面向DSWP并行的OpenMP任务调度机制的扩展与实现

刘晓娴 赵荣彩 丁 锐
(解放军信息工程大学 郑州 450002)

摘 要 多核处理器能够提升多线程程序的性能,但早已存在的诸多单线程程序无法从中获益,程序员也习惯于编写单线程程序。自动并行化技术是将单线程程序移植到多核上的重要手段,但是当循环中存在无法确定的数据依赖或复杂的控制流时,传统的自动并行化技术无法取得良好效果。Ottoni等人针对传统自动并行失败的循环提出了Decoupled Software Pipelining(DSWP)算法用以实现指令级的细粒度并行,但其需要对处理器体系结构的深入了解以及对核间通信队列和专用指令的硬件支持,并行性能和应用广泛性受到限制。基于OpenMP应用编程接口实现的DSWP并行不依赖于硬件上对核间通信队列和专用指令的支持,且不受平台的限制,但现有的OpenMP任务调度机制无法满足DSWP并行中对任务调度的需求。对现有的OpenMP任务调度机制进行扩展,增加了任务与线程绑定的属性,保证了基于OpenMP的DSWP并行程序的正确执行。在GCC的OpenMP运行库libgomp中扩展了任务绑定属性子句的功能,扩展后的GCC作为OpenMP DSWP程序的基础编译器,为自动并行提供支持。通过对基准测试集NPB3.3.1的测试表明,传统自动并行失败的循环,经OpenMP DSWP自动并行后在双核处理器上平均加速比达到1.23以上;使用添加了OpenMP DSWP算法的Open64编译器生成的并行程序,与仅使用传统自动并行方法的Intel编译器和Open64编译器所得程序相比,平均加速比分别高出22%和26%。

关键词 自动并行化,OpenMP,DSWP,任务调度机制,GCC
中图法分类号 TP314 **文献标识码** A

Extension to OpenMP Task Scheduling Mechanism for DSWP Parallelization and its Implementation

LIU Xiao-xian ZHAO Rong-cai DING Rui
(PLA Information Engineering University, Zhengzhou 450002, China)

Abstract While multicore processors increase throughput for multi-programmed and multithreaded codes, many important applications are single threaded and thus are not benefited. Automatic parallelization techniques play an important role in migrating singe threaded applications to multicore platform. Unfortunately, the prevalence of control flow, recursive data structures, and general pointer accesses in ordinary programs renders the existing techniques unsuitable. Ottoni et al. proposed an automatic parallelization algorithm called Decoupled Software Pipelining (DSWP) to exploit fine-grained pipeline parallelism at the instruction level. But it requires knowledge of micro-architectural properties and hardware support of a communication channel and two special instructions. The improved DSWP algorithm based on OpenMP increases the parallel granularity and does not rely on hardware support any more, but the existing OpenMP task scheduling mechanism cannot satisfy the need of DSWP. A new binding clause for the task construct in OpenMP was proposed to extend the task scheduling mechanism. It guarantees the correctness of the OpenMP DSWP parallelization. The new clause is implemented in the GCC runtime library libgomp, which provides support for the compilation of OpenMP DSWP programs. The experimental results show that loops failed to be parallelized by existing techniques can be parallelized by the improved automatic parallelization algorithm and gain significant performance improvement on dual-core CPU. The average performance speedup is up to 1.23. Compared with Intel and Open64 compilers, the compiler with the improved algorithm can increase execution efficiency evidently and the average speedup of the OpenMP DSWP programs generated by it increases more than 22% and 26%.

Keywords Automatic parallelization, OpenMP, Decoupled software pipelining, Task scheduling mechanism, GCC

1 引言

随着主频的提高,处理器功耗以指数速度急剧上升,使得以提高主频来提升处理器性能的方法不再有效[1]。为达到更高性能,硬件开发人员设计实现了多核处理器,在主频不变的情况下,功耗随处理器上核数目的增加线性增长,避免了功耗

到稿日期:2012-11-12 返修日期:2013-02-02 本文受国家"核高基"重大专项(2009ZX01036-001-001-2)资助。
刘晓娴(1985—),女,博士生,主要研究方向为先进编译技术,E-mail:xiaoxian0321@gmail.com;赵荣彩(1957—),男,教授,博士生导师,主要研究方向为高性能计算与先进编译技术;丁 锐(1984—),男,博士生,主要研究方向为先进编译技术。

