软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(1):77−90 [doi: 10.3724/SP.J.1001.2013.04339]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

任务并行编程模型研究与进展*

王 蕾, 崔慧敏, 陈 莉, 冯晓兵

(计算机体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

通讯作者: 王蕾, E-mail: wlei@ict.ac.cn, http://www.ict.ac.cn

摘 要: 任务并行编程模型是近年来多核平台上广泛研究和使用的并行编程模型,旨在简化并行编程和提高多核利用率.首先,介绍了任务并行编程模型的基本编程接口和支持机制;然后,从3个角度,即并行性表达、数据管理和任务调度介绍任务并行编程模型的研究问题、困难和最新研究成果;最后展望了任务并行未来的研究方向.

关键词: 任务并行;并行编程模型;任务窃取调度;并行性表达

中图法分类号: TP312 **文献标识码**: A

中文引用格式: 王蕾,崔慧敏,陈莉,冯晓兵.任务并行编程模型研究与进展.软件学报,2013,24(1):77−90. http://www.jos.org.cn/1000-9825/4339.htm

英文引用格式: Wang L, Cui HM, Chen L, Feng XB. Research on task parallel programming model. Ruanjian Xuebao/Journal of Software, 2013,24(1):77−90 (in Chinese). http://www.jos.org.cn/1000-9825/4339.htm

Research on Task Parallel Programming Model

WANG Lei, CUI Hui-Min, CHEN Li, FENG Xiao-Bing

(State Key Laboratory of Computer Architecture (Institute of Computing Technology, The Chinese Academy of Sciences), Beijing 100190, China)

Corresponding author: WANG Lei, E-mail: wlei@ict.ac.cn, http://www.ict.ac.cn

Abstract: Task parallel programming model is a widely used parallel programming model on multi-core platforms. With the intention of simplifying parallel programming and improving the utilization of multiple cores, this paper provides an introduction to the essential programming interfaces and the supporting mechanism used in task parallel programming models and discusses issues and the latest achievements from three perspectives: Parallelism expression, data management and task scheduling. In the end, some future trends in this area are discussed.

Key words: task parallelism; parallel programming model; work stealing scheduling; parallelism expression

半导体技术的进步,为处理芯片的设计者提供了更多的资源来实现更高性能的芯片.但是由于功耗墙等因素的限制,传统的依靠增加结构复杂度、提高主频而获得性能提升的方法已经很难实施[1,2].为了延续摩尔定律,充分利用日益丰富的片上晶体管资源,自 2005 年起,计算机处理器设计被迫从单核芯设计转向多核芯设计,通过增加片上的并行计算能力继续提高处理器性能.摩尔定律也被重新解释成:在基本不变的主频下,单个芯片上的处理器核的数目每一代(约两年)增加 1 倍.目前,双核、4 核、甚至 8 核的通用多核处理器已经普及,例如:Intel 在 2008 年推出了 4 核 Xeon 5500 系列,2010 年推出了 6 核 Xeon 5600 系列和 8 核 Xeon 7500 系列;IBM 在 2010 年 2 月发布了 Power 7 多核处理器,包含 8 个核,每个核支持 4 个硬件线程,共支持 32 个并发的硬件线程;AMD 已推出 12 核的 Magny-Cours 处理器.

* 基金项目: 国家自然科学基金(61202055, 60970024, 60925009, 60921002); 国家高技术研究发展计划(863)(2012AA010902); 国家重点基础研究发展计划(973)(2011CB302504); "核高基"国家重大科技专项基金(2009ZX01036-001-002, 2011ZX01028-001-002)

收稿时间: 2012-07-23; 定稿时间: 2012-10-19; jos 在线出版时间: 2012-11-23

CNKI 网络优先出版: 2012-11-23 12:14, http://www.cnki.net/kcms/detail/11.2560.TP.20121123.1214.005.html

多核芯片的出现,使得计算机编程模式面临着由传统串行编程模式向新型并行编程模式转变的巨大压力.在多核芯片出现之前,随着单核处理器性能的不断提高,串行应用程序执行速度就会被加快.但目前,追求高性能单核处理器时代已经结束,人们再也享受不到“免费的午餐”,只能通过并行的方式来提升串行应用程序的性能,程序员开始走向并行编程之路[3].

传统的并行编程模型主要有两个推动力:一是科学计算应用(主要是物理问题的模拟)对于性能和问题规模的无止境追求,这类应用常常存在一种较优的、静态的计算划分方法,具有明显的数据局部性,容易实现负载平衡.相应的语言和接口[4]包括分布存储系统上的 MPI 和共享存储上的 OpenMP(2.5 版本之前).这些语言模型都是让用户描述计算在虚拟处理器集合上的映射方法,在集群和传统的共享存储多处理器(SMP)机器上可以得到良好的性能扩展性.但 OpenMP(2.5 版之前)不能适应更广泛的不规则应用,MPI 则让用户手工处理并行进程之间的同步和通信.第 2 个驱动力是一些关键的非科学计算应用希望通过并发执行提高性能,常用的编程接口是 Pthread 等,这里,用户直接操纵线程的创建、同步和通信,编程层次更低.

可以看出,传统的并行编程模型或者面向专家级、资深程序员,或者只能适应规则的应用.多核时代需要的是面向更广阔应用领域的、易编程、高产能的并行编程工具.近几年涌现出许多新型并行编程模型,其中,任务并行编程模型因为具有以下优点而成为多核平台上首选的并行编程模型:

(1) 为程序员提供了两类并行控制结构,分别是支持规则并行的循环并行控制结构和支持非规则并行的嵌套并行控制结构.这两类并行控制结构均能实现数据并行模式和任务并行模式;

(2) 逻辑任务与物理线程分离.程序员只需考虑如何划分逻辑任务,使用合适的并行控制结构并行逻辑任务,而不需要考虑任务调度和处理器核数,提高编程层次,简化并行编程;

(3) 运行时系统负责任务调度,每个核对应一个物理线程,每个物理线程会执行许多逻辑任务.这种在用户态空间进行的任务调度大大降低了调度的开销,从而提高了多线程程序的执行效率.同时,运行时系统采用任务窃取调度算法获得负载平衡,提高多核的使用效率.

基于共享存储的任务并行编程模型[5,6]成为多核平台上广泛研究和使用的并行编程模型,例如,近几年 MIT 提出的 Cilk[5,6]/Cilk++[7]、Intel 提出的 Threading Building Blocks(TBB)[8]、微软提出的 Task Parallel Library (TPL)[9]、Java 的 fork-join 框架[10]、IBM 提出的 X10[11]和 Rice 提出的 Habanero-Java (HJ)[12]都是任务并行编程模型.2008 年,OpenMP 的 3.0 规范[13]也增加了对任务并行的支持.随着任务并行编程模型的逐渐完善,商业软件开发商开始使用它进行并行软件开发.例如:Adobe 多个 Creative Suite 5 系列应用程序采用 Intel 的 TBB 帮助客户创建、交付并优化高度压缩的数字内容;Autodesk、Adobe、梦工厂、Avid 和 Epic Games 等数百家独立软件开发商开始使用 Intel 的 TBB 开发并行软件.

本文第 1 节介绍任务并行编程模型的基本编程接口、运行时支持机制和关键技术挑战.并行性表达、数据管理和任务调度是任务并行编程模型的 3 个主要的研究内容.第 2 节和第 3 节分别介绍这 3 个方向的最新研究成果.第 4 节总结并讨论任务并行编程模型未来所面对的挑战及其发展趋势.

