计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(6): 1455−1468, 2025

SW39000 处理器上顺序任务流多从核辅助并行构图算法

傅 游[1] 贾淑慧[1] 陈 莉[2] 花 嵘[1] 杜云龙[1] 高希然[2,3]
[1]（山东科技大学计算机科学与工程学院 山东青岛 266590）
[2]（处理器芯片全国重点实验室（中国科学院计算技术研究所） 北京 100190）
[3]（中国科学院大学 北京 100190）
（fuyou@sdust.edu.cn）

Multi-Slave Core Assisted Parallel Composition Algorithm for Sequential Task Flows on the SW39000 Processor

Fu You[1], Jia Shuhui[1], Chen Li[2], Hua Rong[1], Du Yunlong[1], and Gao Xiran[2,3]
[1]（*College of Computer Science and Engineering*, *Shandong University of Science and Technology*, *Qingdao*, *Shandong* 266590）
[2]（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）
[3]（*University of Chinese Academy of Sciences*, *Beijing* 100190）

Abstract Sequential task flow (STF) represents the access to shared data as dependencies between tasks. The STF runtime system achieves asynchronous parallelism through task construction, dependency analysis, and task dependence graph (TDG) generation, as well as task scheduling. The overhead of these three steps directly affects the performance of parallel programs. The current AceMesh runtime system, with STF at its core, employs a single master core architecture and multiple slave cores for execution on SW39000 processor. However, the discrete memory access performance of SW39000 processor is weak, and the composition of fine-grained tasks increases the discrete memory access, making the composition more likely to become a bottleneck. In this regard, we propose an algorithm that uses multiple auxiliary cores to assist the main core in patterning. First, we analyze the parallelism in the dependency analysis and TDG generation process, and implement a multi-core assisted parallel graph construction algorithm, parallelized fatTDG building algorithm with helpers (PFBH), based on the fat task dependency graph fatTDG on SW39000 processor, and optimize it. Secondly, in response to the problem of main memory resource contention among threads, a method for adjusting the resources of the subordinate cores and parameter selection during parallel graph construction and execution is proposed. Finally, experiments are conducted under five typical applications: compared with a single-core serial graph construction system, the acceleration ratio could achieve up to 1.75 times in fine-grained task scenarios; compared with the OpenACC model on SW39000 processor, AceMesh could achieve up to 2 times acceleration.

Key words sequential task flow; heterogeneous many-core parallelism; task scheduling; parallel graph construction; multi-slave core collaboration; SW39000

摘 要 顺序任务流（sequential task flow, STF）将对共享数据的访问表示为任务之间的依赖关系, STF 运行时系统通过任务构造、依赖分析和任务依赖图（task dependence graph, TDG）生成、任务调度实现异步

收稿日期: 2025-03-01; **修回日期**: 2025-04-24
基金项目: 山东省自然科学基金项目(ZR2022MF274, ZR2021LZH004)
This work was supported by the Natural Science Foundation of Shandong Province (ZR2022MF274, ZR2021LZH004).
通信作者: 高希然(gxran1993@163.com)

并行,这3个环节的开销直接影响并行程序的性能.目前以STF为核心的AceMesh运行时系统,在SW39000处理器上仅使用单主核构图、多从核执行的方式.然而,SW39000处理器离散访存性能较弱,细粒度任务构图离散访存增多,构图更容易成为瓶颈.对此,提出了一种利用多从核辅助主核进行构图的算法.首先,分析在依赖分析和TDG生成过程中的并行性,在SW39000处理器上实现了一种基于胖任务依赖图(fatTDG)的多核辅助并行构图算法PFBH(parallelized fatTDG building algorithm with helpers)并进行优化.其次,针对线程间的主存资源竞争问题,提出构图与执行并行中从核资源调节方法及参数选择.最终,在5类典型应用下进行实验测试.与单核串行构图系统相比,在细粒度任务场景下最高加速为1.75倍;与SW39000处理器上的OpenACC模型相比,AceMesh最高可达2倍加速.

关键词 顺序任务流;异构众核并行;任务调度;并行构图;多从核协同;SW39000

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202550166 **CSTR:** 32373.14.issn1000-1239.202550166

顺序任务流(sequential task flow, STF)编程模型通过分析任务的数据流访问信息,自动发现任务之间的依赖关系进而构建任务依赖图(task dependence graph, TDG),并通过底层编程系统自动实现任务调度.与具有大量全局同步的fork-join模型相比,STF编程模型能发掘更多细粒度及动态不规则应用的并行性,提供了更多异步并行机会.采用STF编程模型能够降低用户在架构复杂性上的认知负荷.STF的通用性已在科学应用中得到了验证[1-3],并在高性能计算(HPC)领域的应用日益广泛.

StarPU[4], OpenMP4.0[5], OmpSs[6], AceMesh[7]等模型的运行时系统设计皆以STF编程模型为核心.这些模型的运行时系统负责确保依靠任务数据需求来定义的任务执行顺序的正确性.STF编程模型的运行时开销[8]主要在构图和调度2个阶段.构图阶段包含任务构造、依赖分析和TDG生成;调度阶段则主要负责将就绪任务调度到空闲的线程上执行.为了保证任务执行的顺序一致性,运行时系统一般采用单核构图、多核执行的方式,即由一个线程负责构图并将生成的就绪任务分发到多个线程并行执行.

“神威·海洋之光”使用的异构众核处理器SW-39000是我国自主研制的超级计算机.由于内存带宽的限制,SW39000多核处理器需要在强扩展模式下运行,而实现强扩展的前提是能够提取更细粒度的并行性,以保持所有处理核心繁忙.但考虑任务粒度减小给运行时系统带来大量的同步和调度开销,任务粒度也不能无限制的减小,Slaughter等人[9]对最低有效任务粒度与性能损失之间的关系进行了量化,即如果允许应用性能下降50%,在x86多核系统上,STF最小支持的任务粒度为0.1 ms.

然而,AceMesh运行时系统目前在SW39000处理器上采用单主核构图、多从核执行、边构图边执行的方式[10],该方式会产生大量的主核离散访存,且构图与执行之间存在主存带宽竞争,细粒度构图性能受限.我们使用文献[9]提出的量化方法进行测试,结果显示应用性能下降50%的情况下,STF最小支持的任务粒度达到了0.3 ms,需要进行优化.

为兼顾细粒度可扩展和AceMesh运行时系统的高效性,以充分利用SW39000上的众核资源完成高效计算,本文首先提出多核并行构图算法,根据数据分布对依赖分析阶段进行并行化,之后进一步对开销占比较大的任务构造阶段进行并行.为了进一步隐藏构图的开销,实现构图与执行的并行,采用实时收集顶层就绪任务的方法,将任务提前派升至就绪队列中.其次,提出构图与执行并行时的资源调度方法,以最大程度减小内存带宽对辅助构图线程和任务执行线程的影响.通过在4个典型案例下对辅助构图线程的极限并行度进行测试,并评估任务执行线程对辅助构图线程的干扰情况,来决定具体调优参数.之后,将上述技术在AceMesh运行时系统中实现.最后,在SW39000处理器上基于5个典型案例进行实验评估和分析.

针对SW39000处理器的离散访存性能弱及主存带宽与计算能力不匹配的缺点,本文主要提出3个方面的贡献:

1)在维护STF编程模型的顺序一致性的同时,探究依赖分析和任务构造阶段的并行性,实现构图和执行的并行.在SW39000处理器上实现一种基于胖任务依赖图的多核辅助并行构图算法并进行优化,将离散访存压力分散到多个从核,解决细粒度任务场景下TDG构建瓶颈.

2)针对多从核辅助并行构图算法的构图开销无

法隐藏的问题，提出就绪任务的实时收集方法，提前派生就绪任务以实现并发构图. 针对并发构图时因主从核带宽竞争导致的构图效率低下问题，提出构图与执行之间资源调度方法并给出具体调节参数选择方法.

3）基于多从核辅助并行构图的思想，将上述 2 种优化方案在 AceMesh 数据流运行时系统中实现，并进行了实验评估和分析.

1 相关工作

在本节中，主要介绍 STF 编程模型研究现状与 AceMesh 运行时系统、SW39000 处理器架构，并分析在 SW39000 上构建高效 STF 运行时系统面临的挑战.

1.1 STF 编程模型研究现状

STF 编程模型的运行时开销来源于构图和调度 2 个方面，因此已有的运行时系统的构图优化研究主要集中在任务构造与依赖分析 2 个阶段.

