



## 一种针对线性循环结构的非线性静态调度策略

李亚朋<sup>1,2</sup>, 庞建民<sup>2,3</sup>, 徐金龙<sup>2,3</sup>, 聂凯<sup>2,3</sup>

(1. 郑州大学 中原网络安全研究院, 郑州 450001; 2. 中国人民解放军战略支援部队信息工程大学 数学工程与先进计算国家重点实验室, 郑州 450001; 3. 中国人民解放军战略支援部队信息工程大学, 郑州 450001)

**摘要:** 现有 OpenMP 调度策略通常采用动态策略处理程序中的线性循环结构, 存在负载不均衡和调度开销大的问题。提出一种针对线性递增或线性递减循环结构的非线性静态调度策略 Nonlinear\_static。将线性循环负载均匀变化参数与总负载、负载峰值、线程数相结合构建调度模型, 计算循环迭代在线程上的映射, 使迭代块大小呈非线性递增或递减趋势。将线性循环的负载平均地分配在每个线程上, 并在开源 OMPi 编译器中进行编码。在 Adjoint Convolution、Compute Pots、Matrix Multiplication、Mandelbrot Set 应用程序上进行多线程调度, 实验结果表明, 相比静态调度、动态调度、指导调度等策略, Nonlinear\_static 调度策略在处理线性循环结构时执行时间缩短了 5%~10%, 且具有无调度开销的优点。

**关键词:** OpenMP 调度策略; 负载均衡; 调度开销; 静态调度; 线性循环

开放科学(资源服务)标志码(OSID):



中文引用格式: 李亚朋, 庞建民, 徐金龙, 等. 一种针对线性循环结构的非线性静态调度策略[J]. 计算机工程, 2022, 48(1): 155-162.

英文引用格式: LI Y P, PANG J M, XU J L, et al. A nonlinear static scheduling strategy for linear loop structure[J]. Computer Engineering, 2022, 48(1): 155-162.

## A Nonlinear Static Scheduling Strategy for Linear Loop Structure

LI Yapeng<sup>1,2</sup>, PANG Jianmin<sup>2,3</sup>, XU Jinlong<sup>2,3</sup>, NIE Kai<sup>2,3</sup>

(1. Zhong Yuan Network Security Research Institute, Zhengzhou University, Zhengzhou 450001, China; 2. State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Strategic Support Force Information Engineering University, Zhengzhou 450001, China; 3. PLA Strategic Support Force Information Engineering University, Zhengzhou 450001, China)

**[Abstract]** The existing OpenMP scheduling strategies usually use a single dynamic scheduling strategy to deal with linear loops, leading to unbalanced loads and high scheduling overhead. To solve the problem, this paper proposes a nonlinear static scheduling strategy (Nonlinear\_static) for linearly increasing or linearly decreasing loops. Nonlinear\_static combines the loads of linear loops change parameter, and the total load, peak load as well as the number of threads to construct a scheduling model. The model is used to calculate the mappings of loop iterations on threads, making the value of iteration blocks increasing or decreasing nonlinearly. Nonlinear\_static strategy distributes the values of nonlinearly iterative blocks to different threads, so that each thread gets the same load, and encode them in the open-source OMPi compiler. Multi-threaded scheduling is carried out on Adjoint Convolution, Compute Pots, Matrix Multiplication, Mandelbrot Set applications. The experimental results show that compared with the static scheduling strategy, dynamic scheduling strategy and guided scheduling strategy, the execution time of Nonlinear\_static is reduced by 5%~10% for threads when dealing with linear loops. It also displays common advantages of static scheduling, such as zero scheduling cost.

**[Key words]** OpenMP scheduling strategy; load balancing; scheduling overhead; static scheduling; linear loop

DOI: 10.19678/j.issn.1000-3428.0060569

### 0 概述

2003 年以前, 单核处理器的主频在摩尔定律的影响下不断提高, 程序的执行效率主要依赖处理器的主

频。2003 年以后, 由于功耗的原因, 主频遇到技术瓶颈, 多核技术成为提升硬件性能的有效方案。多核处理器的发展, 一方面提高了处理器的性能, 另一方面将提高实际计算能力的任务转移给了软件开发人员。

基金项目: 之江实验室重大科研项目“先进工业互联网安全平台”(2018FD0ZX01)。

作者简介: 李亚朋(1993—), 男, 硕士研究生, 主研方向为高性能计算、编译优化; 庞建民, 教授、博士; 徐金龙, 讲师、博士; 聂凯, 博士研究生。

收稿日期: 2021-01-12 修回日期: 2021-03-02 E-mail: jianmin\_pang@126.com

为充分发挥多核处理器的计算潜力,并行编程模型应运而生,其中OpenMP规范<sup>[1]</sup>对现有编程语言进行了扩展,为软件开发人员提供简单、高效的编程接口,成为应用最广泛的多线程编程模型。OpenMP是一组由计算机硬件和软件供应商联合定义的应用程序接口(API)<sup>[1]</sup>,OpenMP标准规范于1997年首次发布,最新版本为V5.0,为基于共享内存并行程序的开发人员提供一种可扩展的编程模型,目前在主流的编译器中都具有相关技术支持。在OpenMP线程执行模式下,当程序开始时单一主线程串行执行,当程序执行到并行区域时派生出一组分支线程和主线程并行执行,离开并行区域后所有线程进行同步并且自动结束,最后程序只剩一个单独的主线程,继续串行执行。