过大的问题。多核处理器已成为处理器体系结构发展的一个主要方向[2]。尽管多核处理器能够提升多线程程序的性能，但早已存在的诸多单线程程序无法从中获益，程序员也习惯于编写单线程程序。自动并行化技术是将单线程程序移植到多核上的重要手段，通过对单线程程序中蕴含的并行性的分析与发掘，采用程序变换技术自动生成适合多核处理器运行的多线程程序。循环往往占据程序的大部分执行时间，是自动并行化的主要对象[3]。近年来针对循环的自动并行化展开了大量的研究工作，相应技术也发展得较为成熟，迭代次数确定、数据结构规整、数组访问下标使循环索引变量仿射函数的规则循环能够取得良好效果[4]。如开源编译器系统Open64[5]能够通过循环交换、标量扩展和循环分布等程序变换技术得到可并行循环，实现单线程程序中循环的自动并行化；Intel公司发布的商业编译器[6]也提供了对串行程序中循环自动并行化的功能。

然而，对包含复杂控制流、递归数据结构和多重指针访问的一般循环，传统的自动并行化技术无法取得良好的效果[7]，如Open64与Intel编译器在对基准测试集NPB中代码量大、依赖关系和控制流复杂的热点循环实施自动并行时常常失效。这种情况下，传统自动并行化技术无法取得良好效果的原因主要有两方面：一是循环中存在无法确定的数据依赖，这通常由递归数据结构和指针访问引起；二是循环中存在分支结构与循环相互嵌套等复杂控制流带来的控制依赖。

为实现一般循环的自动并行化，Ottoni等人提出了一种指令级的自动任务并行算法Decoupled Software Pipelining (DSWP)[7]。该算法将循环的指令级中间表示划分为不同部分，对应于不同的任务，将各任务分配给不同线程，通过各线程上代码的并行执行来实现任务的流水并行。该方法不要求不同线程执行的任务之间无依赖关系，包括数据依赖和控制依赖。因此，传统方法中限制循环并行的两方面因素对DSWP算法没有影响。但该算法在IMPACT编译器[8]的指令级实现细粒度的任务并行，不仅需要对处理器体系结构的深入了解，而且要求对核间通信队列和produce、consume两条专用指令提供硬件支持，这些限制了DSWP的并行性能和应用广泛性。

针对DSWP算法受到的限制，本文所属课题对算法进行改进，提出了基于OpenMP[9,10]的DSWP自动任务并行算法（简称OpenMP DSWP算法）。一方面，以基本块而非指令作为构建程序依赖图的基本单位，增大了并行的粒度；在Open64编译器的循环嵌套优化遍（Loop Nest Optimization，LNO）中实现该算法。LNO遍使用高级（high）WHIRL[11]中间表示，不包含与处理器结构相关的信息，使得算法不依赖于具体的处理器体系结构。另一方面，使用OpenMP应用编程接口实现线程之间的任务分配和数据传递，不再依赖硬件上对核间通信队列和专用指令的支持；另外，OpenMP的通用性和可移植性使得该算法不受平台的限制，可广泛应用。

OpenMP的天然并行性非常适合多核处理器的并行环境，并从3.0版规范开始提供了对任务并行的支持。但是，现有的OpenMP任务调度机制中不提供任务与线程的绑定，无法满足DSWP并行中对任务调度的需求。为解决这个问题，本文对现有的OpenMP任务调度机制进行扩展，为任务增加了与线程绑定的属性，保证了基于OpenMP的DSWP程序的正确并行。此外，本文在使用广泛、支持OpenMP任务机制的开源编译器GCC的OpenMP运行库libgomp中扩展了任务绑定属性子句的功能，扩展后的GCC作为本文所属课题中自动生成的OpenMP DSWP任务并行程序的基础编译器，为自动并行提供支持。

通过对基准测试集NPB3.3.1的测试表明，传统自动并行失败的循环，经OpenMP DSWP自动并行后在双核处理器上平均加速比达到1.23以上，并行效果明显；使用添加了OpenMP DSWP算法的Open64编译器生成的并行程序，与仅使用传统自动并行方法的Intel编译器和Open64编译器所得程序相比，性能有明显提升，平均加速比分别高出22%和26%。

