OpenMP 源程序级同步段负载监测方法与均衡策略

李建江,舒继武,陈永健,王鼎兴
(清华大学计算机系,北京 100084)

摘 要: 在详细阐述以同步段为最小分析单位对 OpenMP 程序进行负载监测与均衡的重要性之后,本文提出了源程序级同步段负载监测方法与均衡策略. 其中源程序级同步段负载监测方法以隐含同步的显性化为基础,具有实现简单和易于确定有效监测区域的优点. 在获得负载分布信息之后,通过同步段性能评价、筛选、负载扫描与调整实现同步段的负载均衡,这是本文与现有 OpenMP 性能工具不同的地方. 在负载均衡的过程中,根据加权剩余并行效率筛选出需要负载调整的同步段并对其负载进行扫描,最终确定出适合这些同步段的负载调度策略. 实际测试结果表明本文提出的同步段负载监测方法与均衡策略是可行的.

关键词: OpenMP; 同步段; 负载监测; 负载均衡
中图分类号: TP311 **文献标识码**: A **文章编号**: 0372-2112 (2005) 05-0852-05

A Load Monitoring Method and Balancing Strategy at Source Level of OpenMP Programs for Synchronization Segments

LI Jian-jiang, SHU Ji-wu, CHEN Yong-jian, WANG Ding-xing
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

Abstract: After expounding the importance of regarding synchronization segments as minimal units during load monitoring and balancing of OpenMP programs, this paper proposes a load monitoring method and balancing strategy at source level of OpenMP programs for synchronization segments. In this paper, the load monitoring method for synchronization segments is based on unfolding hidden synchronization operations, which is easy to implement and to determine efficient monitoring regions. After obtaining information about load distribution, load balancing is implemented through performance evaluation, choosing, load scanning and adjustment for synchronization segments, which is different from existent OpenMP performance tools. In this process, the load balancing strategy is able to choose those synchronization segments needing load adjustment according to weighted residual efficiency, to scan their load distribution under various scheduling schemes and ultimately to determine proper load scheduling strategies for them. The actual test results indicate that the load monitoring method and balancing strategy for synchronization segments is feasible.

Key words: OpenMP; synchronization segment; load monitoring; load balancing

1 引言

一般来说,实际应用中过多的 fork/join、同步操作以及负载分配的不均衡、不恰当的存储访问模式等都可能使得 OpenMP 程序的性能退化[1]. 其中,不规则的代码[2]或不恰当的调度策略可能导致负载分配的不均衡. 为了改善因此而退化的并行执行性能,OpenMP 程序开发人员首先需要获取各线程的负载分布. 当把含有两个或两个以上同步点的程序段(由一个或多个并行区组成)作为分析的最小单位时,可能掩盖各线程之间负载的不均衡(图 1). 在图 1 中任意两个相邻同步点之间各线程的负载都不均衡,但如果将包含第 i 号同步点与第 $i+4$ 号同步点之间的程序段作为最小分析单位的话,可能出现该程序段内各线程之间负载均衡的假象,因为它将第 $i+1$ 号、第 $i+2$ 号、第 $i+3$ 号同步点处各线程的等待时间一并纳入各线程的计算开销中.

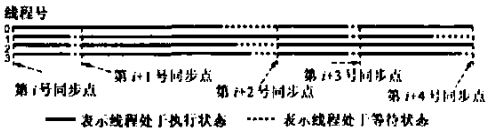

图 1 被掩盖的负载不均衡现象

由于 OpenMP 程序通过调度来实现负载的分配,所以调度是决定 OpenMP 程序中各线程之间负载是否均衡最为重要的操作之一. 如果说图 1 中出现负载均衡的假象是由于分析粒度过粗的话,那么以每次调度为基本单位能否准确地衡量负载的均衡性呢? 在图 2 中,如果以每次调度为基本单位进行分析,则可看到第 i 次调度与第 $i+1$ 次调度使得各线程的

收稿日期:2003-10-20;修回日期:2005-01-04
基金项目:国家自然科学基金(No. 60433040);国家"863"高技术研究发展计划项目基金重点项目(No. 2001AA111110, No. 2004AA111120)

负载都呈现出不均衡的状态. 但图 2 实际的情况却是,各线程在完成第 i 次调度之后,都立即进行第 $i+1$ 次调度,而不必等待别的线程,因为此时各线程之间并不存在同步操作.