1 任务并行编程模型概述

并行编程模型是底层体系结构与上层应用程序之间的桥梁,向上隐藏并行处理器的细节,提供给程序员并行表达的方法;向下充分利用硬件资源、高效且正确地完成应用需求.任务划分、任务映射、数据分布、通信和同步是设计并行编程模型时需要考虑的 5 个关键要素[2].任务并行编程模型主要关注共享存储的平台,数据分为共享和私有两种存储属性,通过共享数据进行通信.因此,该编程模型的研究重点是任务划分、任务映射和同步这个关键要素.

任务并行编程模型把任务作为并行的基本单位,提供任务划分和同步的编程接口,把任务划分和同步工作交给程序员完成,用户可以把应用程序划分出大量细粒度任务.然而,具体到每个任务到底是并行执行还是串行执行、在哪个物理核上执行以及如何实现任务之间的同步则由运行时系统完成.任务并行编程模型提倡嵌套的递归任务,并引入以任务窃取为核心的用户级线程调度,实现程序的高性能和动态的负载平衡.

1.1 任务并行编程模型的基本编程接口

任务并行编程模型有 3 种类型,分别是现有语言的并行扩展(Cilk/Cilk++和 OpenMP3.0)、新的并行语言(X10 和 Habanero-Java)以及并行库(TBB 和 TPL).Rice 大学根据教学经验比较了任务并行语言和并行库在并行编程方面的优缺点[14].他们认为,任务并行语言的优点在于程序员写并行程序时易于表达,其他程序员和程序分析工具阅读并行程序时易于理解;缺点是需要新语言结构的标准化.而任务并行模板库的优点在于不需要改变开发环境,即不需要新的编译器和集成开发环境;缺点是当用库的 API 表达任务并行时,代码很难理解和修改.

下面以 Cilk 语言为基础介绍任务并行编程模型的基本概念.20 世纪 90 年代中期,MIT 提出了多线程并行编程语言 Cilk[5,6],在 C 语言基础上扩展了 spawn,sync 关键字支持并行.spawn 关键字添加在函数调用前用于产生任务,spawn 产生的子任务可以和父任务并行执行;sync 相当于一个局部的栅障(barrier),在该同步点之前产生的所有子任务必须结束后才能继续执行.每个任务函数结束时有一个隐式的 sync,这样,每个子任务结束后都必须返回到父任务.在递归函数中的 spawn 会产生嵌套并行任务,随着执行 spawn 或 return 语句把任务压入或弹出任务队列.在一个 Cilk 程序中,所有的 spawn 调用将产生一个任务派生树,sync 对应的依赖关系在任务之间产生了 join 边,因此,一个 Cilk 程序的多线程执行就对应了一个 DAG 图,其中每个节点对应一个并发任务.spawn-sync 并行结构可以用于非规则的嵌套并行.

1.2 运行时支持

任务并行的核心技术之一是任务窃取(work-stealing)调度算法[5,6],运行时系统采用任务窃取调度算法把逻辑任务映射和调度到线程上执行,从而获得负载平衡.通常,任务窃取调度算法的实现方法是每个处理器核对应一个线程,每个线程维护一个双端队列(deque)存放任务状态信息,这个状态信息包括任务中的局部变量、PC 值、子任务的个数等,用于恢复被挂起任务或执行被窃取的任务.每个线程从自己双端队列的尾部压入准备好的任务或弹出已经执行完的任务;当自己双端队列为空时,从其他线程的双端队列头部窃取任务,首先恢复窃取的任务状态,然后根据状态跳转到 spawn 或者 sync 之后的指令开始执行.用这种双端队列的方法窃取线程可以不用中断工作线程的执行而得到任务,同时获得负载平衡.

任务窃取是一种基于有向无环图(DAGs)的调度算法,DAG 中节点代表一个任务,边代表依赖关系.如果一个任务的所有祖先节点都已完成,这个任务就准备好了.任务窃取调度负责把已准备好的任务派生到处理器核上执行.基于 DAG 的随机任务窃取调度算法在 P 个处理器核上的执行时间是 $T_P=T_1/P+O(T_\infty)$,其中,T_P 是在 P 个处理器核上的执行时间,T_1 是串行程序的执行时间,T_∞是关键路径的执行时间[15].该算法的空间要求是 S_1P,其中,S_1 是串行程序需要的空间[16].

任务窃取调度使得父子任务是否并行执行变得不确定.例如,线程执行到父任务 A 产生(spawn)子任务 B 时,把父任务 A 放入自己的任务队列尾端,然后开始执行子任务 B;若父任务 A 被其他线程窃取,这种情况下,子任务 B 和父任务 A 并行执行;若父任务 A 没有被窃取,线程执行完子任务 B,把父任务 A 从自己的任务队列尾端取出并执行,这种情况下,子任务 B 和父任务 A 串行执行.

综上所述,任务窃取的优点是:1) 根据线程的繁忙情况动态调度任务,获得很好的负载平衡;2) 在同步点,若某个任务 A 必须等待还未完成的子任务,则线程挂起该任务 A 去执行其他任务;执行完最后一个子任务的线程会唤醒并执行被挂起的任务 A.这种数据流驱动的方法能够高效利用计算资源;3) 递归并行与 cache 无关算法的有效结合,使得局部性敏感的应用高效利用 cache.例如,BLAS 库就是采用递归的 cache 无关算法.

1.3 关键技术挑战

任务并行编程模型提供显式的任务划分和同步编程接口以及隐式的任务映射机制.前者关注可编程性,后者关注执行效率.目前,任务并行编程模型支持非规则应用程序,把逻辑任务与物理线程分离,从而独立于处理器核数.但多核时代需要的是面向更广阔应用领域的、易编程、高产能的并行编程工具,该模型的编程接口(并行性表达和数据管理)和运行时支持(任务调度)面临如下挑战:

(1) 该模型的编程接口能支持的并行模式有限,需要丰富编程接口,表达多种多样的并行性.例如,spawn-

sync 能够实现嵌套并行控制结构,但不能高效实现循环级并行,于是,程序员需要把数据并行的应用程序转换成嵌套并行,才能用该模型编写并行程序.另外,无条件原子块结构和有条件原子块结构是重要的并行任务结构,如何表达以及如何高效支持都需要深入研究;

(2) 该模型把数据分为共享和私有两种,通过共享数据进行通信.但有些数据是部分任务共享,或者一个线程内执行的所有任务共享,因此需要对数据进一步区分共享范围,需要研究如何高效实现不同级别的共享数据;

(3) 该模型的运行时系统负责把逻辑任务映射到物理线程上去执行,其核心任务是提高执行效率.存在的问题有:

(a) 运行时系统是一个软件层,与应用程序链接在一起,运行在用户空间上.用软件实现任务窃取是有代价的,问题是能否进一步降低运行时系统开销;

(b) 任务窃取采用最早任务优先窃取策略,该策略的"深度优先执行"能够提高 cache 的利用率.但随机选择线程进行任务窃取,而没有考虑多核处理器的存储层次和处理器架构特点,对于局部性敏感的应用会产生影响.因此,任务调度时需要根据存储部件的层次、容量、访问延迟以及数据的访问局部性、重用度和层次性等因素进行局部性敏感的调度;

(c) 集群系统和众核处理器都远比多核处理器要复杂,拥有更大量的计算资源,如何管理和使用硬件资源,充分利用体系结构的并行性和局部性来提高性能,也需要深入加以研究.

近几年,学术界和工业界深入研究了上述问题,本文第 2 节和第 3 节分别介绍编程接口和运行时支持的国内外最新研究成果.

2 并行性表达和数据管理研究

任务并行编程模型提供显式的任务划分和同步编程接口,学术界和工业界通过深入研究并行性表达和数据管理两个方面以提高该模型的可编程性.下面介绍最新研究成果.