针对不同的循环结构,OpenMP规范提供了多种调度策略,但都存在不同程度的调度开销和负载均衡问题,最终影响热点循环在目标机器上的执行效率。为尽可能地减少调度开销,使得各个处理器负载趋于均衡,软件开发人员需要根据循环特征在调度开销和负载均衡之间进行权衡,找到合适的调度策略。

本文提出一种针对线性循环结构的调度策略Nonlinear\_static,结合线性负载均匀变化参数与总负载、负载峰值、线程数构建Nonlinear\_static调度模型,通过该模型计算每个迭代块的大小使其呈现非线性递增或递减趋势,并在开源OMP编译器上编码实现Nonlinear\_static调度策略,从而消除调度开销并实现负载的均衡分配。

## 1 相关工作

### 1.1 循环结构

科学计算中的二八法则表明,程序中20%的代码占据80%的执行时间,这些占据大量时间的代码通常是程序中的循环结构。因此,挖掘循环的并行性是提高程序执行效率的有效方法<sup>[2]</sup>。循环分为含有依赖循环和不含依赖循环。有依赖的循环无法直接并行,因此不在本文的研究范围内。本文提到的循环均为不含依赖的循环。根据循环中负载的分布情况,常见循环分为4种不同的结构<sup>[3]</sup>,如图1所示,横坐标为循环索引变量*i*,纵坐标 $L(i)$ 表示第*i*次迭代中负载量。

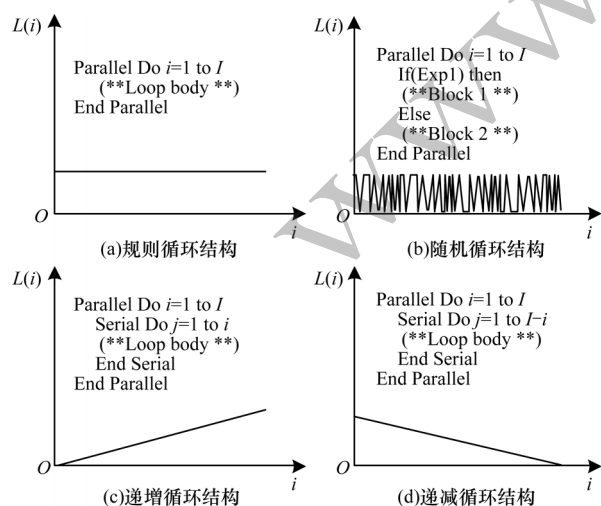


图1 4种循环结构

Fig.1 Four loop structures

从图1(a)可以看出,规则循环结构的负载 $L(i)$ 在迭代索引*i*上的分布是均匀不变的。随机循环结构的负载 $L(i)$ 在迭代索引*i*上随机分布,没有规律。递增循环结构和递减循环结构的负载 $L(i)$ 在循环索引*i*上呈线性递增和线性递减趋势。4种循环结构基本包括绝大部分的应用场景,其中随机循环结构、递增循环结构和递减循环结构称为非规则循环结构<sup>[4]</sup>,而递增循环结构和递减循环结构又称线性循环结构。

### 1.2 OpenMP循环调度策略

在共享存储体系结构中,调度开销和负载均衡是循环级并行化面临的重要问题。不同的调度策略针对特定循环的效果有好有坏,没有任何一种调度策略能够解决所有的循环调度问题<sup>[5]</sup>,关键在于找到适合目标循环特征的调度策略。

目前,OpenMP规范中提供了4种标准的调度策略,分别是静态调度(static)、动态调度(dynamic)、指导调度(guided)以及运行时调度(runtime),其中运行时调度根据环境变量动态选择前3种中的1种,环境变量由程序员提前指定。

静态调度在默认情况下将循环迭代划分为大小相等的块,每个线程分到的迭代块值 $chunksize$ (线程分到迭代块的大小)等于 $P/N$ ( $N$ 代表循环总的迭代数, $P$ 代表使用的线程数),当无法整除时会尽可能平均分配。静态调度还可以指定 $chunksize$ 值,在这种情况下,每个线程依次得到 $chunksize$ 值的迭代块,直到所有的迭代都分配完成。这种调度方法适用于规则循环结构,对于随机循环结构和线性循环结构,不同的线程负载差距较大,负载均衡性较差<sup>[6-7]</sup>,其优点是迭代在线程上的映射编译时已经确定,所以在运行时没有调度开销。