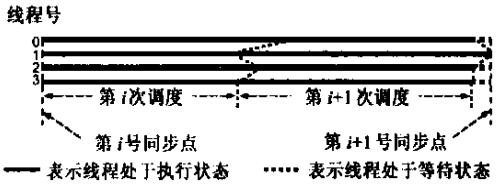

图 2　误判为负载不均衡的现象

综上所述,将 OpenMP 程序中位于两个相邻同步点之间的代码段(本文将其称为"同步段")作为分析的最小单位可以真实地反映各线程之间的负载情况[3~7].

2　源程序级同步段负载监测

2.1　同步段负载监测

目前已有一些现成的、具备同步段负载监测能力的 OpenMP 性能工具[3~7]. 其中: GuideView[3]通过对运行时库进行插桩的办法来捕获那些对并行性能至关重要的时间信息,然后通过图形界面浏览这些性能数据,最终对有助于程序性能提高和导致程序性能退化的并行结构进行快速识别. 它将执行时间分为空闲时间、串行开销、同步时间、锁开销、阻塞时间、非均衡时间、并行开销与并行时间等. Omni[4]在中间表示层面上进行代码转换,从而实现运行时性能的测量,并据此进行"profiled 调度"(一种根据当前性能结果确定下一调度块大小的调度方法). Ovaltine[5]通过对中间表示的相关段进行插桩,为 OpenMP 程序开发人员提供各种引起程序性能下降的开销(包括负载的非均衡性)分析. VTune[6]通过采样、调用图或记数器监测等方式实现 OpenMP 程序中性能问题的识别与定位. TAU[7]在源程序级实现插桩,通过对各线程状态的独立显示让 OpenMP 程序开发人员能够直观观察到何时、何处发生负载不均衡的现象. 然而,这些现有 OpenMP 性能工具(除 Omni 外)并未对监测出的负载非均衡进行相应的处理.

本文基于指导语句改写[8]的方法,在源程序级对 OpenMP 并行结构所隐含的同步操作进行显性化,并进行插桩,从而实现同步段内各线程负载的监测. 然后,在此基础上进行负载的均衡处理.

2.2　源程序级同步段负载监测

任何 OpenMP 程序,均由串行部分和并行部分(含并行结构与大量的同步操作)所组成. 当多线程并行执行某个并行结构时,只有在同步点之后、负载分配之前开始进行时间测量,并在负载执行完成之后、下一同步点到来之前结束时间测量,其间的时间差才能真正反映该并行结构中各线程的负载情况. 在同步点之前就开始时间测量会导致各线程初始化时刻的不一致;在下一同步点到来之后才结束时间测量,又必将掩盖各线程完成各自负载的真实时刻(图 3). 上述两种情况都会扭曲各线程负载的真实情况.