2 DSWP并行

本节介绍Ottoni等人提出的DSWP并行思想并分析DSWP并行对OpenMP任务调度机制提出的要求。

2.1 DSWP并行思想

传统的循环并行方式有DOALL和DOACROSS两种[12]。DOALL并行要求循环的各次迭代之间无依赖，可以按照任何顺序调度执行。DOACROSS并行主要面向包含循环携带依赖、无法进行DOALL并行的循环，将循环的各次迭代分配给不同的线程，采用线程间流水执行来获得并行性，迭代间的依赖通过某种方式的同步得到维持。

DSWP与DOACROSS实现的都是流水并行，但两者之间有明显的区别，下面以图1所示的链表遍历循环进行说明。如图1(a)中代码所示，循环的每次迭代都要使用上一次迭代取得的ptr指针的值，将链表中一个结点的值加1。该循环对应的依赖图如图1(b)所示，其中“LD”表示指针读取操作，X表示循环体。使用DOACROSS方式在双核处理器上并行该循环，循环迭代被交替地分配给机器的两个核，当前循环的循环体执行与下一循环的指针访问并发执行，如图1(c)所示。但相邻迭代之间数据依赖带来的同步开销可能完全抵消甚至超过并行的收益。

DSWP并行时不再以迭代为单位分配循环，而是对循环的代码进行划分，划分所得的第一部分（指针访问）分配给0号核执行，第二部分（循环体）分配给1号核执行。按照这样的方式并行，指针访问引起的迭代间依赖并不随代码的划分成为核之间的依赖，而是保留在核的内部，如图1(d)所示。

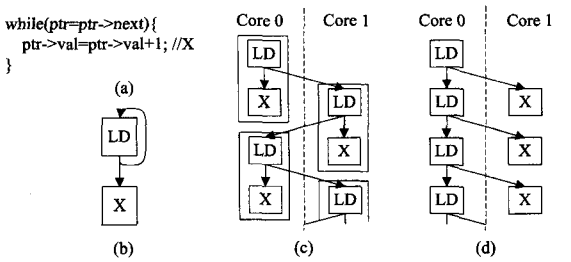

图1 链表遍历循环的DOACROSS和DSWP执行示例

根据上面的例子可知，DSWP是这样一种并行方式：将一段反复执行的代码（循环）划分为多个部分，对应于不同的任

务，各个任务之间只有单向的依赖（包括数据依赖和控制依赖）关系，不存在依赖环，然后将各任务分配给不同线程，随着代码的反复执行，形成线程之间任务的流水并行。

2.2 基于 OpenMP 的 DSWP 并行

本文所属课题以双核处理器为目标平台，因而下面以两线程的 DSWP 并行为例进行说明。循环在并行时被划分为两个部分，对应于主线程和从线程执行的任务。选择恰当的 OpenMP 构造进行主、从线程的任务分配是实现 DSWP 并行的关键。OpenMP API 中提供了 4 种 worksharing 构造、1 个 master 构造和 1 个 tasking 构造，用以实现不同方式的任务分配。

Loop 构造实现循环的并行，但是以循环迭代作为分配的基本单位，与 DSWP 并行的任务分配方式有异；sections 构造面向非循环代码段；single 构造用于指定线程组中单个线程完成的工作，适合 DSWP 并行，可作为备选的构造；workshare 只适合 Fortran 特定结构的并行，不适合 DSWP 并行；master 构造指定线程组中主线程要完成的工作，且入口和出口处皆无 barrier，从线程直接跳过该段代码而无需等待，相比 single，更加适合指定 DSWP 并行中主线程的任务。Task 构造用于标识一个独立的工作单元，能够表示不规则的并行和实现任务的动态生成。与其他构造相比，task 构造更加灵活，适用于 DSWP 并行时从线程的任务分配。

经过上面的分析和比较，在 OpenMP DSWP 并行时，使用 master 构造指定主线程的任务，task 构造指定从线程的任务，图 2 所示为图 1(a)中循环对应的 OpenMP DSWP 并行代码。

```
#pragma omp parallel threads(2)
{
#pragma omp master
  {
while(ptr=ptr->next)
    {
#pragma omp task binding(1) firstprivate(ptr)
      {
    ptr->val=ptr->val+1;
      }
    }
  }
}
```