2.1 尾端严格的特性(terminally-strict computation)

为了有效地进行任务调度,语言设计者总是要求多线程计算的依赖图满足严格特性[15],即在一个应用的 DAG 任务图中,join 边只能从派生树的某节点指向其任一祖先节点.Cilk 中,spawn-sync 的计算模式被称为完全严格的(fully-strict computation)[15],也就是说,在多线程执行的 DAG 中,join 边总是从派生树的某节点指向父亲节点.X10[11]是 IBM 提出的类 Java 的新型并行编程语言,它所表达的任务树比 Cilk 风格的任务树更宽泛,即每条 join 边都是从一个任务的最后一条指令指向其派生树的某祖先,也被称为尾端严格的(terminally-strict computation).

X10[11]主要面向 NUCC(non-uniform cluster computing)结构,属于异步全局分割地址空间(asynchronous partitioned global address space,简称 APGAS)编程模型.库所(place)是 X10 的核心概念,用于表达系统中访存不均匀性.一个库所必须映射到缓存一致性的计算单元上(如 SMP 节点),一个 SMP 节点也可以有多个库所.库所内使用异步活动(asynchronous activities)作为并行执行的基本单位,async 用于产生一个新活动.async (P) S 语句表示产生一个子活动在指定的库所 P 内执行语句 S,子活动可以与父活动并行执行.finish 〈.stmt〉相当于 Cilk 的 sync,是一个同步点,父任务在执行完 stmt 后等待 stmt 期间产生的所有子任务(包括以传递方式派生的子任务)结束后才能继续执行.

async-finish 类似于 Cilk 的 spawn-sync,但在 X10 中每个子活动结束时,没有隐式的 finish,所以 X10 任务结束后可以返回到某个祖先任务.X10 编译器[17]为每个任务并行的区域(每对 async-finish)用 startFinish 和 stopFinish 标记,运行时系统记录此间产生了多少个任务,在 stopFinish 处检查这个并行区域内的任务是否全部完成,称为全局终止的检测.

2.2 循环级并行表达

早期的任务并行机制只处理不规则的并行性,但其实它可以扩展到支持循环级并行,包括没有迭代间依赖的 forall 并行、归约并行和 scan 并行等.

Cilk++[7]是 2008 年 Cilk Arts 公司(2009 年被 Intel 收购)提出的并行语言,将 Cilk 中的技术扩展到 C++语言.Cilk++以 cilk_for 关键字支持循环级并行,用于处理规则的 forall 数据并行.其具体实现方法采用分治法切分迭代空间,把循环并行转换为嵌套并行,从而把任务压入或弹出任务队列.TBB 的 parallel_for 用于支持循环并行[18],除了以上任务调度方法,也支持 auto_partitioner 和 affinity_partitioner 等调度方式,同时提供 range 的类型以支持多维迭代空间上的并行性.

2008 年,OpenMP3.0[13]添加了对任务并行的支持,omp tas*k* 和 taskwait 类似于 Cilk 的 spawn 和 sync,但可以通过 tied/untied 制导选择任务是否绑定到线程上,task 区域绑定到最内层的 parallel 区域,task 区域所绑定的线程集合构成当前的并行组.OpenUH[19]是一个开源的 OpenMP 实现,在其运行时系统中,每个线程有两个任务队列,私有的和共享的.tied 的任务放入私有队列,untied 的任务放入共享队列.该运行时系统与典型的任务并行的运行时系统有两点不同:

- 一个是 Cilk 对 DAG 任务图采用“深度优先执行”,任务队列是 LIFO 的栈,spawn 产生一个子任务,把该子任务放入任务队列中,挂起父任务,开始执行子任务;而 OpenUH 采用“宽度优先执行”,任务队列是 FIFO 的队列,父任务产生完所有子任务,直到遇到 taskwait 后,父任务被挂起,从任务队列中取一个子任务开始执行;
- 另一个是 OpenUH 中的线程只从其他线程的共享队列中窃取任务.

对归约并行,OpenMP 的数据并行结构中提供了 reduction 子句.任务并行的语言中,Cilk++通过指定一个 Reducer 对象(一种特定的超级对象 hyperobject,见第 2.5 节),实现数据结构私有化,与现有的 Cilk 并行结构相结合,实现归约并行.

Threading Building Blocks(TBB,线程构建模块)是由 Intel 公司开发的一个 C++并行模板库[8],提供并行算法模板、同步原语、并发容器和可伸缩的内存分配函数来支持高效并行.TBB 提供丰富的并行算法模板,比如 parallel_for,parallel_do,parallel_for_each,pipeline,parallel_reduce,parallel_scan,parallel_sort,parallel_invoke,显然,归约并行是被支持的.TBB 底层实现运用了 Cilk 的任务窃取调度技术来保障负载平衡.2010 年,微软.NET Framework 4 提供了一个任务并行库 Task Parallel Library(TPL)用于并行编程,与 TBB 类似.并行库为 C++程序员编写并行程序提供丰富的基础支持,但却需要程序员对并行库提供的模板有较为深刻的理解.TBB 和 TPL 的推广使用,标志着任务并行编程模型已经走进工业界.

2.3 原子块结构

无条件原子块结构和有条件原子块结构是重要的并行任务结构.X10 属于异步全局分割地址空间编程模型,是分布式存储系统上的并行编程语言.一个库所被映射到一个共享存储节点上,这样,所有库所都被映射到由共享存储节点组成的分布式存储系统上.X10 语言支持无条件原子块结构 atomic S 和有条件原子块结构 when(E) S.当条件 E 不为真时,有条件原子块结构 when(E) S 只能被挂起.文献[20]的任务窃取调度算法在支持 async-finish 异步任务结构的同时,也支持 when(E) S.具体方法是:每一个库所共享一个额外队列存放被挂起的有条件的原子块;当执行到 when 结构时,E 为假,就把这个原子块放入额外队列中,挂起该原子块,成为空闲线程.当线程执行完某个原子块时,把共享的额外队列中所有原子块放入自己的任务队列中,这样,每一个被挂起的原子块的条件都会重新再计算一遍.

2.4 移相器——支持性能扩展的同步和归约

在线程数越来越多的并行平台上,栅障同步和规约的性能扩展性就成为非常重要的问题.竞赛树和 butterfly barrier 通过层次的同步降低栅障同步的开销,但其通信关系是固定的,不能适应任务并行的动态特性.

2008 年,Rice 大学借鉴 X10 丰富的任务并行结构 async,finish,future,forall,foreach,ateach,提出 Habanero Java

(HJ)任务并行语言[12].Rice 大学的 Habanero Java 语言中引入了移相器(phaser)及其累加器(accumulator)[21]这两个概念,支持集合通信和点到点同步.移相器是 X10 中同步钟的一个扩展,它是一个同步对象,具有 4 种模式(如图 1 所示),支持 4 种操作:创建(new)、注册(phased(ph1⟨mode⟩,…))、退出(drop)和推进(next).移相器与其累加器相结合把集合操作划分为数据发送、计算、结果取回和同步,支持计算和通信的重叠.利用以上语言对象,Habanero Java 能够支持层次的移相器,而每一层的同步关系是动态确定的.此外,移相器具有免死锁的安全特性,简化了用户编程.

