异构代码称为 subdomain-Offload 算法（代码）。

MIC 卡支持包括 OpenMP 在内的多种并行编程模型。将 subdomain-Offload 代码放在本课题组一个 CPU + MIC 结构的服务器单节点上进行调试和运行，并检验了代码执行和计算结果，与原来的纯 CPU 代码是完全一致的。

CNS 程序本身子块具有相同的大小，对于 subdomain-Offload 算法来说，将每一个进程的子块都进行相同的内外子区域划分，并将内部区域放在加速器端计算，外环区域放在 CPU 端计算，这样进程和进程之间仍是负载均衡的，而更重要的是 CPU 和加速器之间的负载均衡。

从 subdomain-Offload 算法的过程来讲，在主循环的一次执行过程中，CPU 端的计算和加速器端的计算在 RK4 内部完全独立的执行，仅在主循环开始和结束的时候存在同步和进行 PCIe 数据传输。因此，通过控制内外子区域划分的比例调整 CPU 端和加速器端的负载分配，完全可以做到使一次主循环中的 CPU 端外环区域的计算及 MPI 通信和加速器端内部区域的计算在时间上几乎相等。本文通过实测对此进行了验证和分析。

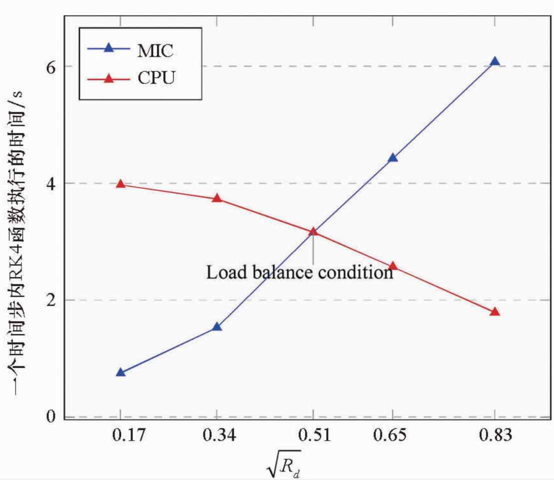
在配置如表 1 所示服务器上，不失一般性地，以使用一块 MIC 加速卡计算单个任务子块的情况为例进行测试分析，将一个数据规模足够大的子块划分为内部区域和外环区域后，将内部区域的网格点数占整个子块网格点数的比例记作 R_d。为了方便，取子块为正方形，并且内外划分也是正方形的。保持子块的数据规模不变，调整 R_d，分别记录 CPU 端执行 RK4 的时间、MIC 端 RK4 的执行时间以及一次主循环的执行时间，结果如图 5所示。当 R_d 逐渐增大，加速器端承担的计算增加，加速器执行 RK4 的时间随之增加，CPU 端承担的计算逐渐减少，CPU 端执行 RK4 的时间随之减少，如图 5（a）所示，对于加速器来说，当 R_d 为某一个值的时候，加速器端和 CPU 端执行 RK4 的时间相等，随后 MIC 端的执行时间继续增加，CPU 端的执行时间继续减少，在这个过程中，程序一次主循环执行的时间，如图 5（b）所示，先随着 CPU 端执行 RK4 时间的减少逐渐减少，然后随着加速器端的执行时间增加而增加，当 $\sqrt{R_d}=0.51$（即 $R_d=0.26$）时，加速器端和 CPU 端执行 RK4 的时间相等，程序主循环执行一次的时间最短，认为此时 CPU 端和加速器端负载是均衡的。