在构图阶段的研究中，Maronas 等人[11]提出了协同工作任务的概念，通过将任务分解为多个细粒度的块，并利用工作共享技术在多个线程之间分配这些块，让单核能够执行多个任务，实现了资源的高效利用. Podobas 等人[12]重新设计了哈希表，将其组织为一种具有部分并发访问的按位字典树，以加速细粒度任务构图下的依赖分析阶段；利用模式保存技术[13]重用之前已解决的依赖模式，显著提高了依赖分析的速度，在 Podobas 等人的设计下单个任务内仍是串行构图. Bosch 等人[14]提出一种基于分布式运行时管理器的异步运行时结构设计，利用空闲线程来修改运行时结构，虽然具有良好的可扩展性，但其 TDG 构建过程仍然是串行的，而本文的 PFBH 算法通过多核协同实现 TDG 构建的并行化，显著降低了细粒度任务场景下的构图开销. Perez 等人[15]提出的弱依赖概念和 Elshazly 等人[16]提出的提前释放依赖项策略，可支持任务嵌套及嵌套图的跨依赖域调度，其中每个父任务作为一个独立依赖域构成一个子图，多个独立依赖域的子图可以并行构建，但是单个依赖域内部仍为串行构图，而 PFBH 算法的思想依旧可在嵌套图中使用，实现对单个依赖域的 TDG 构建并行化. 也有研究通过软硬件协同来加速运行时系统的构图过程，如 Tan 等人[17]将运行时系统转移到特定的硬件中，使用硬件来管理任务之间的依赖关系，PFBH 算法完全基于软件来优化.

在任务调度方面的研究主要集中在任务调度机制、就绪队列的组织形式、负载均衡优化 3 个方面. 在任务调度机制上，Perez 等人[15]提出弱依赖概念，进行更细粒度的依赖关系释放. 在就绪队列的组织形式方面，Álvarez 等人[18]针对任务细粒度场景下构图线程在锁竞争中较为弱势的问题，设计基于动态代理锁的高可扩展的集中式调度器. 在负载均衡方面，Agullo 等人[19]针对分布式存储架构特点，为 STF 运行时系统增加了一个节间数据管理层，通过自动通信来支持运行时的任务调度.

相较于以往研究，本文创新性提出 PFBH 算法，实现多核协同的单个任务内构图的并行化，降低细粒度任务构图开销. 本文的并行构图算法完全基于软件优化，具有更广泛的适用性. 此外，本文针对并发构图时主从核竞争带宽资源问题，提出从核资源调节方法并给出参数以保障负载均衡.

1.2 AceMesh 运行时系统

AceMesh[7]编程模型是一种 STF 制导语言，可以自动地将传统的遗产 MPI 应用代码转换为 TDG 的形式，该编程模型由源到源的翻译器和运行时系统组成，目前支持 x86 多核处理器组成的超算平台、神威海洋之光平台和天河 II 超算平台.

AceMesh 运行时系统主要包括构图和任务调度 2 个部分. 而构图又进一步分为任务构造、依赖分析（包括依赖分析和任务图生成）2 个阶段. 图 1 展示了在 AceMesh 运行时系统中，一个任务在其生命周期内被创建、调度的过程. AceMesh 运行时系统分为任务构造、依赖分析、任务调度 3 个组件. 计算线程（worker）是运行时系统对计算单元 CPE 的抽象.

1）任务构造. 任务构造组件负责收集每个任务

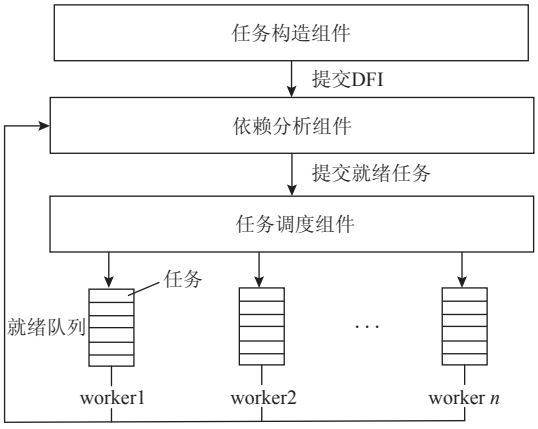


Fig. 1 Illustration of the composition and execution process of AceMesh runtime system

图 1 AceMesh 运行时系统组成及执行过程示意图

在注册阶段描述的数据流信息(data flow information, DFI),构造任务对象并进行任务参数赋值.任务构造通常采用可扩展的内存分配方式,其开销随着数据流的复杂性和任务数量的增加而增大.

由于数据访问区域的依赖分析开销很大,STF 系统[20-22]普遍采用区域首地址代替数据访问区域,因此 AceMesh 的 DFI 用〈任务编号,区域首地址,读写模式〉三元组表示,其中读写模式分为读、写、读后写 3 种.

2)依赖分析.根据任务构造过程收集的 DFI,由依赖管理器(dependence manager, depM)构建正在注册的任务与前驱任务之间的数据依赖边.depM 的核心数据结构通常是哈希表,用于保存数据流地址及其最近的读写信息.

国内研究在 AceMesh 运行时系统的构图阶段进行了深入改进与优化.王松等人[23]采用并发的数据结构实现了构图与执行的并发,通过隐藏构图时间获得了一定效果.孟现粉[24]在 SW26010 处理器上采用可分割的数据结构和分治策略研究了并行构图方法.傅游等人[10]利用 SW39000 处理器的架构特征提出基于代理的 AceMesh 运行时系统构图机制,实现了构图和执行的并发.

3)任务调度.任务调度器根据调度策略、任务之间的依赖关系和系统资源情况,负责将就绪任务调度到空闲的 worker 线程上去执行.异构系统中,任务调度需要考虑不同计算单元的性能差异和通信开销[25].傅游等人[10]利用 SW39000 处理器的架构特征优化调度中的通信开销,提出无锁的集中式任务调度器 LDM-C.

1.3 SW39000 处理器架构介绍

SW39000 处理器是 SW26010 的升级版,具有很高的计算密度,由其构建的“神威·海洋之光”超算系统可达到 1.5 ExaFlops(百亿亿次)的峰值计算能力.

SW39000 处理器的结构如图 2 所示,包括 6 个核组(core group, CG),每个 CG 包括 1 个主核(manage process element, MPE)作为运算控制核心和 1 个 8×8 从核阵列 CPE(computing process element).每个 CG 中都有 16 GB 主存,由 MPE 和 CPE 共享,主存分为可缓存空间和不可缓存空间.有 32 KB 和 512 KB 两级缓存.L1 缓存只保证垂直一致性,不支持与主核、其他从核的 L1 缓存之间的缓存一致性.每个 CPE 除了主存外还有片上的局部存储器,该局部存储器包含一个 32 KB 的 L1 指令缓存和一个 256 KB 的局部数据空间 LDM(local device memory).SW39000 的 LDM 空间比 SW26010 的大 4 倍,用户可以按照比例将其配置为软件管理的局部数据存储 LDM 或硬件管理的 L1 数据缓存(32 KB 或 128 KB).

SW39000 的从核之间可以通过远程内存访问(remote memory access, RMA)进行通信,实现核组内从核 LDM 之间的高效数据传输,支持单从核模式、行多播模式和列多播模式.从核访问主存主要有全局访问指令(ld/st)和直接内存访问(direct memory access, DMA)两种方式.ld/st 可以直接将数据从主存读取到寄存器,或者将数据由寄存器直接写回主存;DMA 实现从主存到 LDM 的快速传输,支持从核对主存数据进行批量访问.

1.4 SW39000 上构建高效 STF 运行时系统的挑战

SW39000 处理器的独特体系结构给 AceMesh 运

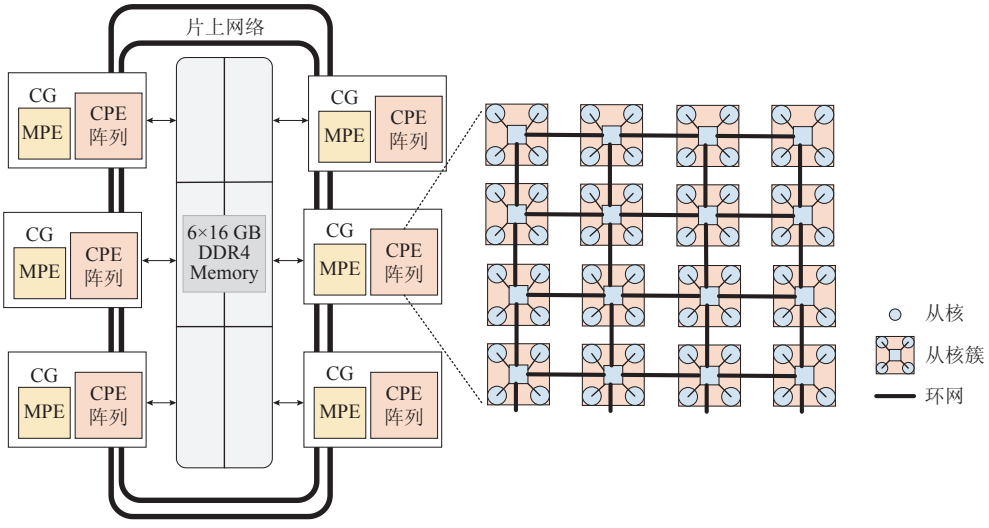


Fig. 2 Architecture diagram of SW39000 many-core chip

图 2 SW39000 众核芯片结构图

行时系统带来了2点挑战：

1）随着任务粒度的减小以及任务数增多，构图的绝对开销线性增加，细粒度场景下构图速度限制了性能扩展. 为了确保顺序一致性，AceMesh运行时系统由一个单独的名为creator的线程负责构图并将就绪任务顺序提交到worker执行，creator同时还负责任务构造和依赖分析，这种单线程的构图方法也称为集中式构图（centralized building, CB）. 显然，在任务粒度比较小或者核心数比较多的多核平台上，集中式构图很容易成为系统瓶颈.