图 2 并行后的图 1(a)所示循环

如图 2 代码所示，程序进入 parallel 并行区域后，依照 threads(2)子句启动一个从线程，与主线程一起组成两线程的线程组，master 构造指示主线程中执行读取指针的操作，task 构造生成一个显式的任务，执行循环体。根据 DSWP 的并行方式，task 构造生成的任务必须由从线程执行。但现有的 OpenMP 任务调度机制无法实现这样的功能，需要进行扩展。

3 对 OpenMP 任务调度机制的扩展与实现

本节首先介绍 OpenMP 中现有的任务调度机制，然后给出本文扩展的任务绑定属性和在 GCC 中的实现算法。

3.1 OpenMP 任务调度机制

OpenMP 中的任务有两种类型，一种是隐式的任务（implicit task），是线程遇到 parallel 构造之后生成的与线程数目相等的工作单元，对应于 parallel 构造的区域，每个线程分得一个这样的任务。这类任务不是独立的工作单元，不能动态调度，只是对引入任务机制前 parallel 区域的扩展。另一种是显式的任务（explicit task），其在线程遇到 task 构造时生成，能够实现动态、异步的并行，是真正的任务。本小节介绍的是第二种任务的调度机制。

对任务调度点（task scheduling point）的处理是 OpenMP 任务调度机制中的关键，OpenMP 中定义了如下任务调度点的位置：

- 紧跟在显式任务生成后
- 任务区域的最后一条指令后
- taskwait 区域
- 隐式或显式的 barrier 区域

此外，可以在无约束（untied）任务内部的任何位置插入任务调度点。在任务调度点进行任务调度时，要满足一组称为任务调度限制（Task Scheduling Constraints）的约束条件：

- 包含 if 子句，并且 if 子句中表达式的值为 false 的显式任务，在生成后必须立即执行；
- 如果要调度新的受约束（tied）的任务（即没有开始执行的受约束的任务），那么与该线程相关的受约束的任务集合是空集，或者新的受约束的任务是任务集合中每一个任务的子孙任务。

当线程遇到任务调度点时，在满足任务调度限制的基础上可以选择下面操作中的一种执行：

- 开始执行一个受约束的任务
- 恢复（resume）任意被挂起（suspended）的和其相关的受约束的任务
- 开始执行一个无约束的任务
- 恢复任意被挂起的无约束的任务

如果线程遇到任务调度点时存在多个可选择的操作，OpenMP 规范中没有明确说明这些选择的优先级，由 OpenMP 的具体实现决定。

对 OpenMP 现有的任务调度机制进行分析，可发现执行任务的线程是不确定的。一个新生成的任务，如果不是非延迟的任务，生成之后被放入任务池中；空闲的线程可以从任务池中选择任意与其线程组绑定的任务执行。这种机制虽然能够较好地实现负载平衡，但同时也导致了任务与线程之间关系的不确定性，使得程序员在使用 task 构造定义任务时不能确定执行该任务的线程，尤其在希望能够明确指定执行任务的线程时，这种机制失效。虽然 OpenMP 中为任务提供了一种受约束与无约束属性用于将任务与执行它的线程进行绑定，但实现的是任务执行之后的绑定，在一个任务开始执行之前，仍然不能确定执行该任务的具体线程，无法满足 DSWP 任务并行的需求。

3.2 扩展的任务绑定属性

本小节对 OpenMP 的任务调度机制进行扩展，为任务增加一个绑定属性，用于将 task 构造生成的任务与某一指定线

程进行绑定，用 binding 子句实现，具体语法如下：

#pragma omp task binding(integer-expression)[clause[[,] clause] …] new-line

structured-block

该子句的功能是将生成的任务与括号中整数所指线程号对应的线程进行绑定。与线程绑定的任务有如下限制：

1)只能与生成它的线程所在线程组中的某个线程进行绑定；

2)不能指定为 untied；

3)一旦执行，不能被挂起，忽略任务调度点，直至执行结束；

4)不允许包含 taskwait 构造；

5)不允许包含 parallel 构造和 task 构造。

被任务绑定的线程选择任务执行时要遵循以下规则：

1)优先选择与其绑定的任务；

2)在选择与之绑定的任务时，按照先进先出的顺序；

3)当与其绑定的所有任务执行完毕后，再选择与其所在线程组绑定的任务执行。

图 3 给出了 binding 子句实现功能的一个示例。线程组中的线程从任务池中选取 1、2、3 这 3 个任务执行，执行各任务的线程和任务之间的执行顺序都是不确定的，图 3(a)给出了一种可能的执行方式；若使用 binding 子句指示 2 号线程执行任务 1、2、3，则能够实现这 3 个任务在同一线程上明确的执行顺序，如图 3(b)所示。