表 1　服务器的配置

Tab. 1　The Configuration of the Server

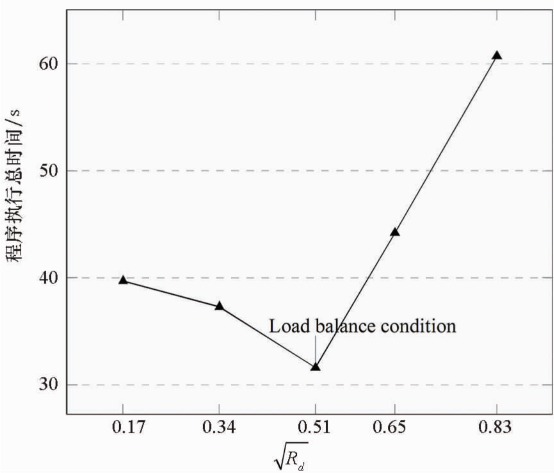
	CPU	MIC
Name	Intel Xeon E5 – 2670 12 – core ×2	Intel Xeon Phi 7120A ×2
Cores	12/CPU	61
Memory capacity	128 GB	16 GB
Peak Performance (DP)	998.4 Gflops	1208.3 Gflops

由于 CNS 程序本身的任务子块都划分成了相同的网格点数，而直接将部分子块整个地传输到加速器端计算的 direct-Offload 算法，虽然可以调整 CPU 端和加速器端计算的负载，但是这种调整必须是以子块为单位的，在实际操作中，这其实



（a）RK4 函数单步执行时间

（a）Single step execution time of RK4 function



（b）程序执行总时间

（b）Total program execution time

图 5　使用一块加速卡时的负载均衡

Fig. 5　Load balance tuning with one accelerator

是难以真正做到 CPU 和加速器之间达到负载均衡的。所以相对于 direct-Offload 算法，subdomain-Offload 算法有更好的负载均衡。

特别地，当负载均衡的时候，CPU 端的计算和 MPI 通信与加速器端的计算在时间上可以完全重叠，程序有更好的异构协同并行性。

4　性能测试与分析

进一步在表 1 配置的服务器上对 subdomain-Offload 算法的单节点性能进行了测试和评估。选取三种较大的数据规模，网格点数分别为 2.88×10^6、5.76×10^6 和 11.52×10^6，在测试的过程中，CPU 端和 MIC 卡上的核心全部被用满，并关闭了 CPU 端超线程开关。测试了 direct-Offload 算法使用单块加速卡的情况，以及重新设计的 subdomain-Offload 算法使用一块和两块 MIC 卡的情况，同时，测试了只使用 CPU 核计算的 MPI/OpenMP 两级并行代码的情况作为对照。

4.1　性能测试

图 6 中蓝色图例是 direct-Offload 算法使用单块加速卡 MIC 的情况，红色图例是 subdomain-Offload 算法使用 MIC 的情况。结果显示，在测试的三种数据规模下，direct-Offload 算法的程序执行总时间平均为 subdomain-Offload 算法执行总时间的 5.9 倍。subdomain-Offload 算法相较于直接将部分子块整个传输到加速器端计算的 direct-Offload 算法性能有较大提升，subdomain-Offload 算法相较于 direct-Offload 算法一方面极大地消除和降低了异构计算资源之间的数据传输，另一方面，subdomain-Offload 算法比 direct-Offload 算法做到更好的负载均衡，这些对 CNS 程序在异构平台的性能提升是有效的。