2）SW39000处理器上离散访存性能较弱，比缓存一致性多核平台的构图开销大. 构图过程中频繁访问TDG和哈希表之类的动态不规则数据结构，使得构图开销对机器的访存延迟非常敏感. 而主存延迟和主存带宽是SW39000处理器的性能瓶颈，导致AceMesh运行时系统在该处理器上的集中式构图更容易因核间竞争成为瓶颈，破坏了AceMesh运行时系统的粒度扩展性.

2 基于胖任务图的多核辅助并行构图算法（PFBH）

本节提出PFBH算法，该算法主要包括3个过程：首先发掘依赖分析阶段的并行性，之后发掘任务构造阶段并行性，最后对构图和执行进行并行优化.

2.1 依赖分析阶段的并行性发掘

依赖关系的表达方式有2种：一种是传统的瘦任务依赖图（thinTDG），该方式不保留数据流访问信息，虽然这种表达方式较为简洁、所占内存少，但可能限制运行时系统的细粒度任务调度能力，且无法利用数据流访问信息进行依赖关系早释放或弱依赖处理等优化. 一种是fatTDG[26]方式，该方式在构图过程中保留大量的数据访问（data access, DA）记录和访问节点信息，会增加内存开销，但有利于对任务图进行更细粒度的调度，也提供更多的优化可能性. 与thinTDG不同的是，fatTDG的DFI以〈DA，区域首地址，读写模式〉三元组的形式来表示.

AceMesh运行时系统的顺序一致性语义要求同一个数据流地址的前后2次访问之间的依赖分析必须保持原有的时间顺序，否则TDG的构建可能会出错. 但不同数据流地址的数据依赖分析完全可以并行进行. AceMesh运行时系统引入对任务访问的数据流地址去重，因此，单个任务内与不同任务之间的数据流地址的依赖分析皆可并行进行，依赖分析阶段存在丰富的数据并行性.

本节在SW39000处理器上对构图过程中离散访存较多的共享数据结构depM进行均匀的线程划分，以加速构图. 基于fatTDG对依赖分析阶段进行并行，引入多个helper并行进行依赖分析，辅助creator构图实现并行构图. 总体思想为：设计映射函数确保DFI均匀分布，按照数据流地址将对应的DFI映射到不同的helper，再均匀映射到哈希表上，为提高通信效率，设计双缓冲消息队列来接收映射来的DFI.

1）映射函数的选择. 依赖分析中，使用共享数据结构哈希表作为依赖管理器，记录所有数据流地址的最新访问信息. 斐波那契哈希函数的核心在于利用斐波那契数列的特点来减少碰撞，使哈希值分布尽量均匀[27]. 因此，本文选择斐波那契哈希函数对DFI进行分发，确保均匀分布.

2）数据流地址向helper索引号及哈希表索引号的映射. 为了均匀分布，上述2个映射过程皆采用斐波那契额哈希函数. 数据流地址 *addr* 映射到helper索引号及哈希表索引号的过程，皆可用式（1）和式（2）进行展示.

$$Hash_Temp = (addr \times Hash_Value), \tag{1}$$

$$idx = (Hash_Temp \gg 44)\&Mask, \tag{2}$$

其中 *Hash_Value* 为斐波那契数列的值，*Hash_Temp* 为临时哈希值；≫44表示右移44位，以提取哈希值中变化较明显的高20位；*Mask* 表示掩码，是一个二进制数，其选择的依据与设计决策和哈希表的大小有关；*Hash_Temp* 与 *Mask* 按位进行与操作，得到从核号. 例如，对于helper索引号的小范围映射，当helper个数为8时，*idx* 需要位于0~7范围内，设置 *Mask* 为“0x7”（低3位全为1）；对于哈希表索引号的较大范围映射，若哈希表索引号大小设置为65 536，则 *Mask* 为“0xffff”（低16位全为1）.

3）双缓冲消息队列设计. 在并行构图过程中，creator作为上游对下游的helper进行构图消息的传递，消息队列的组织形式会影响上下游之间的消息传递效率. 本文将creator映射至主核，helper映射到从核. 由于缓存没有横向一致性，该消息队列只能映射到不可缓存空间. 由于DFI的信息比较大，只能通过主存传递，由creator写入主存后，再由helper以DMA的方式批量从主存读取. 主存上分配 N 套缓冲队列（N 为参与构图的helper个数），每套有2个缓冲区 buf_1 和 buf_2. 如图3所示，creator先向 buf_1 填充数据，在填充完 buf_1 后，利用主核与从核间的全局访问指

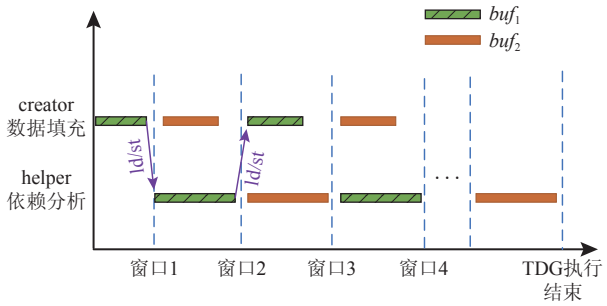


Fig. 3 Design diagram of double-buffered message queue

图 3 双缓冲消息队列设计图

令 ld/st 把 buf_1 的读写权限移交给 helper, 然后切换到 buf_2 进行数据填充. 而 helper 得到 buf_1 的读写权限后, 进入依赖分析阶段. 这种双缓冲设计, 使得 creator 填充 DFI 阶段的开销几乎被依赖分析阶段覆盖.

如图 4 所示, 经过对数据流地址映射, 每个 helper 仅需访问 depM 的一个子集, 且不同 helper 的访问区域互不重叠. depM 被划分为多个子集后, 每个 helper 拥有一个独立的子集作为其私有数据结构, helper 只访问其专属的 depM, 避免了数据竞争. 因其私有性, depM 可以存储在可缓存的主存空间, 也可以存储在 LDM 空间. 由于 depM 的尺寸大、访问模式不规则, 本文将其映射到可缓存的主存空间. 通过以上设计, 消除了构图中的并发访问的需求, 并保持了构图过程中的语义顺序一致性.

为了确保并行构图后任务执行顺序的正确性, 采用构图和执行交替执行方式, 并设置窗口同步边界, 即每当注册的任务个数达到 512 时, 就加入一个全局同步点. 在此之后, creator 扫描所有已经注册的任务, 得到就绪任务 topTasks, 并将其派生到就绪队列当中. 之后, 将 helper 切换到 worker 模式, 执行计算任务, 并等待该窗口任务执行完毕, 再次进行下一轮的构图.

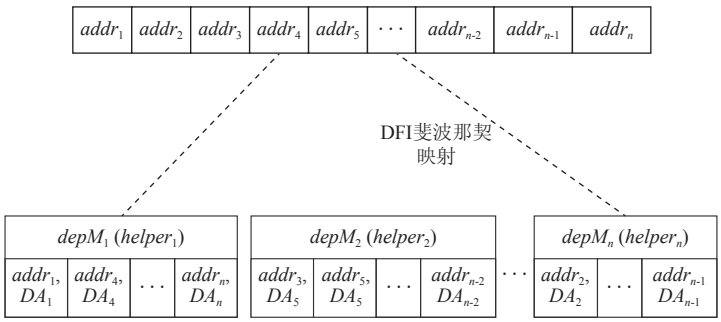


Fig. 4 Schematic diagram of data flow address mapping

图 4 数据流地址映射示意图

上述过程发掘了构图过程中依赖分析阶段的并行性, 将单主核构图时的离散访存压力分散到多个 helper 上以加速构图. 考虑到运行时任务构造阶段的开销不可忽视, 下面发掘任务构造阶段的并行性.

2.2 任务构造阶段的并行实现

任务构造阶段主要包括数据流地址的顺序收集、任务对象的构造(对任务进行存储空间分配与初始化操作)、DA 的构造(对 DA 进行存储空间分配、初始化、数据填充). 图 5 展示了构图阶段的各项开销占比.

本文通过调整案例循环分块的大小来改变 AceMesh 运行时系统的任务粒度(granularity). 为了探索构图优化的潜力, 本文主要针对集中式并发构图时总时间远大于任务执行时间的场景进行研究. 此外, 由于 SW39000 处理器的 LDM 空间大小受限, 本文对应用案例的分块普遍较小, 故统一归入细粒度. 为了方便展示, 使用全部任务的指令总数对每个任务的粒度进行归一化处理, 并在后续实验的测试图表中, 将指令数转换为以 2 为底的对数值.

图 5 显示未引入依赖分析并行前, 任务构造阶段的时间开销占比约为 55%, 随着依赖分析并行的引入(helper 个数为 8), 该阶段时间开销占比升高到 70% 左右. 任务构造阶段主要的时间开销是 DA 的构造, 引入依赖分析并行后约占构图时间开销的 50%.

数据流地址的顺序收集因其数据依赖性和顺序性不能被并行, 同样, 单个任务对象的构造由于内存访问冲突也不能被并行.