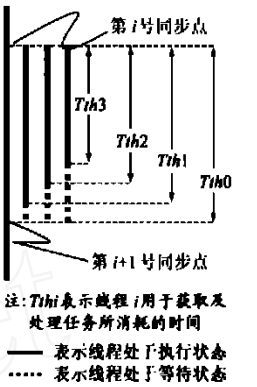

图 3　同步点前开始时间测量和同步点后结束时间测量都会扭曲各线程负载的真实情况

表 1　OpenMP 并行构的等价变换以及时间测量操作的插入

OpenMP 源代码	等价的 OpenMP 代码	插入时间测量后的 OpenMP 代码
! $omp parallel 结构化块 ! $omp end parallel		! $omp parallel 开始进行时间测量 结构化块 时间测量结束 ! $omp end parallel
! $omp do do 循环 ! $omp end do	! $omp do do 循环 ! $omp end do NOWAIT ! $omp barrier	开始进行时间测量 ! $omp do do 循环 ! $omp end do NOWAIT 时间测量结束 ! $omp barrier
! $omp workshare 结构化块 ! $omp end workshare	! $omp workshare 结构化块 ! $omp end workshare NOWAIT ! $omp barrier	开始进行时间测量 ! $omp workshare 结构化块 ! $omp end workshare NOWAIT 时间测量结束 ! $omp barrier
! $omp sections ! $omp section 结构化块 ! $omp section 结构化块 ! $omp end sections	! $omp sections ! $omp section 结构化块 ! $omp section 结构化块 ! $omp end sections NOWAIT ! $OMP barrier	开始进行时间测量 ! $omp sections ! $omp section 结构化块 ! $omp section 结构化块 ! $omp end sections NOWAIT 时间测量结束 ! $OMP barrier
! $omp single 结构化块 ! $omp end single	! $omp single 结构化块 ! $omp end single NOWAIT ! $omp barrier	开始进行时间测量 ! $omp single 结构化块 ! $omp end single NOWAIT 时间测量结束 ! $omp barrier

由于缺省情况下 OpenMP 并行结构结束处都存在一个隐含的同步操作,所以,为了在源程序级上真正实现"在同步点之后、负载分配之前开始进行时间测量,并在负载执行完毕之后、下一同步点到来之前结束时间测量",本文首先基于指导语句改写[8]的方法对需要进行时间测量的 OpenMP 并行结构进行等价变换(实现并行结构中隐含同步的显性化),然后再将时间测量操作插入到合适的位置上去(如表 1 所示).

最后,通过运行在隐含同步显性化之后插入了时间测量操作的 OpenMP 程序便可获得同步段内各线程负载的分布情况,实现对同步段内负载的监测.

本文提出的源程序级同步段负载监测方法具有如下优点:(1)同[3～6]相比,该方法实现简单.在同步操作显性化之后,利用手工、动态插桩工具或通过人机交互等方式都能轻松实现时间测量操作的插入.(2)同[7]相比,该方法易于确定有效监测区域.利用在 3.1 中定义的加权剩余并行效率可以确定哪些并行结构需要进行指导语句的等价改写与插桩处理.

3 源程序级同步段负载均衡策略

同步段负载监测能够提供同步段内各线程负载的分布细节,从而使细致的性能调整成为可能.然而,绝大多数现成的、具备同步段负载监测能力的 OpenMP 性能工具(如[3],[5～7])在获得各线程的负载分布之后,并未对负载的非均衡进行相应的处理(比如负载的调整等).为此,本文提出了如图 4 所示的源程序级同步段负载均衡策略.使用该策略在对影响 OpenMP 程序整体性能较大的那些同步段进行合理的负载调整后,程序的整体性能会得到提高.该策略由同步段性能评价、同步段筛选、同步段负载扫描以及同步段负载调整四部分组成.

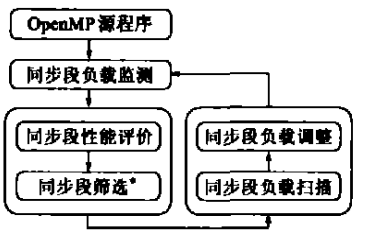

图 4 源程序级同步段负载均衡策略

3.1 同步段性能评价

假设某个 OpenMP 源程序共包含 n 个同步段(分别记为 SS_i,其中 $i \in [1, n]$),串行执行第 i 号同步段所花时间为 $ts(i)$,m 个线程并发执行第 i 号同步段所花时间为 $tp(i,m)$.本文提出如下定义用于同步段的性能评价:

*Def*1. 同步段并行加速比 $\mathrm{SpeedUp}_{of}SS(i,m)$(Speedup of Synchronization Segments):

$$\mathrm{SpeedUp}_{of}SS(i,m) = ts(i)/tp(i,m) \quad (1)$$

其中:$tp(i,m) = \max_{j=0,m-1} tp(i,j)$,$tp(i,j)$ 为第 j 号线程执行第 i 号同步段所花时间.