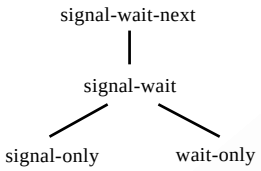


Fig.1 Capability lattice for phasers

图 1 移相器的能力模式偏序格

在 64 线程和 128 线程的 Niagara-2 SMP 上的测试结果表明:利用层次移相器,栅障和规约的开销降低得很明显;在应用程序的测试方面,数据集小、同步开销比例大的情况下,可以看出层次移相器比扁平的移相器更具优势.可以设想,随着 SMP 节点内核心数的增加,层次移相器可以期待有更大的性能收益.

2.5 数据属性表达

基于共享存储的任务并行编程模型用全局变量进行通信和同步.全局变量在串行程序中用于简化编程,但在并行程序中会导致数据竞争.用锁避免对共享数据的竞争会影响并行度,从而不能获得高性能.区分不同的数据属性,并对一些数据属性进行优化是语言和运行时系统的任务.Cilk++提供超级对象(hyperobjects)来描述数据属性,而微软的 TPL 提供了 combinable object 结构.

Cilk++提供超级对象,使多个任务能够互不影响地访问全局变量.运行时系统保证每个线程有全局变量的私有拷贝,线程访问私有拷贝,这样,在不需要锁的情况下消除了线程间竞争的可能性.Frigo 等人[22]描述了 Cilk++对 reducer,holder,splitter 这 3 种数据属性的支持.这 3 种数据属性都是超级对象,reducer 是规约变量,必要时在 spawn 前先私有化,然后在 sync 之后合并结果;holder 是线程内私有变量(thread local storage);splitter 是回溯搜索中记录当前状态的全局标量.它们都是用 hypermap hash table 来实现的.Cilk++提供 reducer_list, reducer_max,reducer_min,reducer_opadd,reducer_opand,reducer_opor,reducer_opxor,reducer_ostream,reducer_string 等类型的规约变量[7].

2.6 并发数据结构

一个并发数据结构允许多个线程并发访问和更新数据结构中的元素,它往往使用细粒度锁或 lock-free 技术等方法加以实现,在保证线程安全的同时得到并行加速比.但并发数据结构一般很复杂,难以理解,而且具有明显的实现开销,比常规、串行的 STL 数据结构性能要差.提供常用的并发数据结构是并行编程语言的任务.

高效的并发集合对象是很难实现的,Cilk++利用 reducer 超级对象实现了这样一个快速访问的并发集合 bag[23],多个线程可以同时向集合 bag 中增加或删除数据.这个并发集合是一个指针数组,每个数组元素是一棵二叉树.指针数组相当于一个二进制数,表明集合中元素的个数.例如,指针数组 A[]是 010111,A[0]是有 1 个节点的二叉树,A[1]是有 2 个节点的二叉树,A[2]是有 4 个节点的二叉树,A[3]和 A[5]没有节点,A[4]是有 16 个节点的二叉树,指针数组 A 共有 23 个节点,即 010111_2 是 23.当有两个线程需要共同访问这个并发集合时,这个指针数组 A 能够快速分裂成两个大小相同的指针数组,两个线程分别访问不同的数组;sync 或 join 之后,也能快速地把两个指针数组合并成一个数组.这种无锁的并发集合比基于锁的并发集合性能好很多.

TBB 也提供一套丰富的并发数据结构,包括映射、动态数组和队列,concurrent_hash_map,concurrent_vector,

concurrent_queue,concurrent_bounded_queue 和 concurrent_unbounded_queue.TBB 中每个线程还有一个内存分配器,提供两种内存分配模板 scalable_allocator 和 cache_aligned_allocator.

2.7 锁外协助

临界区是用于防止多个线程同时执行一段特定代码的机制,常用锁对临界区进行保护,每次只能有一个线程执行临界区内的代码.假设线程 1 获得锁 *L*,开始执行临界区 *A*.这时,线程 2 也想获得锁 *L*,它只能被阻塞,等待线程 1 释放锁 *L*.文献[24]提出 help lock 的方法:当线程 2 不能获得锁 *L* 时,线程 2 不是阻塞而是挂起自己的任务,帮助线程 1 去执行临界区 *A* 的任务.这样,线程 2 没有忙等,充分利用了计算资源.这种 help lock 非常适合临界区较大的程序.

3 任务调度关键技术研究

任务并行编程模型提供隐式的任务映射机制,运行时系统采用任务窃取调度算法,把逻辑任务映射到物理线程上去执行,提高执行效率是运行时系统的核心任务.任务窃取调度算法的研究分为 3 个阶段——理论研究、算法实现方面的优化以及面向处理器结构的优化.

1) 理论研究阶段是从 20 世纪 90 年代到 21 世纪初,研究共享存储多处理器(SMP)架构上的多线程任务窃取调度算法,为任务窃取奠定理论基础,见第 3.1 节.
2) 算法实现方面的优化阶段是从 2006 年开始至今,研究多核平台上任务窃取实现方面的优化,主要包括控制任务粒度和局部性敏感的任务调度研究以及任务队列改进的研究,见第 3.2 节、第 3.3 节和第 3.6 节.
3) 目前正在研究面向处理器结构特点的任务调度,包括众核和集群上的任务窃取关键技术研究,见第 3.4 节和第 3.5 节.

3.1 任务窃取调度算法研究

20 世纪 80 年代,函数式语言的研究引出最早任务窃取的实现,包括 Burton 和 M.R.Sleep[18]以及 Halstead[25]和 Mohr 等[26].20 世纪 90 年代和 21 世纪初,共享存储多处理器架构上多线程的任务窃取调度算法在各方面都有突破性的进展,如下:

- Blumofe 和 Leiserson[15]描述了多线程的任务窃取调度算法;
- Squillante 和 Nelson[27]研究了任务窃取和任务共享两种调度算法,认为在共享存储系统中,任务窃取调度优于任务共享调度;
- Frigo 等人[5,6]提出了 Cilk-5 并行语言的实现,是任务窃取调度算法实现的经典文章;
- Blelloch 等人[16]给出多线程的任务窃取调度所需的时间和空间上限;
- Fatourou 和 Spriakis[28]扩展 Cilk 多线程计算模型,Cilk 中的任务只能依赖于父任务,但在这个扩展模型中,任务可以依赖于祖先任务;
- Berenbrink 等人[29]经理论分析后得出任务窃取具有稳定性的结论.

这些工作从不同方面对任务窃取调度进行了理论分析,为任务窃取调度打下基础.

有许多工作从不同侧面研究任务窃取调度的优化:

(1) 亲密性调度研究.Squillante 和 Lazowsks[30]提出共享存储多处理器上使用 cache 局部性的调度算法,其中建立了 cache 模型,评测 cache reload time 和 bus 堵塞情况;Acar 等人[31]利用数据局部性信息进行调度,每个处理器除了有自己的任务队列以外,还有一个队列用于指向与自己相关的任务,窃取时首先考虑这个队列里的任务.
(2) 负载平衡调度策略研究.Hamidzadeh 和 Lilja[32]提出集中式调度策略,专门拿出一个处理器考虑局部性因素进行调度;Rudolph 等人[33]通过分层的负载共享方法,结合集中式调度和分布式调度,获得二者的好处.

(3) 异构系统调度算法.Bender 和 Rabin[34]假设共享存储系统中各个处理器的执行速度不同,在这种异构系统中,执行速度快的处理器可以打断执行速度慢的处理器,并把它正在执行的任务窃取过来执行.

3.2 控制任务粒度的调度算法

细粒度并行有许多优点[35],例如:在许多应用中可以暴露更多的并行性,对于拥有上百个核的片上多处理器(CMP)结构非常重要;在非规则计算中,运行时系统有更多的自由度避免负载不平衡;当环境发生变化时,例如,某个核发生故障或多个进程一起运行时导致可获得的核数发生变化,运行时系统可以根据当前可获得的核数进行负载平衡调度,从而适应环境变化.任务并行编程模型属于细粒度并行,它让程序员去表达所有可获得的并行性,然后运行时系统负责把这些任务映射和调度到物理线程上去执行.这种编程方法会产生大量细粒度逻辑任务以确保负载平衡;并用逻辑任务代替物理线程,独立于处理器个数,提高了并行编程层次.