由于依赖分析并行时已经将 DFI 以〈DA, 区域首地址, 读写模式〉三元组的形式, 通过数据流地址均匀映射到各个 helper 上. 因此, 相关 DA 的初始化和信息填充也可以通过数据流地址, 被分配到对应的

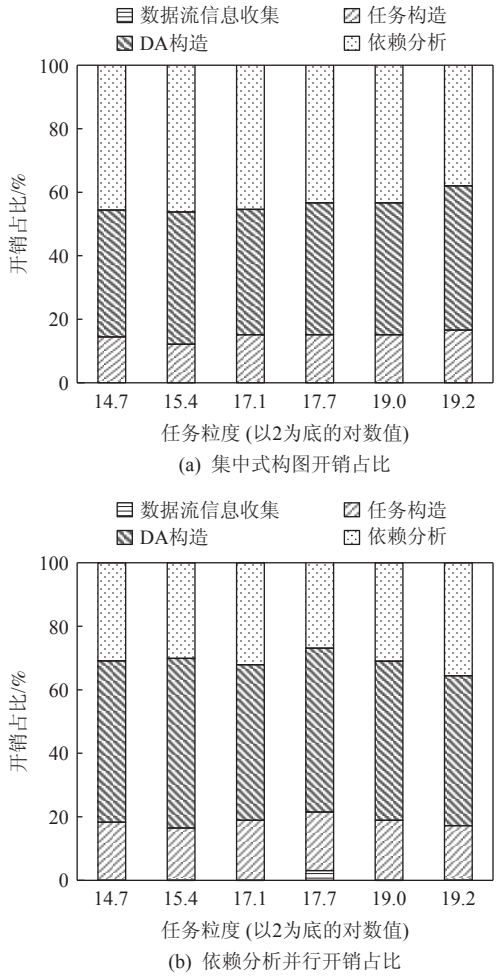


Fig. 5 Overhead proportions in TDG building

图 5 构造 TDG 开销占比

helper 上，从而可以借助多线程的并行访存能力，将任务构造阶段中 DA 的存储空间分配、初始化、数据填充操作在各个 helper 上完成. 由于需要在 helper 中将 DA 与对应任务进行关联操作，本节将 DFI 三元组的形式改成四元组〈DA，任务编号，区域首地址，读写模式〉的形式，由 creator 通过消息队列传送给 DA.

算法 1 和算法 2 分别展示 fatTDG 图结构下，creator 和 helper 参与并行构图的伪代码. 为了确保 STF 的顺序一致性，creator 负责启动 helper 和进行任务构造，按照串行顺序将 DFI 映射到各个 helper，同步窗口的任务图以及收集所有无前驱的任务 topTasks 和无后继的任务 bottomTasks. helper 则负责接收 creator 提供的 DFI，进行依赖分析并给 creator 发送同步消息，确保顺序一致性的同时发现构图中的并行性.

算法 1. creator 参与构图的伪代码.

① start_helpers; /*启动 helpers*/
② for (each task in window) /*窗口内任务注册*/
③ collect_addr (&addr_list); /*数据流地址收集*/
④ init_task(&task, args); /*任务的构造*/
⑤ for (each DFI) /*遍历 DFI*/
⑥ map_addr (addr, & addr_Dest); /**addr* 映射*/
⑦ send_DFI (DFI, DA_Dest); /* DFI 传输*/
⑧ end for
⑨ end for
⑩ bcast_and_finalize (); /*同步操作*/
⑪ start_exe_workers; /*启动从核参与计算*/
⑫ spawn_topTasks; /*派生就绪任务*/
⑬ build_join_tasks; /*构造 TDG 出口*/
⑭ wait_TDG (); /*等待 TDG 执行结束*/
⑮ return.
/*helper 参与构图的伪代码*/
⑯ while (1) /*轮询*/
⑰ recv_msg (&my_type); /*接收消息类型*/
⑱ if (my_type==finalize)break; /*终止构图消息*/
⑲ else if (my_type==df) /*进行构图消息*/
⑳ dfList=get_msg_data (); /*获取 DFI*/
㉑ for (each DFI in dfList) /*遍历 DFI*/
㉒ init_DFI (DA, task, *addr*, mode); /*DA 的构造*/
㉓ dep_analysis (DFI, &it, &eList); /*查找哈希表*/
㉔ update_depM (it, DFI); /*更新 depM*/
㉕ add_edge (e); /*添加依赖边*/
㉖ end for
㉗ end if
㉘ end while
㉙ finalize_building; /*消息同步*/
㉚ return.

2.3 构图和执行的并行优化

2.2 节介绍了采用构图和执行的交替模式实现 PFBH 算法，这种交替模式具有 2 个缺点：1）在构图时浪费了大量的计算核；2）构图时间不能被较好地隐藏，在构图时间占比较小时性能可能不如集中式构图. 因此，本节实现算法的构图和执行并行.

任务依赖图构造时，需要 2 类信息：1）topTasks，即就绪任务，它们需要被派生进就绪队列. 2）bottomTasks，帮助实现任务图同步和判断任务图完成情况. 当窗口边界采用扫描法进行收集时，会导致 topTasks 在窗口构图结束后才能被收集，从而造成 topTasks 收集滞后. 本节实现 topTasks 的定期收集，并将识别 topTasks 的时机提前.

常用的 topTasks 定期收集方法有 2 个. 一个是数

据流计数法. 该方法在依赖分析时判断该数据流是否为最后 1 个依赖边, 此方法需要 creator 使用计数器记录 DA 总数, 并且每个 helper 进行 DA 处理时将计数器加 1, 并判断 DA 是否为任务的最后 1 个 DA, 并在分析完任务的最后 1 个 DA 后判定该任务是否为 topTask, 这给本来完全并行的 DA 处理引入了大量的原子同步, 降低了数据并行性, 开销较大. 另一个方法是异步识别+定期收集的方法, 让 helper 从局部视角找到前驱边数为 0 的任务, creator 定期收集这些信息, 把所有 helper 都认可的前驱边数为 0 的任务标记为真正的 topTasks, 并将识别 topTasks 的时机提前至窗口内. 对比第 1 种方法, 第 2 种方法中每个 DA-没有额外开销, 只在切换任务时才去做一些处理, creator 也是定期收集任务. 因此, 本文选用效率更高的第 2 种方法收集任务, 具体收集过程如图 6 所示, 主要可以分为 3 个阶段.

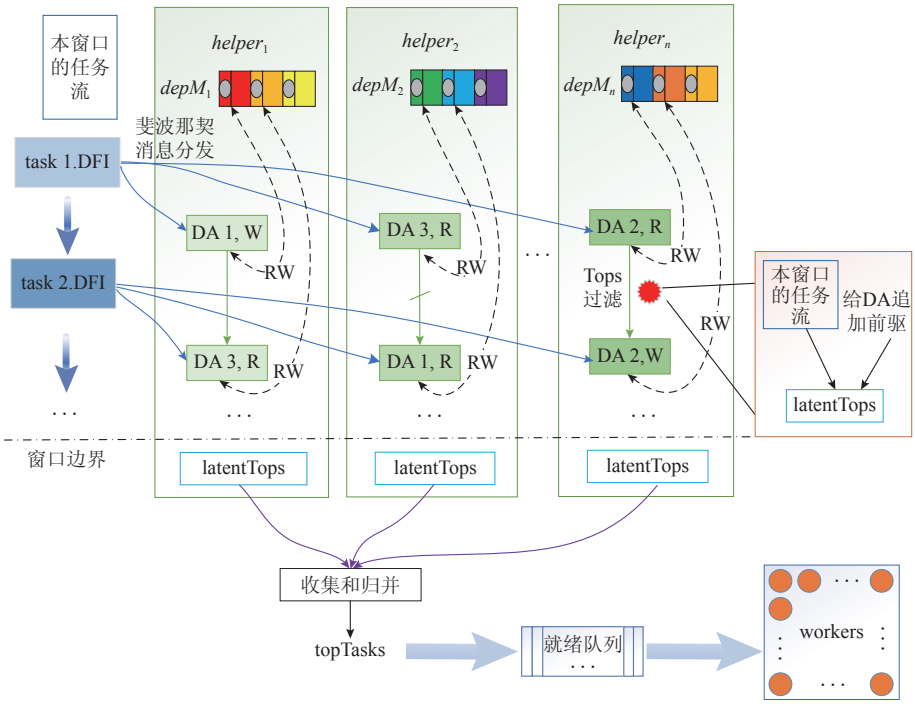


Fig. 6 Asynchronous identification and scheduled collection process of topTasks

图 6 topTasks 异步识别和定期收集过程

1)任务流收集及数据流消息分发. creator 负责收集本窗口的任务流, 并将 DFI 发给各个 helper.

2)临时就绪任务识别. helper 处理来自 creator 的 DFI 进行依赖分析, 每个 helper 维护一个临时就绪任务列表(latentTops), 收集暂时没有前驱的任务.

3)就绪任务列表归并. 每当到达窗口边界时, creator 对各个 helper 收集的临时 latentTops 进行归并, 求出真正的 topTasks, 并将其派生到就绪队列中, 调度到 workers 上去执行.

在定期收集 topTasks 后, 需要在窗口边界进行高效的任务图结束判断. 本文在窗口边界构造 endTask 任务作为该窗口的任务图出口, 并将 endTask 的 DA(简称 endDA)在窗口边界以消息的形式发给各个 helper, 由 helper 负责 endDA 与哈希表中无后继的 DA 建边. 当 endTask 就绪时, 代表任务图执行结束.