*Def*2. 同步段并行效率 $\mathrm{Efficiency}_{of}SS(i,m)$(Efficiency of Synchronization Segments):

$$\mathrm{Efficiency}_{of}SS(i,m) = \mathrm{SpeedUp}_{of}SS(i,m)/m \quad (2)$$

从 *Def*1 和 *Def*2 可看出,尽管同步段并行效率能够刻画某个同步段已经达到的并行化效果,却无法揭示该同步段性能改进的潜力以及该同步段性能改进后对整个程序性能的影响.对于那些并行效率很低的同步段,如果其执行时间占整个程序执行时间的比重也很小的话,那么这些同步段性能的改进对程序整体性能的影响将非常小;反之,对于并行效率较高的同步段,如果其并行执行时间占整个程序并行执行时间的比重很大的话,那么改进这些同步段有可能较大地提高程序的整体性能.因此,评判是否需要(或值得)对某个同步段进行性能改进,在考虑该同步段并行效率高低的同时还应当考虑该同步段并行执行时间占整个程序并行执行时间之比.为此,本文提出同步段加权剩余并行效率的概念(见 *Def*3).

*Def*3. 同步段加权剩余并行效率 $WRE_{of}SS$(Weighted Residual Efficiency of Synchronization Segments):

$$WRE_{of}SS(i,m) = (tp(i,m)/\sum_{i=1,n} tp(i,m)) \times (\max\mathrm{Efficiency}_{of}SS(i,m) - \mathrm{Efficiency}_{of}SS(i,m)) \quad (3)$$

其中:$tp(i,m)/\sum_{i=1,n} tp(i,m)$ 被定义为权值,表示第 i 号同步段并行执行时间占整个程序中所有同步段并行执行时间之比,这 n 个同步段的权值总和为 1;$\max\mathrm{Efficiency}_{of}SS(i,m) - \mathrm{Efficiency}_{of}SS(i,m)$ 被定义为剩余并行效率,表示第 i 号同步段已实现的并行效率与其可实现的最大并行效率之间的差距.由于测量某个同步段可实现的最大并行效率存在一定困难,而实际上 $\forall(i,m)$,$\exists eff \in (0\%, 100\%]$ 使得 $\max\mathrm{Efficiency}_{of}SS(i,m) = eff$,所以本文简单地将同步段可实现的最大并行效率近似为 100%,此时式(3)可被改写为:

$$\begin{aligned} WRE_{of}SS(i,m) &\approx (tp(i,m)/\sum_{i=1,n} tp(i,m)) \times (100\% - \mathrm{Efficiency}_{of}SS(i,m)) \\ &= (tp(i,m) - ts(i)/m)/\sum_{i=1,n} tp(i,m) \end{aligned} \quad (4)$$

式(4)既考虑了同步段已实现的并行效率与可实现的最大并行效率之间的差距,又兼顾了同步段并行执行时间占整个程序中所有同步段并行执行时间的比重.

并行加速比、并行效率、加权剩余并行效率等概念可被扩展至并行区或函数区一级.不过,随着范围的不断扩大,所描述性能细节的精度将会越来越差.

3.2 同步段筛选

由于加权剩余并行效率越大的同步段其负载调整所带来的性能收益也可能越大.所以负载调整的对象应该是加权剩余并行效率较大的那些同步段.可以采用如下方法来选取需要进行负载调整的同步段:(1)阈值法.设定某个阈值,由用户选取加权剩余并行效率高过该阈值的那些同步段;(2)排序法.对所有同步段的加权剩余并行效率进行降序排列,由用户选取该排列中位于前列的那些同步段.

在保证性能收益的前提下,同步段的筛选能较大地减少负载调整的同步段数目,从而明显降低负载调整的总开销.从图 4 来看,同步段的筛选工作可以是一个迭代的过程.当进行下一轮同步段负载调整的时候,所有上次调整过的同步段将不再被选取,只需对剩余的同步段按照上述方法进行同样的处理.当然,也可仅进行一轮同步段的筛选与调整.这是因为对程序整体性能影响较大的那些同步段绝大多数已在第一轮的筛选中被选中,其负载已得到仔细的调整.进行迭代筛选的目的仅仅在于对程序的性能作进一步的微调.