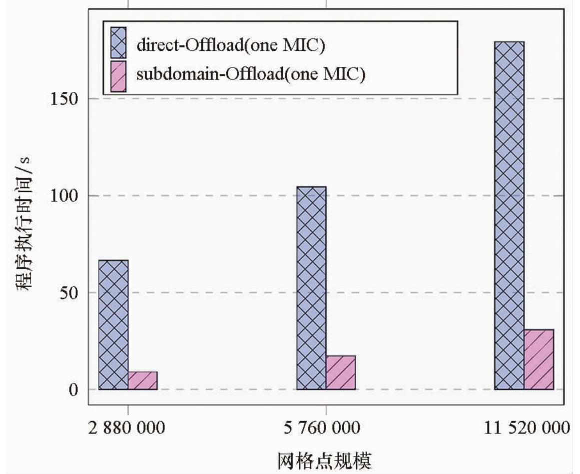


图 6　相对于 direct-Offload 算法的性能提升

Fig. 6　Performance improvement over direct-Offload

图 7 是 subdomain-Offload 算法分别使用单块 MIC 以及两块 MIC 卡的情况。在单块加速卡的性能测试中，使用一块 MIC 加速卡计算内部区域，24 个 CPU 核全部计算外环区域，任务分配如图 3（b）所示，依据负载均衡，使内部区域的网格点的数目占总区域网格点数目的 26% 左右，理论上应获得的性能加速约为 1.35 倍。实际测试结果如图 7 中蓝色图例所示（黄色图例是纯 CPU 核计算的情况）：纯 CPU 计算的程序执行时间分别为三种规模下 subdomain-Offload 使用 CPU + MIC 计算执行时间的 1.23 倍、1.24 倍、1.27 倍，实际的性能加速低于理论的性能加速，造成实际性能加速低于理论性能加速的因素包括异构计算资源之间的数据传输开销，以及计算 ghost 网格点的额外开销等。

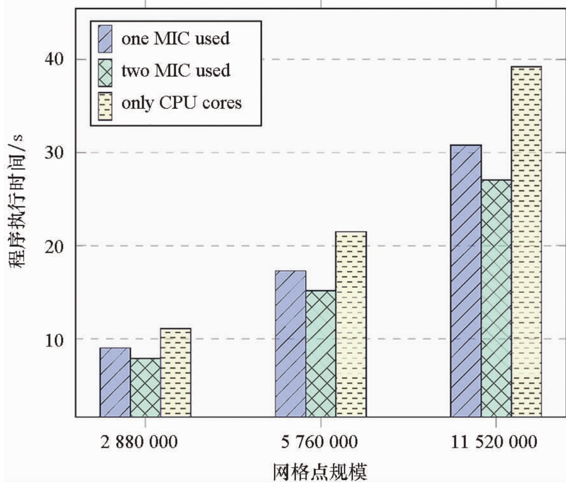


图 7　服务器上 subdomain-Offload 算法的性能加速

Fig. 7　Acceleration of subdomain-Offload on server

在使用两块加速卡的性能测试中，任务分配如图 3(e)所示，任务被分配到两个进程中，两块加速卡计算各自任务子块的内部区域，两块 MIC 加速卡共同承担总计算任务的 42% 左右，24 个 CPU 核心此时承担的外环区域的计算任务约占总计算任务的 58%，此时负载是均衡的，理论上，可以获得 1.72 倍左右的性能加速。实际测试结果如图 7 中绿色图例所示：纯 CPU 计算的程序执行时间分别为三种规模下 subdomain-Offload 使用 CPU +2MICs 计算执行时间的 1.40 倍、1.42 倍、1.45 倍。

结果表明，subdomain-Offload 异构算法，使用 MIC 卡加速器时可以获得加速，在测试的服务器上，最大获得了 1.45 倍的性能提升。比较三种数据规模的结果，发现单节点计算的数据规模越大，性能提升相应增加，使用两块加速卡比使用一块加速卡能获得更多的性能提升。

4.2　问题分析

subdomain-Offload 算法使 CNS 程序在异构平台上得到了性能提升，但加速效果并不理想，在测试的数据规模中，使用一块加速卡时的最大性能提升为 1.27 倍，使用两块加速卡的最大性能提升为 1.45 倍，原因有以下几个方面。

首先，测试节点上的 MIC 加速卡在 CNS 程序中能发挥的计算性能，较 CPU 能发挥的计算性能是更劣势的。结合图 5，不难发现，随着 R_d 的增大，MIC 上承担的计算任务增加，CPU 端的计算任务减少，计算任务从 CPU 端转移到 MIC 上，MIC 卡的任务执行时间剧烈的增加(图 5 (a)中的蓝线比红线更陡)，图 5 (b)中主循环的执行时间也随着 MIC 上任务的执行时间增加而剧烈增加，这说明节点由 CPU 主要承担计算转向 MIC 主要承担计算后，整个节点的等效计算能力是降低的。并且当负载均衡时，一块 MIC 卡在 CNS 程序中发挥的计算性能仅相当于 CPU 计算性能的 35%(R_d = 0.26 时负载均衡，MIC 的计算时间和 CPU 相等)。这意味着使用一块加速卡最多也只能获得性能提升至 1.35 倍，两块卡最多提升至 1.72 倍，这还没有考虑数据传输等引起的开销。