3 构图与执行并行中从核资源调节方法及参数选择

3.1 构图与执行之间的从核资源调节方法

构图与执行并行的场景下, helper 构图与 worker 执行任务会产生主存资源竞争, 需要确定从核资源 helper 的数量及 worker 的数量的分配. helper 作为生产者且数量更少, 因此, 需要减少 worker 对 helper 的干扰, 保护 helper.

减少 worker 对 helper 干扰的方法有 2 种: 一是调节 helper 个数(n_{helper}), 二是调节 worker 个数(n_{worker}). PFBH 算法中 depM 为 helper 私有的数据结构, 因此调节 helper 需要对 depM 进行重组, 很难且开销很大. 所以, 本文选择固定 n_{helper}, 通过调整 n_{worker} 来提高或

降低 helper 的优先级. 通过减少 n_{worker} 来减少 worker 对主存带宽的使用, 进而提高 helper 优先级的方法; 相反, 增加 n_{worker} 则是降低 helper 优先级.

当选用集中式调度器 LDM-C 时, 集中式队列的长度可以被看作是上游生产率的评估指标. 通过将集中式队列长度与 n_{worker} 建立关系, 并结合实验给出 2 个阈值 L_{hungry} 和 L_{busy}. 当队列中的任务数量低于 L_{hungry} 时, 表明上游的产出不足, 需要给 helper 提供充足的内存带宽. 相反, 如果队列中的任务数量超过了 L_{busy}, 则表示队列中的任务个数足以保证 CPE 阵列始终处于忙碌状态, 构图 helper 的优先级降低.

从核资源的调节方法如下:

1)固定 n_{helper}, n_{worker} 存在最大值 n_{wmax} 与最小值 n_{wmin}. n_{worker} 的初始值设置为 n_{wmin}.

2)定期检查就绪队列的长度 L_q, 并依据 L_q 与 L_{busy} 的大小调整 n_{worker}. 若 $L_q \geqslant L_{busy}$, 将 n_{worker} 值调整为 n_{wmax}; 若 $L_q < L_{hungry}$, 将 n_{worker} 值调整为 n_{wmin}.

3.2 从核资源调节的参数选择

基于 3.1 节描述的构图与执行并行场景下从核资源的调节方法, 本节结合 SW39000 的硬件特性说明如何确定 n_{helper}, n_{worker}, L_{hungry}, L_{busy} 的具体值.

1)n_{helper} 的确定

在 SW39000 处理器上, 主核密集不规则的离散访存是延迟敏感的, 从核的离散访存操作会对主核的访存延迟产生负面影响. 因此, 为保证主从核协同加速最大, 需要适度削减从核的参与.

当从核访存数据规模超过 L2 缓存(512 KB)时, 主核访存延迟受从核干扰明显. 由于单个从核的离散访存操作主要包括对哈希表的访存, 因此 n_{helper} 的大小为:

$$\frac{MC_{\text{L2 cache}}}{MC_{\text{depM}}} \leqslant n_{\text{helper}} \leqslant \frac{MC_{\text{L2 cache}}}{MC_{\text{depM}} \times Data_{\text{ratio}}}, \tag{3}$$

其中 $MC_{\text{L2 cache}}$ 代表 L2 缓存大小(512 KB); MC_{depM} 代表单个从核的哈希表大小(128 KB); $Data_{\text{ratio}}$ 表示在依赖分析时, 每个从核实际访存的数据量占哈希表大小的比例, 一般小于哈希表大小($Data_{\text{ratio}} < 1$). 受案例特点的影响, 不同案例的 $Data_{\text{ratio}}$ 也较为不同, 通过实验测试, 取各案例的平均 $Data_{\text{ratio}}$ =0.5, 相应的, 通过计算选取 n_{helper}=8.

2)n_{worker} 的确定

n_{worker} 的选择需要平衡构图和执行之间的主存资源竞争, 取值区间为 $n_{work} \in [n_{wmin}, n_{wmax}]$. n_{wmin} 代表 worker 在 wmin 个数下, 仍能够保证 helper 在构图压力较大时获得足够的主存带宽, n_{wmax} 则是为了构图压力较小时, 确保从核资源被充分用于任务计算, 二者计算公式为:

$$B_{\text{helper}} = \frac{B_{\text{total}} \times B_{\text{helper_ideal}}}{B_{\text{helper_ideal}} + n_{\text{wmin}} \times B_{\text{worker_ideal}}}, \tag{4}$$

$$1 - \frac{B_{\text{helper}}}{B_{\text{helper}_{\text{ideal}}}} = 50\%, \tag{5}$$

$$n_{\text{wmax}} = 64 - (n_{\text{helper}} + 1), \tag{6}$$

其中, B_{helper} 为 helper 在与其他 worker 共享主存带宽时实际能够使用的带宽; B_{total} 为可用的主存带宽总量; $B_{\text{helper_ideal}}$ 为没有其他 worker 竞争带宽时, helper 能够使用的最大带宽; $B_{\text{worker_ideal}}$ 为每个 worker 理想情况下的主存带宽. 式(5)中的“50%”表示随着 n_{worker} 增大, 允许 helper 的构图性能下降 50%; 式(6)中的“1”表示作为集中式调度器的 1 个线程.

通过实验测得细粒度下的 $B_{\text{helper_ideal}}$ 约为 43 MB/s, $B_{\text{worker_ideal}}$ 约为 430 MB/s, B_{total} 约为 3.3 GB/s, 计算得到 n_{wmin}=16; n_{wmax} 则设为剩余可用从核数的最大值, 即 64−9(8 个 helper 加 1 个集中式调度器)=55.

3)L_{hungry}, L_{busy} 的确定

L_{hungry}, L_{busy} 的设置需要在不同负载时, 在响应速度和资源利用率之间取得平衡. 实验逐步调整 L_{hungry} 和 L_{busy}, 为方便表示, 记 L_{busy}= $X \times n_{worker}$, L_{hungry}=$Y \times n_{worker}$ (X, Y 为正整数值). 其中 X 和 Y 分别表示 L_{busy} 和 L_{hungry} 相对于 n_{worker} 的倍数关系, 实验结果如图 7 所示.

图 7 显示当 $Y \leqslant 5$ 时, 频繁调整 n_{worker} 导致产生额外开销; 当 $Y \geqslant 15$ 时, 系统响应上游产出不足的延迟增加, 导致构图效率下降; 当 X=25 时, 系统性能较为稳定, 能够有效利用从核资源. 因此, 设置 L_{hungry}=10×n_{worker}, L_{busy}=25×n_{worker}.

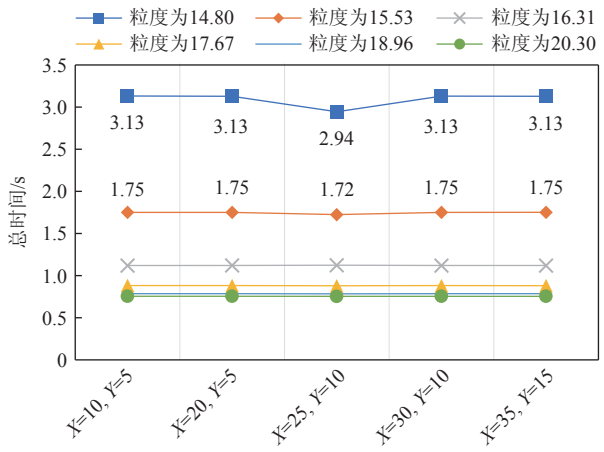


Fig. 7 Impact of values of L_{hungry} and L_{busy} on the total time under various granularities

图 7 不同粒度下 L_{hungry} 和 L_{busy} 的值对总时间的影响

3.3 buf 容量选择

因从核每次只接收 1 个缓冲区的数据, 为减少 2 个缓冲区之间的数据读写等待开销, 取 buf1 与 buf2 容量相同, 即有 $SIZE_{buf1}=SIZE_{buf2}$, 统一用 $SIZE_{buf}$ 表示. 根据神威使用手册, 在实际应用中, 当 DMA 一次传输的数据量是 1 024 B 的整数倍时, 能够实现高达 80% 的带宽使用效率, 即有:

$$SIZE_{buf}=N\times 1\,024\ \text{B}, \tag{7}$$

$$MC_{DFI}\times NUM_{DFI}\leqslant MC_{buf}. \tag{8}$$

下面确定 N 的值. LDM 除去分配的缓存空间(32 KB)和分配给其他变量的空间(32~150 KB)后, 剩余空间方可用作缓冲区. 在 LDM 空间中为缓冲区分配 32 KB 大小的空间, 记作 $MC_{buf}=32$ KB; NUM_{DFI} 表示缓冲区传输的 DFI 实例个数; MC_{DFI} 表示传输时单个 DFI 占用的空间大小, 其值为 32 B; 由式(7)计算可得 NUM_{DFI}=1 024, $SIZE_{buf}=32\ \text{B}\times 1\,024$, 故 N=32.