3.3 同步段负载扫描

使用不同的调度策略(含 STATIC、DYNAMIC、GUIDED 等调度方式及不同大小的调度块)和不同的线程数对选取出来的同步段进行负载调度的试验. 通过同步段负载监测获得这些同步段内各线程的负载情况(执行的迭代次数与消耗的时间)及同步段的开销. 对于试验所用调度块的大小,可使用如下序列:$\{chunk/(i\times m)\mid i=n',1,-1\}$,其中 chunk 为调度块的总大小,$m$ 为同步段内的线程数,n 为一个整数且 $n'\in[1,chunk/m]$. n 越大,负载扫描的精度就越高,相应开销也越大;反之,负载扫描的精度就越低,开销相应地减少. 同步段负载扫描建立在同步段筛选的基础上. 只对的确需要进行性能调整的同步段进行扫描,既可将扫描的总开销限定在用户可容忍的范围内,又可对选中的同步段进行高精度的扫描.

3.4 同步段负载调整

根据同步段负载扫描结果,可以找到一种合理的调度策略,使得该同步段的开销最小. 在对所有被筛选出来的同步段进行了负载调整之后,其并行效率的提高将使 OpenMP 程序的整体性能得到较大的改善.

4 实例分析

4.1 测试环境与测试对象

本文进行的所有实际测试均基于一台 4cpu 的 Itanium2,所使用的 OpenMP 编译器为 Intel 的 *ecc*7.0 与 *efc*7.0,编译优化级别为 *O*3,OpenMP 程序并行执行所使用的线程数目均为 4. 测试对象为 NAS NPB2.3 中的 BT benchmark(*C* 版本),测试规模包括 *S*、*W* 及 *A* 三种. NAS NPB2.3 是由从航空物理学应用衍生而来的五个计算流体力学核心应用(*FT*, *MG*, *CG*, *EP* 和 *IS*)与三个计算流体学模拟应用(*BT*, *SP* 和 *LU*)所组成. 目前,NAS NPB2.3 已成为衡量计算机性能的一个标准测试集,其中的 BT benchmark 中重要的函数包括:initialize、lhsinit、exact_rhs、compute_rhs、lhsx、x_solve_cell、x_backsubstitute、lhsy、y_solve_cell、y_backsubstitute、lhsz、z_solve_cell 与 z_backsubstitute.

4.2 负载监测与负载均衡

对于 W 规模,测出 BT 中各函数区的并行效率(图 5)与对应的加权剩余并行效率(图 6). 从图 5 可以看出并行效率最低的函数区是 lhsinit,只有 26.529 %. 但由于其并行执行时间只占整个程序中所有函数并行执行时间的 0.011 %,所以如果按函数区(或同步段)并行效率的高低来决定是否应当进行性能调整将可能导致如下问题:抓住了一些对程序整体性能提高微不足道的部分,而丧失那些对程序整体性能影响较大部分的调整. 比较合理的一个评判依据是不同函数区(或同步段)的加权剩余并行效率(式(4)、图 6). 从图 6 可知,函数区 lhsy 的加权剩余并行效率最大,所以应当首先对该函数区中的同步段进行性能调整.

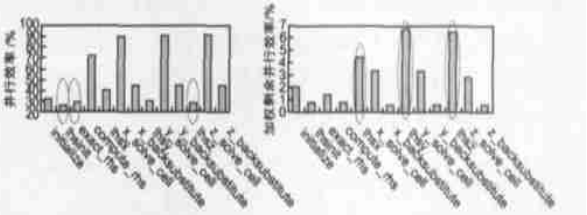

图 5 不同函数区的并行效率　　图 6 不同函数区的加权剩余并行效率

图 7、图 8 分别展示的是在不同调度策略下 lhsy 函数中第一号同步段(注:进行过简单的循环变换)内四个线程的负载分布情况及总开销的变化情况. 从图 7 可以看到当线程数为 4 时,采用 DYNAMIC 调度方式、调度块为 chunk/4 的时候,虽然四个线程的平均负载不是最小,但它们的均衡性却是最好. 图 8 显示当采用该调度策略时该同步段的总开销最小. 对 lhsy 函数中第一号同步段进行负载均衡最终使得该同步段的性能提高了约 50 %.