运行时系统用软件实现任务窃取调度算法是有代价的,产生大量的细粒度任务会加重系统开销;相反地,产生少量的粗粒度任务会造成负载不平衡,从而使性能下降.由此可见,系统开销与任务数量成正比,而负载平衡与任务数量成反比,合适的任务粒度是在系统开销和负载平衡两个因素之间加以权衡.如何获得合适的任务粒度是任务并行编程模型的一个重要问题.

一个任务对应一个函数,例如执行 spawn foo()语句时,线程把任务 foo 的状态信息放入自己任务队列的尾部,同时在栈空间上分配一个 foo 函数的栈帧.若只是函数调用,就不会有任务产生的开销.所以,串行函数可以减少系统开销,并行任务用于负载平衡,串行函数和并行任务互惠的方法是控制任务粒度的主要手段.目前,任务窃取的实现方法只能做到从并行任务转换到串行函数,还不能做到串行函数转换到并行任务.下面的方法主要从串/并行互惠角度来控制任务粒度,cut-off 策略还是只能从并行任务转换到串行函数,只是需小心选择转换点;自适应任务粒度能做到并行任务和串行函数相互切换,由于没有考虑局部性,局部性敏感的应用可能使性能提升不大;把运行时栈和任务队列合二为一,这样可以减少任务队列维护的开销,但这种方法需要修改操作系统.

3.2.1 Cut-Off 策略

一些研究用 cut-off 策略控制函数调用的递归深度以减少任务数量,从而降低任务产生的开销,同时也控制了任务粒度[36−38].Cut-Off 策略通常指定在派生树(或函数调用树)上的一个递归深度,当超过这个深度时,不产生任务.对于平衡的派生树,这个策略工作得很好;但对于不平衡的派生树,cut-off 策略会引起系统“饥饿”,即有些线程由于没有任务可以执行,而被迫闲置,导致负载不平衡,从而降低执行效率.

Cut-Off 策略的 5 种实现方法是:

方法 1.程序员提供一个 cut-off 递归深度,或运行时系统设置一个缺省的深度.这种方法简单,但不能适应环境变化.

方法 2.batching[39],运行时系统根据当前任务队列的大小设置 cut-off 深度,从而控制任务粒度.但是这种方法需要程序员设置串行程序的阈值,并且进行手动的性能调优.

方法 3.profiling[40],首先进行工作集轮廓信息收集,然后用收集后的信息执行 cut-off.该方法对于一些有工作集的应用比较有效,但对于没有明确工作集的应用,例如回溯搜索、分支界限搜索和游戏树,该方法就无能为力了.

方法 4.运行时系统支持的自适应 cut-off 技术[41],运行时系统收集每一层产生的任务数,如果任务数大于两倍的线程数,就 cut-off,不再产生新任务.但对于不规则应用程序,这种自适应的 cut-off 技术无法预测出哪个分支是任务多的分支,所以会造成负载不均衡,从而影响性能.

方法 5.用户制导的自适应 cut-off 技术[42],希望能够预测每个分支的大小,直到这个分支不太大时就 cut-off.具体实现方法是,程序员为每个函数提供一个代价计算语句,然后运行时系统预测每个分支的大小,并决定是否 cut-off.

3.2.2 自适应任务粒度

自适应任务粒度的核心思想是,当所有线程都繁忙时就不产生任务,直到有线程空闲需要任务时,繁忙线程再产生任务.任务窃取调度算法的传统实现方法使得并行任务可以调用串行函数,但串行函数不能产生并行任

务,所以,当进入串行函数后就无法再产生任务.要得到自适应的任务粒度,难点在于怎样在串行函数中产生并行任务.

文献[43]可以得到嵌套并行中的自适应任务粒度.该方法引入一个特殊任务用于在串行函数中再次产生并行任务.特殊任务保存了串行函数的执行状态信息,不能被窃取,必须等待它的所有子任务结束后才能继续执行.这种自适应任务产生策略可以根据实际执行情况在并行任务和串行函数间切换,从而得到自适应任务粒度.由于cilk_for循环并行具体实现时采用分治法切分迭代空间,把循环并行转换为嵌套并行,因此该方法也能解决循环并行的任务粒度问题.

文献[44]可以得到循环并行中的自适应任务粒度.具体方法是:每次执行一个迭代前检查自己的任务队列是否为空,如果为空,就把剩下没有执行的迭代空间一分为二,并把要执行的任务放入自己的任务队列,继续执行;否则,不放入任务队列中,继续执行.这种方法每次切分派生树中的叶子节点,从而解决循环并行的任务粒度问题.

3.2.3 其他方法

以下研究是从操作系统、硬件结构和编译优化的角度来解决运行时系统代价大的问题.

文献[45]通过修改操作系统来支持线程私有的内存映射(thread-local memory mapping,简称 TLMM),采用TLMM机制合并任务队列和函数调用栈,这样就可以实现串行函数到并行任务的转换,从而提高性能.TLMM可以指定一个进程的某个虚地址空间区域为线程私有的,亦即对于这个虚地址空间,每个线程有相同的虚地址,但占有不同的物理内存页.用TLMM实现每个线程的栈空间,窃取时窃取任务和该任务祖先的栈帧,这样就把任务队列和函数调用栈合并到一起,降低了任务产生的开销.

文献[35]观察到:用软件实现调度方法灵活性大,但代价也大;硬件实现调度方法代价低,但缺乏灵活性.他们用软、硬件结合的方法,软件维护任务队列,硬件进行任务窃取.这种方法能够降低任务产生和任务队列管理的开销,从而提高性能.

文献[46]针对双层循环,采用编译优化的方法把内层 parallel_for 交换到外层,降低了任务数量,从而减少了任务产生和结束的代价,提高了性能.

3.3 局部性敏感的调度算法

任务窃取采用最早任务优先窃取策略,该策略的“深度优先执行”能够提高cache的利用率.但随机选择线程进行任务窃取,而没有考虑处理器架构特点,对于局部性敏感的应用会有影响.局部性敏感的调度算法主要关注如何选择要窃取的线程.

3.3.1 Cache 亲密性调度

文献[31]提出 cache 亲密性调度方法,提高 cache 利用率.具体方法是,每个线程除了任务队列外还有一个邮箱(mailbox),用于存放与该线程亲密的任务.当线程 1 产生一个与线程 2 有亲密性的任务时,线程 1 把指向该任务的指针放入线程 2 的邮箱中.当线程 2 完成自己任务队列中的任务时,先检查自己的邮箱,执行邮箱中的任务,最后再窃取任务.

由于在开始执行一个循环时大部分线程都处于空闲并不停地窃取任务,TBB 的 Robison 等人[47]改进 cache 亲密性调度方法,规定窃取线程不允许窃取已经放入某个空闲线程的邮箱中的任务,从而解决初始阶段不能保证 cache 亲密性调度的问题.

3.3.2 多路多核处理器架构上的局部性敏感调度

目前,服务器基本上都是多路多核架构(multi-socket multi-core architecture),处理器包含多个多核芯片,片内多核共享 L3 cache,片间多核共享内存,提供 cache 一致性.任务窃取调度策略是随机选择一个线程进行任务窃取,没有区分片内和片间的线程,而对于局部性敏感的应用,这种随机任务窃取调度策略降低了 cache 利用率,从而也导致性能下降.