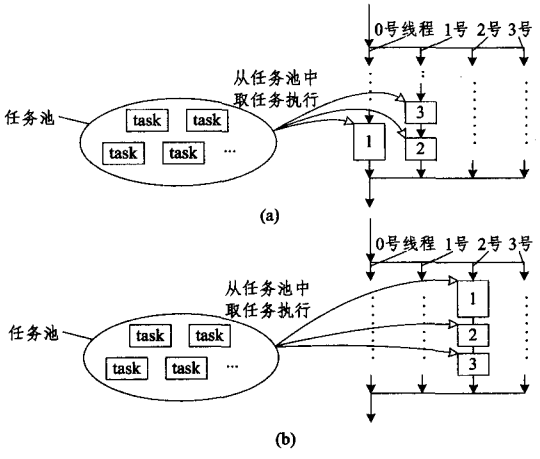

图 3 binding 子句功能示意图

任务的绑定属性不是必需的，无绑定属性的任务按 OpenMP 原本定义的方式运行。绑定属性在保持任务灵活性的同时，增强了任务与线程之间的联系，让 OpenMP 使用者在使用 task 构造时有更多的选择，也能够实现更多的功能。

图 2 中将循环体所对应的任务与编号为 1 的线程绑定，因此生成的任务都由编号为 1 的从线程执行，根据先进先出的要求，执行顺序与生成顺序一致。这样能够维持任务代码中包含的依赖关系，保证 OpenMP DSWP 并行程序运行结果的正确性。

3.3 任务绑定属性在 GCC 中的实现

3.3.1 GCC 的编译流程

在 GCC 的编译过程中，根据各阶段代码优化的不同需要，前端(Front End)至后端(Back End)分别使用 GENERIC、GIMPLE 和 RTL 3 种不同的中间表示，其编译流程如图 4 所示。

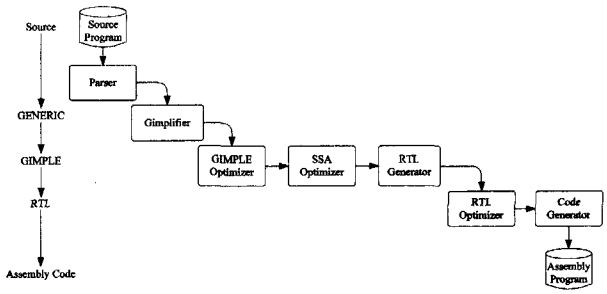

图 4 GCC 编译流程和对应的中间表示

GCC 中对 OpenMP 的支持由 4 个部分构成：语法分析器(parser)、中间表示(intermediate representation)、代码生成(code generation)和运行库(runtime library) libgomp。OpenMP 指示对应的中间表示是对 GCC 中 GENERIC 和 GIMPLE 中间表示的扩展，作为调用运行库的接口。

OpenMP 程序的编译过程中，不同于普通串行程序的步骤如下：

1)语法分析器识别 OpenMP 指示并验证其正确性，生成对应的 GENERIC 表示；

2)Gimplifier 分析并行区中使用的变量集合，并根据数据共享子句建立变量的映射关系；

3)Pass_lower_omp 为实现并行区中所使用数据的映射关系定义相应的数据结构，将变量的值写入 struct. omp_data_s 中，展开一些同步结构并在 OpenMP 编译指示对应区域的结尾处添加 OMP_RETURN 标记；

4)Pass_lower_cf 将编译指示对应的区域线性化，去除它们的嵌套特性，为构建控制流图做准备工作；

5)Pass_build_cfg 构建控制流图，并确保控制流图中进入并行区的边保留在其中；

6)Pass_expand_omp 在代码转换成 SSA 格式之前执行，将每个 omp parallel 对应的单入口、单出口区域标出并将其中所有的指示展开为对运行库 libgomp 的调用或是对应的 GIMPLE。