4 实验评估及结果分析

4.1 实验方法和测试用例

本节测试均使用表 1 展示的 5 个案例. 这些案例一般有 2 个参数: 一是矩阵维数 N_{md}, 决定了案例的问题规模; 二是程序 kernel 的迭代次数 N_{kl}.

Table 1 Test Cases and Their Corresponding Descriptions

表 1 测试案例及对应的描述

案例	描述
Multisaxpy	2 个变量的乘加运算: $Y=Y+\alpha\times X$
Cholesky	使用 4 种线性算子分解厄米特正定矩阵, 为计算密集型
SparseLU	对稀疏矩阵的 LU 进行分解
Gauss-Seidel	分块求解二维矩阵热方程
Stencil-3D7P	Jacobi 迭代测试程序, 采用 2D 分块划分任务

1) 实验环境

实验在 SW39000 处理器上进行, 测试案例使用的主核栈空间配置为 1 GB、内存空间大小配置为 13 GB, 从核数据缓存大小配置为 32 KB; 操作系统版本为 CentOS Linux 7; 编译工具链包括 GCC 4.8.5, CMake 3.23.0 和 Make 3.82.

2) 实验数据统计方法

本文对每个测试案例均进行 5 次测试, 取算术平均值作为不同配置下的最终结果. 在测试 OpenACC 版本时, 参考文献 [26] 的实验方法.

4.2 整体性能评估

本节将基于前述技术构造的多从核辅助并行构图的 AceMesh 运行时系统 AMS-PFBH(AceMesh's parallelized fatTDG building algorithm with helpers)与 AceMesh 运行时系统的主核集中式并发构图 AMS-CB (AceMesh's centralized building)进行对比. 两者采用相同的 LDM-C 调度器.

不同案例的输入规模如下: Cholesky, SparseLU 的 N_{md}=2 560, N_{kl}=100; Gauss-Seidel 的 N_{md}=3 840, N_{kl}=50; Multisaxpy, Stencil-3D7P 的 N_{md}=256, N_{kl}=100.

本节在细粒度场景下对 AMS-PFBH 与 AMS-CB 的性能进行对比, 评估其整体加速效果. 图 8 展示了 5 种典型案例的整体性能对比. 图 8 的数据表明, 所有案例都得到了正向的加速效果. 由于粒度在 20 左右时任务执行时间占比较大, 并行构图加速效果有限, AMS-PFBH 相对 AMS-CB 的整体性能加速效果有限, 且由于 AMS-PFBH 可以根据构图的压力自适应调节 worker 个数, 也没有带来负加速.

4.3 分项性能贡献

本节探讨各个优化技术对 AMS-PFBH 的贡献. 将主核构图交替执行作为初始版本, 然后增量地开启各个优化: 1)Anal-Parallel, 对依赖分析阶段进行并行, 交替构图; 2)Cons-Parallel, 对任务构造阶段进行并行, 交替构图; 3)BE-Parallel, 构图与执行并行; 4)BE-Sense, 构图与执行间的从核资源调节方法.

图 9 为各优化技术对性能提升的堆积图, 测试使用 4.2 节所采用的 5 个测试程序, 测试规模也与 4.2 节一致. 将打开全部优化之后的性能作为基准, 依次地打开各个优化, 将每项优化带来的性能提升占基准的百分比作为该优化的性能贡献.

图 9 显示对依赖分析阶段进行并行的交替构图, 在细粒度下的优化效果多数为 20%~40% 的性能提升; 对任务构造阶段进行并行的交替构图对性能的贡献效果较为显著, 多数测试案例占比在 30%~40%. 构图和执行并行的构图优化效果也很显著, 多数案例有 20%~30% 之间的性能提升, 构图与执行间从核资源调度方法占比最小, 多数案例的性能提升占比约在 10% 以内.

4.4 AMS-PFBH 与 OpenACC 的性能对比

本节在相同平台上进行实验, 并将最终优化结果与"神威·海洋之光"主流编程模型 OpenACC 的性能进行对比.

图 10 显示了使用效率(Efficiency)[28] 作为细粒度下的性能衡量指标. 由于 OpenACC 采用固定的任务粒度以及静态循环调度, 本文选择一个性能最优的尺寸. 在图 10 中 OpenACC 仅标注该该尺寸对应的任