文献[48]提出了 Cache Aware Bi-tier(CAB)调度策略,根据硬件结构对线程进行分组,片内的线程分为一组,每个线程有一个组内任务队列,每个组有一个组间任务队列.运行时系统顺着调用树把任务分成组内任务和组

间任务,然后放入相应的任务队列中.线程执行完自己的组内任务队列中的任务后,随机选取同组其他线程进行任务窃取;当一个组的所有组内任务队列为空时,从本组的组间任务队列中窃取任务;当一个组的所有组内任务队列和组间任务队列都为空时,就从其他组的组间任务队列中窃取任务.

文献[49]提出了分层调度策略,不是每个线程有一个任务队列,而是芯片内的线程共享一个任务队列,这样只在芯片间进行任务窃取调度.在这种方法中,共享任务队列可能会成为瓶颈.

3.4 集群系统上的分层任务调度

X10 属于异步全局分割地址空间编程模型,X10 的任务调度研究需同时考虑库所内和库所间的任务调度.

库所内的任务调度与 Cilk 等传统的任务调度存在一些不同:对于 X10 中尾端严格的特性,第 2.1 节已经介绍了全局终止检测的方法;对于函数的并行特征没有标明的问题,运行时系统需要为每个线程维护一个函数调用关系队列和一个任务队列,函数调用关系队列包含串行函数和并行任务,任务队列只存放并行任务.这种处理方法类似于把每个函数(不区分串行函数和并行任务)都放入任务队列,系统实现代价大.文献[17]还针对图的生成树应用容易造成栈溢出问题提出了 help first 调度策略,即产生任务时不立即执行子任务,而是把子任务放入任务队列中继续执行父任务.文献[50]介绍了如何自适应地在 help first 和 work first 两种调度策略中进行转换.文献[39]描述了支持 X10 的运行时库.

库所间的负载平衡需要把任务窃取扩展到分布式存储系统上,但任务窃取在共享存储系统上研究的比较多,在集群上对该算法还认识不够.文献[51]实现了集群上的任务窃取调度算法,并提出几种优化方法改善性能:

1) 分裂任务队列,每个任务队列分成局部访问和共享访问两部分:从 head 到 split 是共享访问部分,用于其他线程任务窃取;从 split 到 tail 是局部访问部分,用于本线程存取任务.局部访问不需要锁,共享访问部分需要锁.线程定期调整 split 点,平衡任务量;
2) 共享存储系统上的任务窃取时,一般从任务队列头部窃取一个任务.但在分布式存储系统上,窃取任务代价大,所以每次窃取多个任务.实验结果显示:每次窃取任务队列中的一半任务,性能最好.

文献[52]认为,在分布式存储系统上实现任务窃取调度算法有两个困难:

1) 在共享存储系统上,窃取线程争取做到不打扰工作线程窃取到任务.而在分布式存储系统上,很难做到窃取任务的同时不干扰工作线程;
2) 如何确定分布式任务结束是个难点.

文献[52]中提出生命线图解决思路,每个库所增加一项记录被窃取的线程号,当某个库所没有任务时,尝试窃取这个库所任务的窃取线程号就被记录下来.这样,库所和被窃取的线程号就形成一个有向图,这个有向图被称为生命线图,被窃取的线程号称为库所的入边.当一个库所窃取到任务后,就把它的任务分给入边线程,然后清零入边.若一个线程尝试窃取 w 次后都未成功,则这个线程变成静止节点,不再窃取任务,直到根据生命线图被分到任务后再次被激活.当所有线程都变成静止节点后,整个任务结束.

3.5 众核处理器上的任务调度

文献[53]针对众核处理器系统的核资源利用效率较低的问题,提出一种支持核资源动态分组的自适应调度算法 CASM.该算法基于资源分区自治思想,通过对任务簇的拆分与合并,实现核资源的隔离优化访问.

众核处理器系统上软件实现运行时系统性能不好,于是,文献[54,55]研究用硬件实现任务调度.文献[54]研究了众核体系结构对任务并行编程模型的硬件支持,提出 DAG consistency 的缓存一致性协议,在此基础上实现了任务并行编程模型.实验结果表明,当线程数大于 16 时,由于静态路由导致片上网络带宽利用不均衡和有限的访存带宽,因而难以获得理想的加速效果.文献[55]采用硬件结构支持任务窃取调度算法,并根据硬件资源使用情况有条件地产生任务,实现快速的任务分解.

3.6 任务队列的改进

由于工作线程和窃取线程都直接操作工作线程的任务队列,当一个窃取线程尝试窃取一个任务,而这个任务正是工作线程正要弹出的任务时就发生数据竞争.任务窃取机制用 THE 协议解决这种竞争[6].因为 THE 协议

需要原子操作,代价比较大.文献[56]采用复制队列(duplicate queue)的方法来代替传统的 THE 协议以避免数据竞争,复制队列不需要原子操作,从而提高了性能.

运行时系统启动时,为每个线程创建一个固定大小的任务队列,这可能会造成任务队列溢出.文献[57]提出了动态循环任务队列,可以解决任务队列溢出的问题.

4 结论和未来的挑战

任务并行编程模型是多核芯片出现后为了简化并行编程而研究的一类新型并行编程模型.程序员只需关注问题本身的并行性,而运行时系统进行负载平衡的任务窃取调度.目前,任务窃取调度是基于共享存储多核芯片,采用移动计算而不移动数据的方法.经过过去 15 年的发展,任务并行机制已被大量的并行语言所支持.现有的任务并行机制已能表达更宽泛的并行性,且支持多种数据属性的管理,任务调度技术得到深入研究,已经能够适应各种不同的应用类型.Intel 的 TBB 和微软 TPL 的推广使用也标志着任务并行编程模型走入工业界.比如,Intel 推出用于简化向量/矩阵操作的 Array Building Block(ArBB)编程模型的运行时系统就是通过调用 TBB 模板库来完成并行化[58].

最近几年,硬件和软件方面都发生了变化.硬件方面,基于 NUMA(non-uniform memroy access)结构的多路多核处理器是未来的主流.例如,Intel 的 Nehalem 架构和 Oracle 的 Niagara 架构,处理器包含多个多核芯片,每个多核芯片有自己的内存控制器.这样,内存访问不再是统一访问的模式,而是分为本地内存和远程内存,本地内存直接通过 IMC(integrated memory controller)访问,速度快;远程内存通过 QPI(quickpath interconnect)访问,速度慢.另外,异构平台也是一个发展趋势,即通用多核处理器加上 GPU 等加速部件构成的异构平台,这种平台如何编程也是一个难题.软件方面,随着云计算和社交网络的兴起,涌现出一系列新兴应用.这些新兴计算领域涉及的核心算法大部分属于非数值计算范畴,包含非规则的并行性,且具有数据密集的特点.例如,社交网络、网络安全、数据挖掘、生物信息学等新兴领域的一些具体应用,经过建模后一般抽象成一个复杂网络,利用图相关的算法对这些网络进行分析.因此,任务并行编程模型如何支持这些新兴的非规则应用也将是另一个研究方向.

我们认为,任务并行编程模型还需要在以下几个方面进行深入研究:

(1) 针对 NUMA 结构的多路多核处理器,任务并行编程模型需要考虑提供合适的数据分布接口.任务并行模型认为每个核访问共享内存的速度是一致的,不存在数据分布的问题.但 NUMA 结构造成内存访问速度不一致,如果数据和计算在相同的多核芯片上,计算速度就会大为提高.因此,编程模型需要提供合适的数据分布接口,这个接口需要权衡可编程性和执行效率,即在给程序员暴露多少硬件信息和希望获得高性能之间加以权衡.