完成以上步骤之后，中间代码由 SSA Optimizer 进行下一步的优化，之后的编译过程无异于普通串行程序。

3.3.2 任务绑定属性的实现

开源编译器 GCC4.7 在其运行库 libgomp 中提供了对 OpenMP 的任务调度机制的支持。在 libgomp 的实现中，每个线程组有一个任务共享池，所有新生成的任务都放在共享池中。线程组的共享池由两个双向循环链表组合的混合链表实现。添加一个任务到共享池中或从共享池中取走一个任务，要分别修改任务的 queue 链和 child 链。queue 链按照先进先出的顺序维护共享池中的所有任务，child 链按照先进先出的顺序维护线程组中有共同父任务的任务。因此，共享池中包含一条 queue 链和多条 child 链。

对于 OpenMP 中定义的 4 类任务调度点，GCC 在实现时，对前两个任务调度点作简单处理：对于紧跟在显式任务后的调度点，只有当新生成的任务是一个必须立即执行的任务时，才将当前任务挂起，转去执行新生成的任务；一个任务结

束后，执行该任务的线程恢复之前被该任务中断的任务，继续执行。taskwait 区域和 barrier 区域是任务调度的重点。

当线程遇到 taskwait 构造时，会创建一个 taskwait 区域，查看当前任务是否有子任务，如果有，并且处于等待运行的状态，就调度其子任务运行，同时把当前任务保存在线程的栈帧中。如果子任务都在运行，则线程挂起，等待所有子任务的完成。所有子任务都完成后，发送信号通知当前任务，然后线程继续运行当前任务函数 GOMP_taskwait 实现上述功能。

当线程遇到一个隐式或显式的 barrier 时，如果当前线程是组内最后一个到达该 barrier 的线程，且总任务数不等于零，则设立标识，表示所有线程已经到达栅障。如果共享池中有任务，按照先来先服务的顺序依次从共享池中提取任务执行；当任务完成后，断开 child 链，释放其子任务的 parent 指针；如果其父任务处于挂起状态，且当前任务是最后一个完成的子任务，则通知父任务继续执行，然后释放任务空间。依次从共享池中提取任务执行上述操作，直到最后一个线程到达且所有任务已经完成。函数 gomp_barrier_handle_tasks 实现上述功能。

```
1. procedure GOMP_taskwait()
2. //功能:实现线程遇到 taskwait 时,对任务的调度
3.   if 当前任务为空 ‖ 当前线程组为空 then reture;
4.   while(1) do begin
5.     if 当前任务的子任务全部被执行 begin
6.       if 待释放任务指针 to_free 不为空
7.       then 释放 to_free 指向的任务;
8.       reture;
9.     end
10.    if 当前任务有子任务等待执行 then begin
11.      从当前任务 child 链头取下一个任务 child_task;
12.      将 child_task 的 queue 链断开;
13.    end
14.    else 将当前任务的 in_taskwait 属性置为 true;
15.    if to_free 不为空 then 释放 to_free 指向的任务;
16.    if child_tasek 不为空 then 执行 child_task;
17.    else begin
18.      将当前任务的 in_taskwait 属性值为 false;
19.      return;
20.    end
21.    if child_task 不为空 then begin
22.      将 child_task 的 child 链断开;
23.      置空 child_tasek 所有子任务的 parent 指针;
24.      to_free=child_task;
25.      child_task=NULL;
26.    end
27.  end
28. end GOMP_taskwait
```

图 5　taskwait 同步算法

```
1. procedure gomp_barrier_handle_tasks()
2. //功能:实现线程遇到 barrier 时,对任务的调度
3.   if 最后一个线程到达 && 所有任务已经完成 then reture;
4.   while(1) do begin
5.     if 与线程绑定的任务队列不为空 then 从队列上取一个任务
       child_task;
6.     else if 线程组的 queur 链不为空 then 从 queue 链上取一个任
       务 child_task;
7.     if 待释放任务指针 to_free 不为空 then 释放 to_free 指向的任
       务;
8.     if child_task 不为空 then 当前线程执行 child_task;
9.       else reture;
10.    if child_task 不为空 then begin
11.      断开 child_task 的 child 链
12.      if child_task 是其父任务最后一个完成的子任务 then 通知
         父任务;
13.      置空 child_task 的所有子任务的 parent 指针;
14.      to_free=child_task;
15.      child_task=NULL;
16.    end
17.  end
18. end gomp_barrier_handle_tasks
```