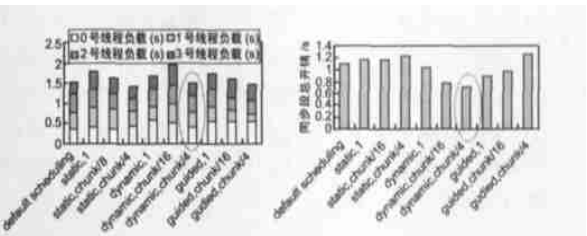

图 7 不同调度策略下 lhsy函数中第一号同步段内各线程的负载分布情况　　图 8 不同调度策略下 lhsy 函数中第一号同步段总开销的变化情况

虽然 z_solve_cell 函数区的并行效率最高,但是其加权剩

表 2 调整前后同步段内各线程负载的分布情况

运行规模		各线程的负载(1)(s)	总负载(s)	平均负载(s)	负载均方差(s)	同步段开销(2)(s)	同步隐性化后开销(3)(s)	性能改进(%)
S	负载调整之前	0.278 0.374 0.398 0.138	1.187	0.297	0.205	0.870	2.750	16.32(1) 3.51(2) 1.90(3)
	负载调整之后	0.254 0.235 0.333 0.296	1.119	0.280	0.076	0.840	2.697	
W	负载调整之前	5.430 5.416 5.446 3.569	19.862	4.965	1.612	5.460	5.557	7.30(1) 7.27(2) 5.85(3)
	负载调整之后	5.012 5.046 5.049 5.007	20.114	5.029	0.039	5.063	5.232	
A	负载调整之前	116.837 117.319 116.830 102.395	453.380	113.345	12.650	117.445	118.138	-0.28(1) -0.27(2) -0.10(3)
	负载调整之后	117.114 117.177 117.183 117.646	469.120	117.280	0.426	117.768	118.259	

其中:(1)、(2)、(3)分别指对同步段内负载、同步段开销、同步操作隐性化后开销的性能改进情况

余并行效率仍然位列所有函数区加权剩余并行效率中的第六位.为了说明对并行效率已经很高但其对应的加权剩余并行效率仍然较大的同步段进行负载调整的必要性,本文就 *S*、*W* 及 *A* 规模又分别对 z_solve_cell 函数区中第二号同步段进行了性能分析与调整(注:进行过简单的循环变换),实际测试结果如表 2 和图 9 所示.

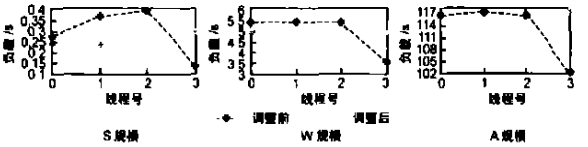

图 9 调整前后同步段内各线程负载分布情况

在对 z_solve_cell 进行负载调整后,通过分析得知每个线程处理的负载量都比较均衡.最终结果表明在规模为 *S* 和 *W* 时,该同步段的并行性能得到了不同程序的提高.如果仅就该同步段的调度与负载执行(不含同步开销)情况来看,*S* 规模时性能的提高达到 16.32 %,*W* 规模时性能提高了 7.30 %.而当计算规模为 *A* 时,性能略有下降,下降幅度为 0.28 %,这是因为当计算规模为 *A* 且四个线程并发执行该同步段时,各个线程的负载已经较为均衡,在进行负载均衡后因均衡负载而引入的开销超过了其获得的性能收益,导致性能有小的下降.这也说明负载分布、负载调整效果与计算规模(即程序输入)有着密切的关系,这是因为不同计算规模对应的负载非均衡性可能有所不同.