(2) 针对异构平台,任务并行编程模型需要考虑提供数据分布和通信的编程接口以及相关优化的支持.例如,任务并行机制的理念是将应用划分成大量细粒度任务并行执行,这样,在异构平台上会出现通用处理器与 GPU 之间大批的小数据量通信.而通用处理器与 GPU 之间传输数据非常慢,但每次传输小数据和大数据的时间是一样的.所以,任务调度时可以考虑将大批量细粒度任务聚合成一个大任务再交给 GPU 执行,这样则可减少通信次数,从而提高性能.

(3) 针对新兴的非规则应用,任务并行编程模型需要提供更丰富的数据管理组件提高可编程性.例如,并行图计算时,一种方法是并行处理活跃节点,而图计算中常用队列和优先队列等来管理活跃节点,因此,若能提供高效的并发队列和并发优先队列,便能提高这类应用的并行编程效率.

总之,随着多核/众核芯片的发展以及一系列新兴应用的涌现,我们认为,任务并行编程模型正步入面向新处理器结构和新兴应用的优化阶段,如何高效使用新多核/众核结构和如何支持非规则应用是将来的发展方向.

References:

[1] Jack D. The promise and perils of the coming multicore revolution and its impact. CTWatch Quarterly, 2007,3(1):1−33.

[2] Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: A view from berkley. Technical Report, UCB/EECS-2006-183, Berkeley: University of California, 2006.

[3] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005,30(3):202−210.

[4] An H, Chen GL. Parallel programming models and languages. Ruanjian Xuebao/Journal of Software, 2002,13(1):118−124 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/20020117.htm

[5] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou YL. Cilk: An efficient multithreaded runtime system. In: Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 1995. 207−216. [doi: 10.1145/209936.209958]

[6] Frigo M, Leiserson CE, Randall KH. The implementation of the Cilk-5 multithreaded language. In: Proc. of the ACM SIGPLAN'98 Conf. on Programming Language Design and Implementation (PLDI). 1998. 212−223. [doi: 10.1145/277650. 277725]

[7] Intel Inc. Intel® Cilk++ SDK programmer's guide. 2009. http://software.intel.com/en-us/articles/download-intel-cilk-sdk/

[8] Intel. Intel(R) threading building blocks 3.0 reference manul. 2010. http://www.threadingbuildingblocks.org

[9] Leijen D, Schulte W. The design of a task parallel library. In: Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2009. 227−242. [doi: 10.1145/1640089.1640106]

[10] Lea D. A Java fork/join framework. In: Proc. of the ACM 2000 Conf. on Java Grande. 2000. 36−43. [doi: 10.1145/337449.337465]

[11] Charles P, Grothoff C, Saraswat VA, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: An object-oriented approach to non-uniform cluster computing. In: Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). 2005. 519−538. [doi: 10.1145/1094811.1094852]

[12] Cavé V, Zhao JS, Shirako J, Sarkar V. Habanero-Java: The new adventures of old X10. In: Proc. of the 9th Int'l Conf. on the Principles and Practice of Programming in Java (PPPJ). 2011. [doi: 10.1145/2093157.2093165]

[13] OpenMP Application Program Interface. Version 3.0. 2008.

[14] Cavé V, Budimlic Z, Sarkar V. Comparing the usability of library vs. language approaches to task parallelism. In: Proc. of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). 2010. [doi: 10.1145/1937117.1937126]

[15] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. Journal of the ACM, 1999,46(5):720−748. [doi: 10.1145/324133.324234]

[16] Blelloch RD, Leiserson CE. Space-Efficient scheduling of multithreaded computations. In: Proc. of the 25th Annual ACM Symp. on Theory of Computing. 1993. 362−371. [doi: 10.1145/167088.167196]

[17] Guo Y, Barik R, Raman R, Sarkar V. Work-First and help-first scheduling policies for async-finish task parallelism. In: Proc. of the 2009 IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS). 2009. 1−12. [doi: 10.1109/IPDPS.2009.5161079]

[18] Burton FW, Sleep MR. Executing functional programs on a virtual tree of processors. In: Proc. of the Conf. on Functional Programming Languages and Computer Architecture. 1981. 187−194. [doi: 10.1145/800223.806778]

[19] Addison C, LaGrone J, Huang L, Chapman B. OpenMP 3.0 tasking implementation in OpenUH. In: Proc. of the Open64 Workshop in Conjunction with the Int'l Symp. on Code Generation and Optimization. 2009. http://www.capsl.udel.edu/conferences/open64/2009/

[20] Tardieu O, Wang HC, Lin HB. A work-stealing scheduler for X10's task parallelism with suspension. In: Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2012. 267−276. [doi: 10.1145/2370036.2145850]

[21] Shirako J, Sarkar V. Hierarchical phasers for scalable synchronization and reduction in dynamic parallelism. In: Proc. of the 24th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). 2010. 1−12. [doi: 10.1109/IPDPS.2010.5470414]

[22] Frigo M, Halpern P, Leiserson CE, Lewin-Berlin S. Reducers and other Cilk++ hyperobjects. In: Bender MA, ed. Proc. of the 21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). New York: ACM Press, 2009. 79−90.

[23] Leiserson CE, Schardl TB. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In: Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). 2010. 303−314. [doi: 10.1145/1810479.1810534]

[24] Agrawal K, Leiserson CE, Sukha J. Helper locks for fork-join parallel programming. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2010. 245−256. [doi: 10.1145/1837853.1693487]

[25] Jr Halstead RH. Implementation of multilisp: Lisp on a multiprocessor. In: Proc. of the Symp. on Lisp and Functional Programming. 1984. 9−17. [doi: 10.1145/800055.802017]

[26] Mohr E, Kranz DA, Jr Halstead RH. Lazy task creation: A technique for increasing the granularity of parallel programs. In: Proc. of the ’90 ACM Conf. on LISP and Functional Programming (LFP’90). New York: ACM Press, 1990. 185−197. [doi: 10.1145/91556.91631]

[27] Squillante MS, Nelson RD. Analysis of task migration in shared-memory multiprocessor scheduling. In: Proc. of the ’91 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems. 1991. 143−155. [doi: 10.1145/107971.107987]

[28] Fatourou P, Spriakis P. Efficient scheduling of strict multithreaded computations. Theory of Computing Systems Journal, 2000, 33(3):173−332. [doi: 10.1007/s002240010002]

[29] Berenbrink P, Friedetzky T, Goldberg LA. The natural work stealing algorithm is stable. In: Proc. of the IEEE Symp. on Foundations of Computer Science. 2001. 178−187. [doi: 10.1137/S0097539701399551]

[30] Squillante MS, Lazowska ED. Using processor-cache affinity information in shared-memory multiprocessor scheduling. IEEE Trans. on Parallel and Distributed Systems, 1993,4(2):131−143. [doi: 10.1109/71.207589]

[31] Acar UA, Blelloch GE, Blumofe RD. The data locality of work stealing. In: Proc. of the 12th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA). 2000. 1−12. [doi: 10.1145/341800.341801]

[32] Hamidzadeh B, Lilja DJ. Dynamic scheduling strategies for shared-memory multiprocessors. In: Proc. of the 16th Int’l Conf. on Distributed Computing. 1996. 208−215. [doi: http://dx.chinadoi.cn/10.3760/cma.j.issn.1674-2907.2010.17.004]

[33] Lo M, Dandamudi SP. Performance of hierarchical load sharing in heterogeneous distributed systems. In: Yetongnon K, ed. Proc. of the Int’l Conf. on Parallel and Distributed Computing Systems. Dijon: ISCA, 1996. 370−377.