图 6　barrier 同步算法

为实现本文扩展的功能，需要增加一种表示与同一线程绑定的任务链表。GCC 实现的任务共享池中已存在两种链表，如果再增加一种，对任务共享池的维护将变得极其复杂，时间复杂度也有所增加。本文选择在任务共享池之外为线程组创建一组新的任务队列，用于管理与线程组中各线程绑定的任务。一个队列中的任务使用 queue 链相互链接，与线程组共享池中任务的 queue 链不相链接，child 链能够在新的任务队列和任务共享池的任务之间链接。增加扩展属性后任务的 barrier 同步和 taskwait 同步的算法分别如图 5 和图 6 所示。

4　实验结果与分析

基准测试集 NPB 是美国 Numerical Aerodynamic Simulation 项目开发的并行基准测试程序，用于测试并行系统的性能，本节使用 NPB3.3.1 对基于扩展调度机制实现的 OpenMP DSWP 自动并行算法进行测试。测试平台为 IBM x3650 系列的双核服务器，其中每个核的主频为 2.13GHz，内存为 4GB，使用的操作系统为 Redhat Enterprise 5，OpenMP DSWP 并行程序的基础编译器是本文扩展后的 GCC4.7。

4.1　对热点循环的测试

本节从测试集中选出一组循环，它们占各自所在测试程序总执行时间的 10%至 99%以上，在进行传统自动并行化时失败。表 1 给出了所选循环的相关信息，包括所属程序名、所在函数、占测试程序执行时间的百分比，以及传统自动并行化失败的原因。

表 1　NPB3.3.1 热点循环信息

程序名	所在函数	执行时间所占百分比(%)	传统自动并行失败的原因
BT	x_solve	26.6	存在阻碍并行的依赖
EP	EMBAR	99.5	存在阻碍并行的依赖
LU	blts	15.7	存在阻碍并行的依赖
SP	compute_rhs	10.1	存在阻碍并行的依赖
UA	convect	28.9	无法并行的循环结构

本小节对以上循环两线程 OpenMP DSWP 的并行性能

进行测试，图 7 给出了各热点循环的 S、A、C 3 个不同规模在测试平台上的并行加速比。测试结果显示，热点循环 3 个规模的平均并行加速比均达到 1.23 以上，其中，函数 x_solve、blts 和 convect 中的热点循环并行加速比均达到 1.35 以上，与传统自动并行失败、只能串行执行的情况相比，获得明显的性能提升。OpenMP DSWP 并行时任务生成和调度的开销较大，代码规模较小的循环并行后的开销可能会完全抵消甚至超过并行的收益，函数 EMBAR 和 compute_rhs 中热点循环的并行正是这种情况，如图 7 所示。另外，随着热点循环测试规模的增大，并行开销占循环执行时间的比例有所下降，并行加速比有所提升，如图 7 中所示。

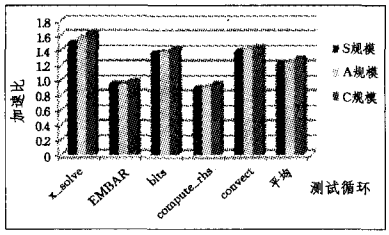

图 7　热点循环的并行加速比

4.2　对 NPB3.3.1 的测试

本小节对 NPB3.3.1 中的 BT、EP、LU、SP 和 UA 这 5 个程序进行测试，测试内容包括 3 个并行加速比，分别是 Intel 12.0 编译器的自动并行、Open64-5.0 编译器的自动并行和添加了基于扩展调度机制所实现的 OpenMP DSWP 算法后的 Open64-5.0 编译器的自动并行。添加了 OpenMP DSWP 算法的 Open64 编译器对循环实施自动并行时，先使用传统自动并行方法，对于那些传统自动并行失败的循环，再使用本文的算法。图 8 为测试程序 A 规模时在双核测试平台上的加速比。