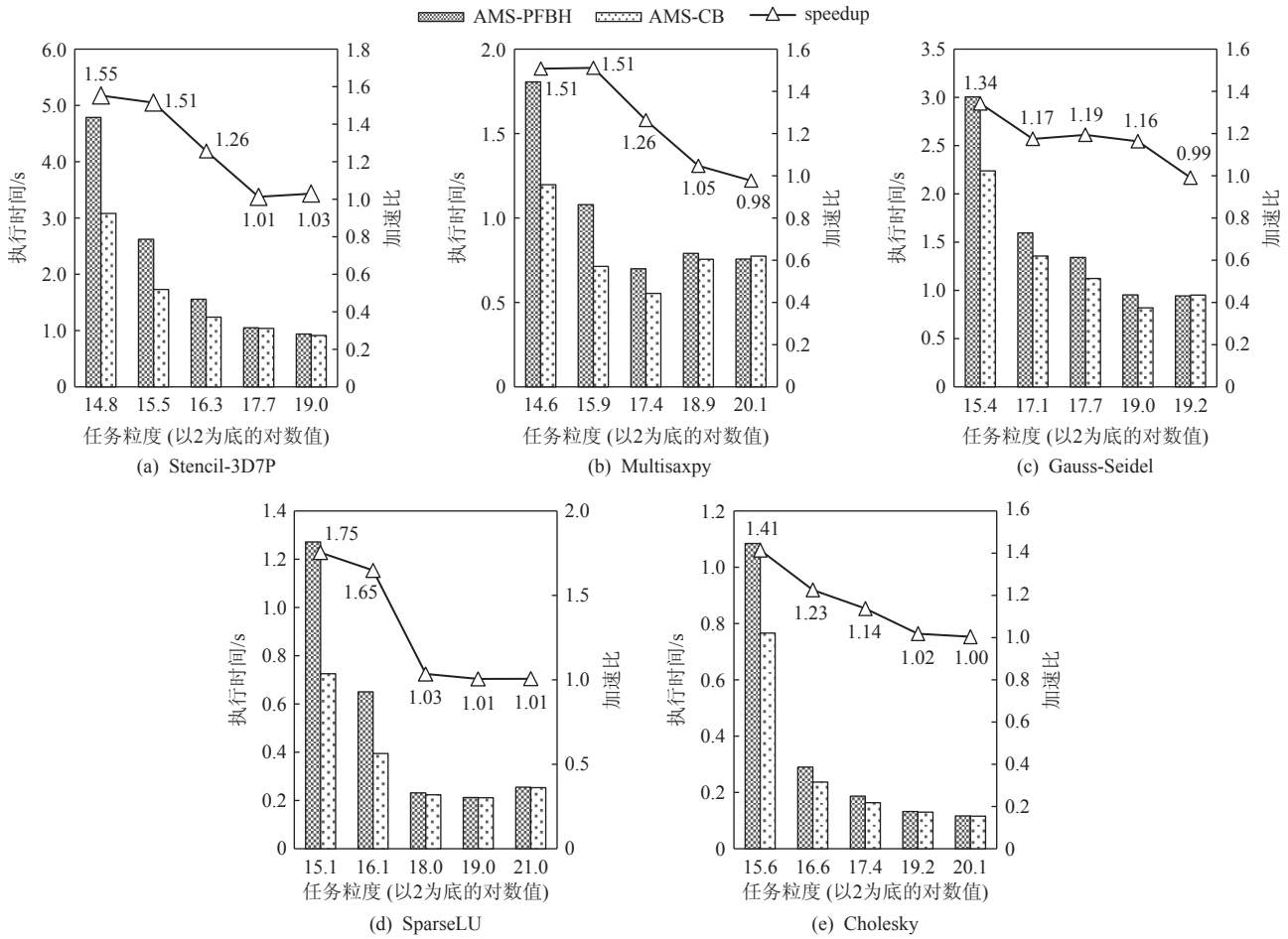


Fig. 8 Performance comparison of AMS-PFBH and AMS-CB building diagram

图 8 AMS-PFBH 和 AMS-CB 构图的性能对比

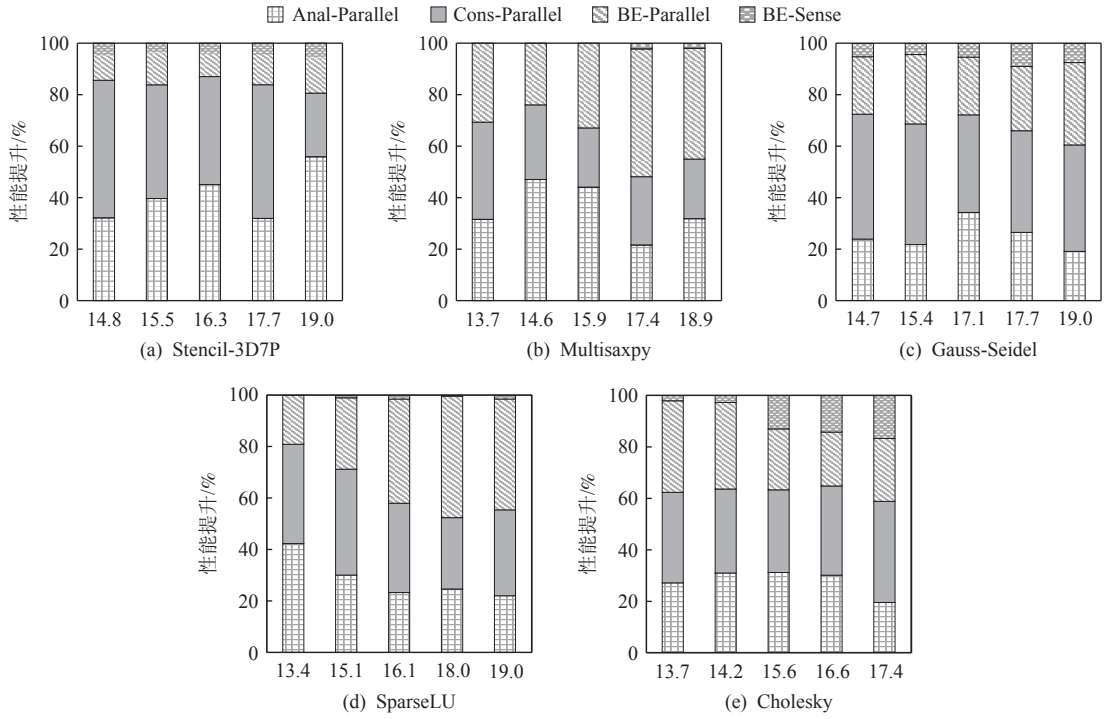


Fig. 9 Stacked diagram for performance improved by each optimization technology

图 9 各优化技术对性能提升的堆积图

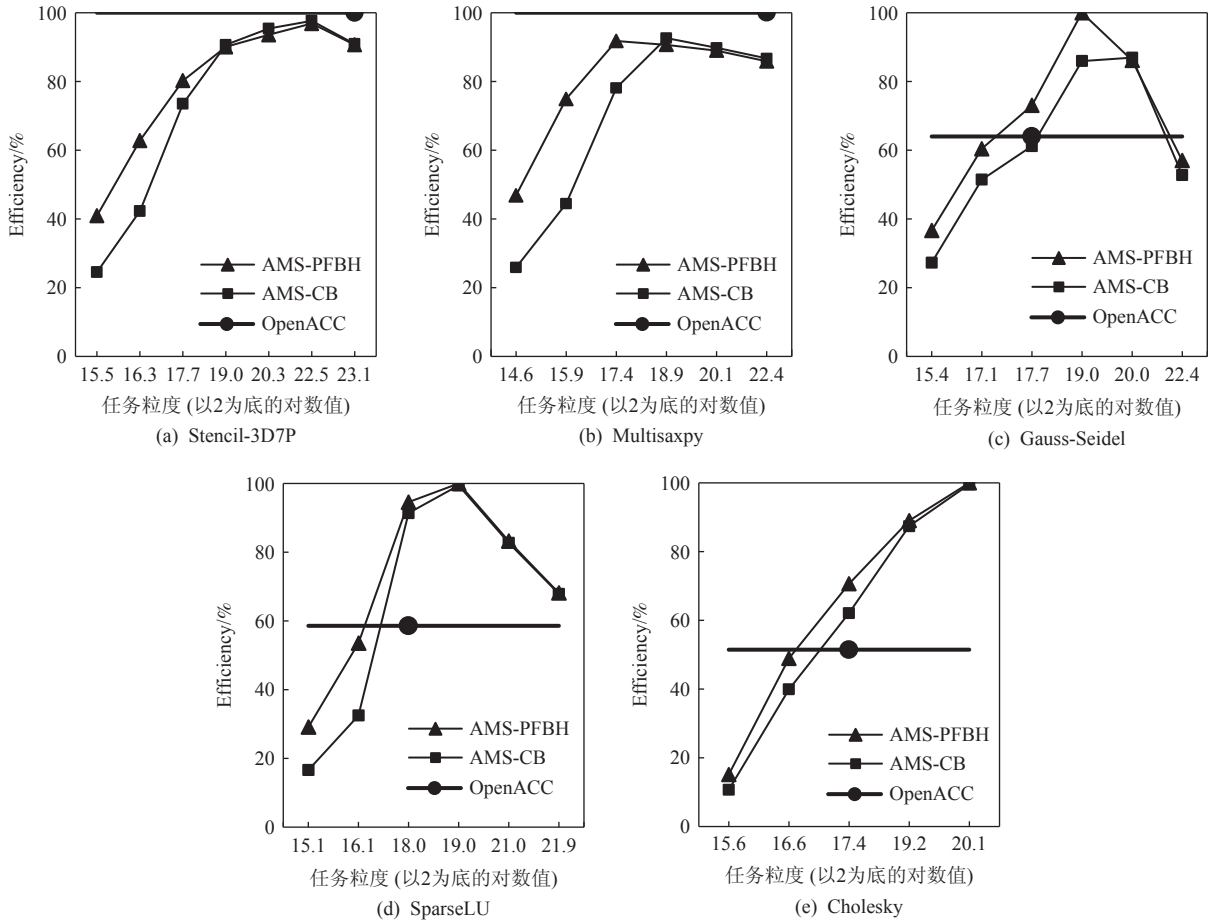


Fig. 10 Performance comparion of OpenACC and AceMesh

图 10 AceMesh 与 OpenACC 的性能对比

务粒度, 其他粒度上的性能不做标注. 为了方便对比, 我们将 OpenACC 该粒度下的 efficiency 数据向两端延长, 并在图 10 中用直线表示.

图 10 表明在细粒度任务下, AMS-PFBH 的性能优于 AMS-CB. AMS-PFBH 在一些应用上比 OpenACC 表现得更为出色. 例如, 对于循环间具有更多动态调度机会的 Gauss-Seidel, SparseLU, Cholesky, AceMesh 能够对任务进行异步并行及消除全区同步, 从而得到更高的加速, 最高加速比接近 2. 而对于 Stencil-3D7P 的数据访问模式规则, 每个计算点依赖于周围的几个点, 任务被划分为独立的块, 块之间的依赖关系较少, 更适合 OpenACC 的块同步模式. Multisaxpy 涉及多个向量的乘加操作, 每个向量的乘加可以独立执行, 也更适合 OpenACC 的块同步模式.

4.5 AMS-PFBH 扩展性评估

本节在启动多核组运行的模式下(即 SW39000 处理器的大共享模式), 以 AMS-CB 作为基准, 分别评估 AMS-PFBH 在 2 个核组、4 个核组和 6 个核组上的加速效果. 为了保证每个从核都获取到足够的任务数, 将各个案例的迭代次数 N_{kl} 分别比单核组增加 2 倍、4 倍和 6 倍, 实验结果如图 11 所示.

由图 11 可以看出, AMS-PFBH 在多核组下扩展性较好, 在 2, 4, 6 个核组下最高加速比分别为 1.69, 1.58, 1.52. 因此, 本文提出的 PFBH 算法不仅在单核组内能显著提升细粒度任务的构图效率, 还能在多核组扩展场景中维持高效的性能表现.

5 结 论

STF 编程模型通过将对共享数据的访问表示为任务之间的依赖关系, 实现了异步并行, 相比于 fork-join 模型, 能够更好地适应动态不规则的应用. 然而, 随着任务粒度变小, 集中式构图方式使 STF 编程模型的性能受限于构图过程的访存开销. 针对 SW39000 处理器的性能瓶颈, 本文基于 STF 制导的 AceMesh 运行时系统提出一系列优化措施, 以更好地利用众核资源. 首先, 在 SW39000 处理器上实现了 PFBH 算法, 对依赖分析与任务构造阶段进行并行, 并实现构

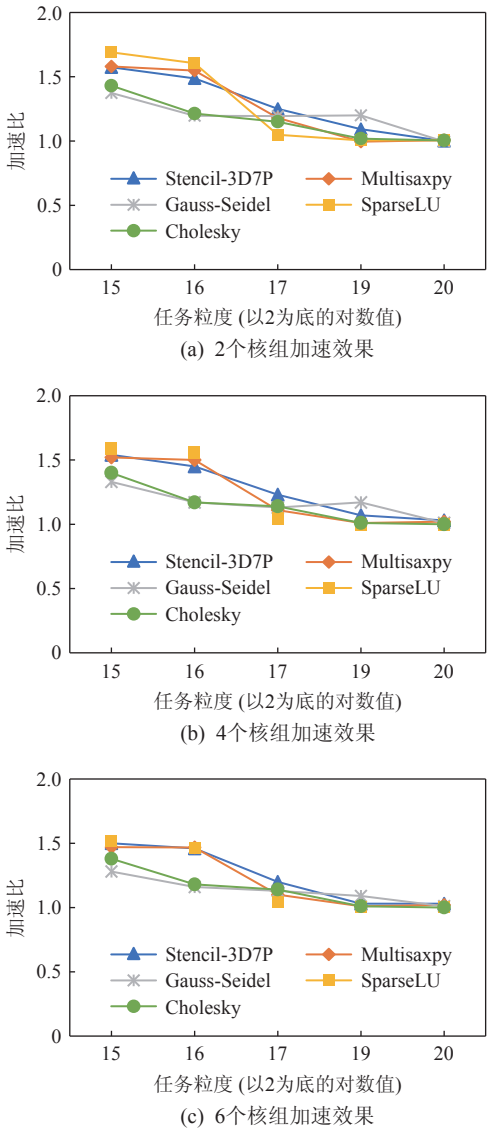


Fig. 11 Scalability assessment of AMS-PFBH

图 11 AMS-PFBH 的扩展性评估

图与执行并行. 其次, 针对构图与执行并行中线程间的主存资源竞争问题, 提出了从核资源调节方法及参数选择方法. 最后, 进行实验测试, 结果表明: 新实现的 AceMesh 运行时系统在细粒度任务场景下超越了主核集中式并发构图, 相比于 SW39000 处理器上的 OpenACC 编程模型表现更为优越.