5 结论

在详细阐述以同步段为最小分析单位进行 OpenMP 程序负载监测与均衡的重要性之后,本文提出了源程序级同步段负载监测方法与均衡策略.其中源程序级同步段负载监测方法基于隐含同步的显性化,同[3～6]相比它的实现更简单,同[7]相比它具有易于确定有效监测区域的优点.在此基础上提出的以同步段性能评价、筛选、负载扫描与调整为核心的同步段负载均衡策略是本文与现有 OpenMP 性能工具所不同的地方,该策略具有如下优点:(1)根据加权剩余并行效率能够筛选出负载调整后对整个 OpenMP 程序性能改进影响程度较高的那些同步段,这减少了需要进行负载监测、扫描与均衡的同步段个数,从而能够将开销控制在用户可接受的范围内;(2)能够对筛选出的同步段进行精细的负载扫描.通过精细的负载扫描,可以清楚地观察到不同调度方案对同步段性能的影响情况,并能够据此确定出这些同步段各自所适用的调度策略.实际的测试结果表明本文提出的同步段负载监测方法与均衡策略是可行的.当然,负载均衡并不能保证某个并行结构或整个程序并行执行所消耗的时间最少.这是因为负载均衡所引入的额外开销(如:调度次数的增加)或所导致的性能退化(如:局部性变差)有可能抵消甚至超过负载均衡带来的收益(即各线程之间同步开销的减少).所以,负载均衡仅仅是一种重要的性能改进手段,只有在负载监测工具的帮助下,进行恰当的负载均衡处理才可能达到真正改善 OpenMP 程序并行执行性能的目的.

以同步段为单位进行负载监测与均衡的另外一个明显优势在于该过程易于并行实现,从而可能加快负载的监测与均衡.当然,为了实现负载监测与均衡的并行化,必将因读写相关信息而引入额外开销,所以对计算量不大,但数据量较大的应用来说,负载监测与均衡的并行化可能得不偿失.但对于拥有足够计算量的应用来说,并行化的优势将会比较明显.同时,将不同同步段的负载监测与均衡进行独立处理,还能避免处理那些与负载监测与均衡无关的代码,从而减少开销.至于如何高效实现同步段负载监测与均衡的并行化以及监测与均衡负载的额外开销分析是我们将要开展的研究工作.

参考文献:

[1] Marc Gonzalez, Albert Serra, et al. Applying interposition techniques for performance analysis of OpenMP parallel applications[A]. 14th International Parallel and Distributed Processing Symposium[C]. Cancun, Mexico: IPDPS, 2000. 235 - 240.

[2] Dixie Hisley, et al. Porting and performance evaluation of irregular codes using openmp[J]. Concurrency - Practice and Experience, 2000, (12): 1241 - 1259.

[3] Intel. GuideView[EB/OL]. http://www.intel.com/software/products/kappro/perfvis.htm.

[4] Yoshiaki Sakae, et al. Preliminary evaluation of dynamic load balancing using loop re-partitioning on Omni/SCASH[A]. 3st International Symposium on Cluster Computing and the Grid(CCGrid 2003)[C]. Tokyo, Japan, 2003. 463 - 470.

[5] M K Bane, G D Riley. Automatic overheads profiler for openMP codes[A]. Second European Workshop on OpenMP(EWOMP 2000)[C]. Edinburgh, Scotland, 2000.

[6] Intel. VTune analyzer[EB/OL]. http://www.intel.com/software/products/vtune/.

[7] University of Oregon. TAU[EB/OL]. http://www.cs.uoregon.edu/research/paracomp/proj/tau/.

[8] Bernd Mohr, Allen D. Malony, et al. Towards a performance tool interface for OpenMP: An approach based on directive rewriting[A]. 3st European Workshop on OpenMP(EWOMP 2001)[C]. Barcelona, Spain, 2001.

作者简介:

李建江 男,1971 年生于四川广安,博士生,研究方向:并行编译,多线程技术.E-mail:jj-lioo@mails.tsinghua.edu.cn.

舒继武 男,1968 年生于湖北孝感,博士,副教授,研究方向:大规模科学与工程计算中并行算法,并行处理技术及并行应用软件.