其次，subdomain-Offload 算法在一次时间推进步中，仍需进行两次 PCIe 数据传输，引起的开销是不可忽略的。在每一次主循环开始的时候，CPU 端和加速器端需要通过 PCIe 交换 ghost 区域的数值，这是保证 subdomain-Offload 算法正确性所需要的。这不仅带来 PCIe 传输开销，对填充 ghost 区域的数值在传输前后进行打包与释放，以及 ghost 网格点本身也产生额外的计算，这些都将引入一些开销。二维情况下，任务子块的网格点总数为 $NX \cdot NY \cdot 1$(二维情况，可令 $NZ = 1$)，内部区域的网格点数为$(NX \cdot \delta) \cdot (NY \cdot \xi)$。$\delta$ 和 ξ 分别为内部区域 xy 方向上的长度占整个任务子块对应长度的比例，即有 $R_d = \delta \cdot \xi$。那么 inner ghost 网格点的数量可以表示为$(2W_g + NX \cdot \delta) \cdot (2W_g + NY \cdot \xi) - NX \cdot \delta \cdot NY \cdot \xi$，outer ghost 区域的网格点数为 $NX \cdot \delta \cdot NY \cdot \xi - (2W_g - NX \cdot \delta) \cdot (2W_g - NY \cdot \xi)$，则 ghost 区域网格点的总数为两者之和，可以整理为 $4W_g \cdot (NX \cdot \delta + NY \cdot \xi)$。综上，引入的 ghost 网格点的总数所占任务子块网格点数的比例 P_g 可以表示为：

$$P_g = \frac{4W_g \cdot (NX \cdot \delta + NY \cdot \xi)}{NX \cdot NY} \tag{7}$$

设所有的区域都是方形的，即 $NX = NY = N$ 且 $\delta = \xi = \sqrt{R_d}$，则式(7)可以进一步简化为：

$$P_g = \frac{8W_g \cdot \sqrt{R_d}}{N} \tag{8}$$

从式(7)或式(8)中不难发现 ghost 网格点数量占任务子块网格点数量的比例 P_g 是随着任务子块网格点数的增加而降低的，因此，为了使 ghost 网格点引起的开销对整个任务子块计算性能的影响减小，就需要设置较大的子块数据规模，如果我们在一个 $N = 500$ 的方形区域上，若 R_d = 0.26，那么由式(8)所估算出来 ghost 网格点所占任务子块总网格点数量比例将达到 40%。这时由 ghost 网格点引起的传输和计算等开销对程序整体执行性能的影响将是十分明显的。可以通过增加子块网格点数规模来降低 ghost 区域带来的影响，但回收内部区域计算结果带来的传输开销是伴随着加速器端任务增加而增加的。

再次,内外子区域划分,造成 CPU 计算的外部区域的形状是不规则的,降低了 CPU 端计算的访存效率等,也是使异构程序性能受限的原因。

最后,测试平台上的 Haswell CPU 的计算能力也很强,其峰值性能接近单块 MIC 卡的峰值性能,因此 CPU + MIC 异构并行相对于纯 CPU 并行的性能加速效果不那么显著。

5 结论与未来工作

本文重新设计了高精度 CFD 求解器 CNS 在 Offload 模式下的异构程序,结合结构化网格下有限差分计算和 4 阶 Runge-Kutta 显式时间推进方法的特点,提出了一种基于对任务子块内外子区域再划分的"CPU + 加速器"异构混合并行算法,即 subdomain-Offload 算法。该算法从降低异构计算资源之间的 PCIe 数据传输开销出发,同时改善了 CPU 和加速器之间的负载均衡,明显地提高了异构程序的单节点性能。测试和分析结果表明:

1)subdomain-Offload 算法相对原有异构并行版本(direct-Offload 算法)有平均 5.9 倍的性能提升,降低异构计算资源之间的数据传输开销,改善 CPU 和加速器的负载均衡对 CNS 异构程序的性能提升是显著的。