未来, 将进一步研究嵌套图的并行构图算法, 并探索更高效的调度策略.

作者贡献声明: 傅游确定论文的整体逻辑架构及研究思路, 修改论文; 贾淑慧负责撰写论文、进行部分实验方案的设计与验证; 陈莉确定论文研究思路并指导论文撰写; 花嵘负责完善论文内容; 杜云龙负责完成部分实验的设计与验证; 高希然负责部分算法思路与论文结构的讨论及修改.

参 考 文 献

[1] Bosilca G, Bouteiller A, Danalis A, et al. DAGuE: A generic distributed DAG engine for high performance computing[R]. Knoxville: Innovative Computing Laboratory, 2010

[2] Augonnet C, Thibault S, Namyst R, et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures[C]//Proc of the Int Conf on Parallel Processing Workshops. Piscataway, NJ: IEEE, 2009: 474−483

[3] Perze J, Badia R, Labarta J. A dependency-aware task-based programming environment for multicore architectures[C]//Proc of the IEEE Int Conf on Cluster Computing. Piscataway, NJ: IEEE, 2008: 142–151

[4] Augonnet C, Thibault S, Namyst R, et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures[C]//Proc of Euro-Par 2009 Parallel Processing: 15th Int Euro-Par Conf. Berlin: Springer, 2009: 863–874

[5] OpenMP ARB. OpenMP application program interface version 4.0[C]//Proc of the OpenMP Forum, Technical Report. New York: OpenMP ARB, 2013

[6] Duran A, Ayguade E, Badia R M, et al. OmpSs: A proposal for programming heterogeneous multi-core architectures[J]. Parallel Processing Letters, 2011, 21(2): 173−193

[7] Chen Li, Tang Shenglin, Fu You, et al. AceMesh: A structured data driven programming language for high performance computing[J]. CCF Transactions on High Performance Computing, 2020, 2(2): 309−322

[8] Fu Haohuan, Liao Junfeng, Yang Jinzhe, et al. The Sunway Taihulight supercomputer: System and applications[J]. Science China Information Sciences, 2016, 59(7): 1−16

[9] Slaughter E, Wu Wei, Fu Yuankun, et al., Task bench: A parameterized benchmark for evaluating parallel runtime performance[C]//Proc of SC20: Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020: 1–15

[10] Fu You, Du Leiming Gao Xiran, et al. Efficient task flow runtime for the new generation Sunway processor[J]. Computer Science, 2024, 51(12): 137−146 (in Chinese)
(傅游, 杜雷明, 高希然, 等. 新一代神威处理器上高效任务流并行系统[J]. 计算机科学, 2024, 51(12): 137−146)

[11] Maronas M, Sala K, Mateo S, et al. Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism[C]//Proc of 2019 IEEE 26th Int Conf on High Performance Computing, Data, and Analytics. Piscataway, NJ: IEEE, 2019: 383−394

[12] Podobas A, Brorsson M, Vlassov V. Turbobłysk: Scheduling for improved data-driven task performance with fast dependency resolution[C]//Proc of Using and Improving OpenMP for Devices, Tasks, and More: 10th Int Workshop on OpenMP (IWOMP 2014). Berlin: Springer, 2014: 45–57

[13] Pereira R, Roussel A, Carribault P, et al. Investigating dependency graph discovery impact on task-based MPI+ openMP applications performances[C]//Proc of the 52nd Int Conf on Parallel Processing.

New York: ACM, 2023: 163–172

[14] Bosch J, Álvarez C, Jiménez-González D, et al. Asynchronous runtime with distributed manager for task-based programming models[J]. Parallel Computing, 2020, 97(3): 102664

[15] Perez J M, Beltran V, Labarta J, et al. Improving the integration of task nesting and dependencies in OpenMP[C]//Proc of 2017 IEEE Int Parallel and Distributed Processing Symp (IPDPS). Piscataway, NJ: IEEE, 2017: 809−818

[16] Elshazly H, Lordan F, Ejarque J, et al. Accelerated execution via eager release of dependencies in task-based workflows[J]. The International Journal of High Performance Computing Applications, 2021, 35(4): 325−343

[17] Tan Xubin, Bosch J, Vidal M, et al. General purpose task-dependence management hardware for task-based dataflow programming models[C]//Proc of the Parallel and Distributed Processing Symp (IPDPS). Piscataway, NJ: IEEE, 2017: 244–253

[18] Álvarez D, Sala K, Maronas M, et al. Advanced synchronization techniques for task-based runtime systems[C]//Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2021: 334−347

[19] Agullo E, Aumage O, Faverge M, et al. Achieving high performance on supercomputers with a sequential task-based programming model[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(12): 2744-2758

[20] Yu Chenle, Royuela S, Quiñones E, Enhancing openmp tasking model: Per- formance and portability[C]//Proc of the 17th Int Workshop on OpenMP. Berlin: Springer, 2021: 35–49

[21] LLVM Foundation LLVM-project[DB/OL]. 2024[2025-02-20]. https://github.com/llvm/llvm/project/tree/main/ openmp/runtime

[22] GNV Project GCC-project[DB/OL]. 2024[2025-02-20]. https://github.com/gcc-mirror/gcc

[23] Wang Song, Hua Rong, Meng Xianfen, et al. Concurrent composition method for data-driven task graph execution[J]. Computer Engineering and Design, 2018, 39(3): 758−762 (in Chinese)
(王松, 花嵘, 孟现粉, 等. 数据驱动的任务图执行中并发构图方法[J]. 计算机工程与设计, 2018, 39(3): 758−762)

[24] Meng Xianfen. Composition optimization and parallelization of spacecraft application based on task graph scheduling system[D/OL]. QingDao: Shandong University of Science and Technology, 2019[2025-02-20]. https://link.cnki.net/doi/10.27275/d.cnki.gsdku.2019.000596 (in Chinese)
(孟现粉. 基于任务图调度系统的构图优化及航天飞行器应用并行化[D/OL]. 青岛: 山东科技大学, 2019[2025-02-20]. https://link.cnki.net/doi/10.27275/d.cnki.gsdku.2019.000596)

[25] Tang Xuhao, Liu Fagui, Wang Bin, et al. Research on GPU task scheduling[J]. Journal of Computer Research and Development, 2023, 60(6): 1262−1275 (in Chinese)
(唐续豪, 刘发贵, 王彬, 等. 跨云环境下任务调度综述[J]. 计算机研究与发展, 2023, 60(6): 1262−1275)

[26] Du Leiming. Implementation and optimization of sequential task flow runtime system on SW39000 processor[D]. QingDao: Shandong University of Science and Technology, 2024
(杜雷明. SW39000 处理器上顺序任务流运行时系统的实现与优化[D]. 青岛: 山东科技大学, 2024)

[27] Alakuş T B, Türkoğlu İ. A novel Fibonacci Hash method for protein family identification by using recurrent neural networks[J]. Turkish Journal of Electrical Engineering and Computer Sciences, 2021, 29(1): 370−386

[28] Álvarez D, Sala K, Maroñas M, et al. Advanced synchronization techniques for task-based runtime systems[C]//Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2021: 334–347



Fu You, born in 1968. PhD, professor, PhD supervisor. Member of CCF. Her main research interests include high performance computing, distributed computing, and intelligent computing.

傅　游, 1968 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为高性能计算、分布式计算、智能计算.



Jia Shuhui, born in 2001. Master candidate. Her main research interest includes high performance computing.

贾淑慧, 2001 年生. 硕士研究生. 主要研究方向为高性能计算.



Chen Li, born in 1970. PhD, associate professor. Member of CCF. Her main research interests include parallel programming languages and parallelizing compiling techniques.

陈　莉, 1970 年生. 博士, 副研究员. CCF 会员. 主要研究方向为并行编程语言、并行编译技术.



Hua Rong, both in 1969. PhD, associate professor, PhD supervisor. Member of CCF. His research interest includes high performance computing, artificial intelligence, and big data processing.

花　嵘, 1969 年生. 博士, 副教授, 博士生导师. CCF 会员. 主要研究方向为高性能计算、人工智能、大数据处理.



Du Yunlong, born in 1999. Master candidate. His main research interest includes high performance computing.

杜云龙, 1999 年生. 硕士研究生. 主要研究方向为高性能计算.



Gao Xiran, born in 1993. PhD Candidate. His main research interest includes programming models and optimization of HPC programs.

高希然, 1993 年生.博士研究生. 主要研究方向为高性能计算程序的编程模型和优化.