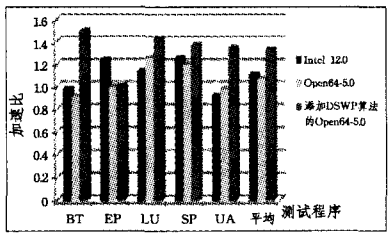

图 8　NPB3.3.1 的并行加速比

图 8 表明，添加了 OpenMP DSWP 算法的 Open64 编译器能够更有效地对循环进行并行，所得并行程序的平均加速比为 1.33，与仅使用传统自动并行方法的 Intel 编译器和 Open64 编译器相比，分别高出 22%和 26%。

结束语　传统自动并行技术对包含复杂控制流、递归数据结构和多重指针访问的一般循环无法取得良好效果；Ottoni 等人提出的 DSWP 算法实现指令级的细粒度并行，需要对处理器体系结构的深入了解以及对核间通信队列和专用指令的硬件支持，并行性能和应用广泛性受到限制。基于 OpenMP 应用编程接口实现的 DSWP 并行不依赖硬件上对核间通信队列和专用指令的支持，且不受平台的限制，可广泛应用。但现有的 OpenMP 任务调度机制无法满足 DSWP 并行中对任务调度的需求。为解决这个问题，本文对现有的 OpenMP 任务调度机制进行扩展，增加了将任务与线程绑定的属性，保证了基于 OpenMP 的 DSWP 并行程序的正确执行。文中首先介绍了 DSWP 并行思想和对 OpenMP 任务调度机制提出的要求，然后详细讨论了对 OpenMP 任务调度机制的扩展和在 GCC 中的实现。实验结果表明，基于扩展调度机制的 OpenMP DSWP 自动并行算法能够使传统自动并行失败的循环获得明显的性能提升，有效弥补了传统自动并行技术的不足。下一步工作的重点是优化 GCC 中任务调度机制的实现方法，减少任务调度带来的并行开销。

参 考 文 献

[1] Benoit A, Melhem R, Renaud-Goud P, et al. Power-aware Manhattan routing on chip multiprocessors[C]// Proceedings of 26th International Parallel and Distributed Processing Symposium. Shanghai, 2012: 189-200

[2] Jin Hao-qiang, Jespersen D, Mehrotra P, et al. High performance computing using MPI and OpenMP on multi-core parallel systems[J]. Parallel Computing, 2011, 37(9): 562-575

[3] 丁锐，赵荣彩，韩林. 基于主导值的计算和数据自动划分算法[J]. 计算机科学，2012，39(3)：290-294

[4] Allen R, Kennedy K. Optimizing compilers for modern architectures: a dependence-based approach [M]. California: Morgan Kaufmann Publisher, 2001: 63-68

[5] Lin Yu-te, Wang Shao-chung, Shih Wen-li, et al. Enable OpenCL compiler with Open64 infrastructures[C]// Proceedings of 13th IEEE International Conference on High Performance Computing and Communications. Alberta, 2011: 863-868

[6] Gerber R, Smith K B, Bik A J C, et al. The sofware optimization cookbook: high-performance recipes for IA-32 platforms(2st ed) [M]. Hillsboro: Intel Press, 2006: 13-27

[7] Ottoni G, Rangan R, Stoler A, et al. Automatic thread extraction with decoupled software pipelining[C]// Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, 2005: 105-118

[8] August D I, Connors D A, Mahlke S A, et al. Integrated predication and speculative execution in the IMPACT EPIC architecture[C]// Proceedings of the 25th International Symposium on Computer Architecture. Barcelona, 1998: 227-237

[9] 富弘毅，丁滟，宋伟，等. 一种利用并行复算实现的 OpenMP 容错机制[J]. 软件学报，2012，23(2)：411-427

[10] Thoman P, Jordan H, Pellegrini S, et al. Automatic OpenMP loop scheduling: a combined compiler and runtime approach[C]// Proceedings of 8th International Workshop on OpenMP. Rome, 2012: 88-101

[11] Ramshankar R. Open64 Compiler Developer Guide [OL]. http://developer. amd. com/tools/cpu/ open64 /Documents/ open64_compiler_developer_guide. html, 2009-12

[12] Hurson A R, Lim J T, Kavi K M, et al. Parallelization of DO-ALL and DOACROSS loops——a survey[J]. Advances in Computers, 1997, 45: 53-103