2)基于内外子区域划分的异构并行算法相较于纯 CPU 的 MPI/OpenMP 两级并行算法使用双 Intel Haswell CPU(Xeon E5 - 2670,每 CPU 12 核)计算的情况,使用单 MIC 卡(Xeon Phi 7120A)时的最大性能加速为 1.27 倍,使用双 MIC 卡时的最大性能加速为 1.45 倍。

本文引入的基于内外子区域划分的异构协同并行算法可进一步推广到结构化网格下基于有限差分和显式求解偏微分方程的其他问题,具有良好的可迁移性。未来,将从提高 CNS 程序在 MIC 加速器端的单核计算能力和整体计算性能着手,进一步提升 subdomain-Offload 异构算法的单节点性能,包括进行向量化、编译指令优化等。同时,将尝试将 subdomain-Offload 算法向其他结构的异构平台(如 CPU + GPU)进行移植并观察其性能表现。

参考文献(References)

[1] Tu J Y, Guan H Y, Liu C Q. Computational fluid dynamics: a practical approach [M]. Burlington: Butterworth-Heinemann, 2008.

[2] Che Y G, Yang M F, Xu C F, et al. Petascale scramjet combustion simulation on the Tianhe-2 heterogeneous supercomputer[J]. Parallel Computing, 2018, 77: 101 - 117.

[3] Agerwala T. Exascale computing: the challenges and opportunities in the next decade [J]. ACM SIGPLAN Notices, 2010, 45(5): 1 - 2.

[4] Liao X K, Xiao LQ, Yang C Q, et al. Milkyway-2 supercomputer: system and application [J]. Frontiers of Computer Science, 2014, 8 (3): 345 - 356.

[5] Yang X J, Liao X K, Lu K, et al. The Tianhe-1a supercomputer: its hardware and software [J]. Journal of Computer Science and Technology, 2011, 26(3): 344 - 351.

[6] Liu Y Q, Yang C, Liu F F, et al. 623 tflop/s HPCG run on Tianhe-2: leveraging millions of hybrid cores [J]. The International Journal of High Performance Computing Applications, 2016, 30(1): 39 - 54.

[7] Wang Y X, Zhang L L, Liu W, et al. Performance optimizations for scalable CFD applications on hybrid CPU + MIC heterogeneous computing system with millions of cores[J]. Computers & Fluids, 2018, 173: 226 - 236.

[8] Chen J. Towards exascale direct numerical simulations of turbulent combustion on heterogeneous machines[R]. ISC High Performance Computing Conference, Frankfurt, Germany, 2016.

[9] Gorobets A, Soukov S, Bogdanov P. Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers [J]. Computers & Fluids, 2018, 173: 171 - 177.

[10] Álvarez X, Gorobets A, Trias F X, et al. HPC^2—A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD [J]. Computers & Fluids, 2018, 173: 285 - 292.

[11] Cai Y, Ao Y L, Yang C, et al. Extreme-scale high-order WENO simulations of 3-D detonation wave with 10 million cores [J]. ACM Transactions on Architecture and Code Optimization, 2018, 15(2): 1 - 21.

[12] Svärd M, Carpenter M H, Nordström J. A stable high-order finite difference scheme for the compressible Navier-Stokes equations, far-field boundary conditions [J]. Journal of Computational Physics, 2007, 225(1): 1020 - 1038.

[13] Pulliam T H, Zingg D W. Fundamental algorithms in computational fluid dynamics [M]. New Delhi: Springer International Publishing, 2014.

[14] 王铜铜, 杨梅芳, 董仕, 等. 可压缩 N - S 方程求解软件优化及大规模异构并行计算 [C]. 全国高性能计算学术年会, PAC2016 全国并行应用挑战赛优秀论文, 2016.
WANG Tongtong, YANG Meifang, DONG Shi, et al. Optimization and large-scale parallel computing of compressive N - S equation solver[C]. CCF National Annual Conference on High Performance Computing, Parallel Application Challenge 2016 Excellent, 2016. (in Chinese)