[34] Bender MA, Rabin MO. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. In: Proc. of the Theory of Computing Systems Special Issue on SPAA. 2002. [doi: 10.1007/s00224-002-1055-5]

[35] Sanchez D, Yoo RM, Kozyrakis C. Flexible architectural support for fine-grain scheduling. In: Proc. of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2010. 311−322. [doi: 10.1145/1736020.1736055]

[36] Mohr E, Kranz DA, Jr HalsteadRH, Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Trans. on Parallel and Distributed Systems, 1991,2(3):264−280. [doi: 10.1109/71.86103]

[37] Loidl HW, Hammond K. On the granularity of divide-and-conquer parallelism. In: Turner DN, ed. Proc. of the Glasgow Workshop on Functional Programming. Ullapool: Springer-Verlag, 1995. 8−10.

[38] Duran A, Corbalán J, Ayguadé E. Evaluation of OpenMP task scheduling strategies. In: Eigenmann R, Supinski BRD, eds. Proc. of the 4th Int’l Workshop on OpenMP (IWOMP). Berlin: Springer-Verlag, 2008. 100−110.

[39] Guojing C, Sreedhar K, Sriram K, Doug L, Vijay S, Tong W. Solving large, irregular graph problems using adaptive work-stealing. In: Proc. of the 37th Int’l Conf. on Parallel Processing. 2008. 536−545. [doi: 10.1109/ICPP.2008.88]

[40] Chen SM, Gibbons PB, Kozuch M, Liaskovitis V, Ailamaki A, Blelloch GE, Falsofi B, Fix L, Hardavellas N, Mowry TC, Wilkerson C. Scheduling threads for constructive cache sharing on CMPs. In: Proc. of the 19th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA). 2007. 105−115. [doi: 10.1145/1248377.1248396]

[41] Duran A, Corbalán J, Ayguadé E. An adaptive cut-off for task parallelism. In: Proc. of the 2008 ACM/IEEE Conf. on Supercomputing. 2008. [doi: 10.1109/SC.2008.5213927]

[42] Acar UA, Charguéraud A, Rainey M. Oracle scheduling: Controlling granularity in implicitly parallel languages. In: Proc. of the 2011 ACM Int’l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2011. 499−518. [doi: 10.1145/2076021.2048106]

[43] Wang L, Cui HM, Duan YL, Lu F, Feng XB, Yew PC. An adaptive task creation strategy for work-stealing scheduling. In: Proc. of the 8th Annual IEEE/ACM Int’l Symp. on Code Generation and Optimization. 2010. [doi: 10.1145/1772954.1772992]

[44] Tzannes A, Caragea GC, Barua R. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2010. 179−190. [doi: 10.1145/1693453.1693479]

[45] Lee ITA, Boyd-Wickizer S, Huang ZY, Leiserson CE. Using memory mapping to support cactus stacks in work-stealing runtime systems. In: Proc. of the 19th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2010. 411−420. [doi: 10.1145/1854273.1854324]

[46] Zhao JS, Shirako J, Nandivada VK, Sarkar V. Reducing task creation and termination overhead in explicitly parallel programs. In: Proc. of the 19th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2010. 169−180. [doi: 10.1145/1854273.1854298]

[47] Robison A, Voss M, Kukanov A. Optimization via reflection on work stealing in TBB. In: Proc. of the IEEE Int’l Symp. on Parallel and Distributed Processing (IPDPS). 2008. 1−8. [doi: 10.1109/IPDPS.2008.4536188]

[48] Chen Q, Huang ZY, Guo MY, Zhou JY. CAB: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In: Proc. of the Int'l Conf. on Parallel Processing (ICPP). 2011. 722−732. [doi: 10.1109/ICPP.2011.32]

[49] Olivier SL, Porterfield AK, Wheeler KB, Prins JF. Scheduling task parallelism on multi-socket multicore systems. In: Proc. of the Int'l Workshop on Runtime and Operating Systems for Supercomputers (ROSS). 2011. [doi: 10.1145/1988796.1988804]

[50] Guo Y, Zhao JS, Cavé V, Sarkar V. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In: Proc. of the 24th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). 2010. 1−12. [doi: 10.1145/1693453.1693504]

[51] Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J. Scalable work stealing. In: Proc. of the ACM/IEEE Conf. on High Performance Computing. 2009. [doi: 10.1145/1654059.1654113]

[52] Saraswat VA, Kambadur P, Kodali S, Grove D, Krishnamoorthy S. Lifeline-Based global load balancing. In: Proc. of the 16th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2011). 2011. 201−212. [doi: 10.1145/1941553.1941582]

[53] Cao YJ, Qian DP, Wu WG, Dong XS. Adaptive scheduling algorithm based on dynamic core-resource partitions for many-core processor systems. Ruanjian Xuebao/Journal of Software, 2012,23(2):240−252 (in Chinese with English abstract). http://www.jos.org.cn/ 1000-9825/4141.htm [doi: 10.3724/SP.J.1001.2012.04141]

[54] Long GP, Zhang JC, Fan DR. Architectural support and evaluation of Cilk language on many-core architectures. Chinese Journal of Computers, 2008,31(11):1975−1985 (in Chinese with English abstract). [doi: http://dx.chinadoi.cn/10.3321/j.issn:0254-4164.2008.11.012]

[55] Li Z, Certner O, Duato J, Temam O. Scalable hardware support for conditional parallelization. In: Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2010. [doi: 10.1145/1854273.1854297]

[56] Michael MM, Vechev MT, Saraswat VA. Idempotent work stealing. In: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2009. 45−54. [doi: 10.1145/1594835.1504186]

[57] Chase D, Lev Y. Dynamic circular work-stealing d-e-que. In: Proc. of the 17th Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). 2005. 21−28. [doi: 10.1145/1073970.1073974]

[58] Chen L, Huo W, Long XJ, Tang SL. Parallel programming languages on multi-core and many-core architectures. Information Technology Letter, 2012,10(1):23−40 (in Chinese with English abstract).

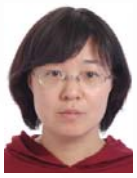
附中文参考文献:

[4] 安虹,陈国良.并行程序设计模型和语言.软件学报,2002,13(1):118−124. http://www.jos.org.cn/1000-9825/20020117.htm

[53] 曹仰杰,钱德沛,伍卫国,董小社.众核处理器系统核资源动态分组的自适应调度算法.软件学报,2012,23(2):240−252. http://www.jos.org.cn/1000-9825/4141.htm [doi: 10.3724/SP.J.1001.2012.04141]

[54] 龙国平,张军超,范东睿.众核体系结构对 Cilk 语言的硬件支持及评测研究.计算机学报,2008,31(11):1975−1985. [doi: http://dx.chinadoi.cn/10.3321/j.issn:0254-4164.2008.11.012]

[58] 陈莉,霍伟,卢兴敬,唐生林.多核/众核系统上的并行编程语言.信息技术快报,2012,10(1):23−40.



王蕾(1976－),女,江苏铜山人,博士,助理研究员,主要研究领域为并行计算,编译系统和相关工具.
E-mail: wlei@ict.ac.cn

陈莉(1970－),女,博士,副研究员,主要研究领域为并行程序设计语言和编译器,并行化编译和工具,并行程序检错.
E-mail: lchen@ict.ac.cn

崔慧敏(1979－),女,博士,副研究员,主要研究领域为并行编译,并行编程.
E-mail: cuihm@ict.ac.cn

冯晓兵(1969－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为先进编译技术及相关工具环境.
E-mail: fxb@ict.ac.cn