动态调度在程序运行中每个线程会申请 $chunksize$ 大小的迭代块,当某个线程执行完任务处于空闲状态但还有迭代未被执行时,它会再次请求 $chunksize$ 大小的迭代块,直到所有的循环都执行完成。在默认情况下,当不指定 $chunksize$ 时,其值为1。相较于静态调度,动态调度能够更好地适应非规则循环结构,但其会面临 $chunksize$ 值的抉择,过大的块会引起线程间的负载不均衡,过小的块则会引起频繁的线程调度,造成严重的调度开销<sup>[8]</sup>。

指导调度<sup>[9]</sup>是一种启发式的自调度方法,其迭代块值不是恒定不变的而是逐渐减小的,先申请任务线程得到的迭代块会比较大,之后逐渐减小。迭代块值是一个可选参数,迭代块的变化呈指数下降,直至下降到迭代块值保持不变,当不指定迭代块值时,其值默认为1。在OMP编译器中迭代块是按照 $ch=(ch+t->nth-1)/t->nth$ 来计算,其中 $ch$ 表示剩余迭代的值, $t->nth$ 表示线程数。指导调度在面临递增型循环结构时表现出良好的负载均衡性,但是当遇到随机型循环特别是递减型循环时,容易造成负载集

中于前面几个线程的现象。

除 OpenMP 规范中标准调度策略之外, 梯式调度 (TSS)<sup>[3]</sup> 综合了动态调度和指导调度优点, 常用于处理非规则循环。相似于动态调度, 梯式调度迭代块值的变化是线性的, 与动态调度不同的是其迭代块值下降到一定程度后将保持不变, 在一定程度上缓解了动态调度和指导调度后期调度开销大的问题。线性变化的迭代块值使得梯式调度在面对递增循环结构时相较于指导调度具有更优的负载均衡性。

Factoring 调度 (FSS)<sup>[10]</sup> 采用分段的方法处理循环迭代。在同一个段内, 每个线程分配的迭代块值是相同的, 而段与段之间是线性减小的。这种方法在面对非规则循环时相较于梯式调度负载均衡性更优, 但实现复杂且开销更大。从整体上分析, Factoring 调度的迭代块值同样是不断减小的, 所以在处理递减型循环时, 随着负载的不断减小, 分配到的迭代块也是越来越小, 造成严重的负载失衡。

针对指导调度和 Factoring 调度不适用于递减型循环的问题, 文献[12]借鉴  $\alpha$  调度策略<sup>[11]</sup>, 提出 new\_guided 调度策略。在循环的前半段使用静态调度, 后半段使用指导调度, 保证调度初期迭代块值不至于过大, 在一定程度上缓解了负载在最初几个线程过于集中的问题。在面对递减型循环时, new\_guided 调度相较于指导调度具有更优的均衡效果, 但在本质上并没有解决现有调度策略迭代块递减的弊端。

在处理负载均衡问题上, 一种解决方法是由用户定义的调度策略<sup>[13-14]</sup>, 使用户收集应用的负载信息再反馈给调度策略, 然后进行针对性的迭代块划分, 缺点是搜集负载的难度较大, 增加了用户的工作量, 实际操作较困难; 另一种解决方法是动态循环自调度技术 (DLS)<sup>[15]</sup>, 但其仅适用于计算密集型以及不规则循环的调度。

在硬件方面, 硬件感知编译器增强调度<sup>[16]</sup> 结合硬件特点和调度策略, 充分发挥硬件的算力, 更适用于异构平台。

以上调度策略中  $N$  代表循环总的迭代数,  $P$  代表使用的线程数, 迭代块值代表线程分到迭代块的值。当循环迭代数为 1 000, 线程数为 4 时, 不同调度策略的迭代块值如表 1 所示, 动态调度的迭代块值是 100, 指导调度最小为 100。

表 1 不同调度策略的迭代块值

Table 1 Chunksize of different scheduling strategies

调度策略	迭代块值
静态调度	250,250,250,250
动态调度	100,100,100,100
指导调度	250,187,140,105
梯式调度	125,118,111,104
Factoring 调度	125,125,125,125,62,62,62,62
new_guided 调度	125,125,125,125,94,71,53,50

1.3 现有调度策略

现有调度策略无论是 OpenMP 标准调度策略还是梯式调度、Factoring 调度策略、new\_guided 调度策略, 迭代块值的变化主要是线性、非线性、分段线性、线性和非线性相混合。从表 1 可以看出, 除了静态调度和动态调度的迭代块值一直保持不变以外, 其他调度策略迭代块值总体上都是不断减小, 调度策略难以解决负载集中在先分配的迭代块的问题, 特别是在处理递减型循环时, 这种现象尤为严重。针对这种问题, 本文提出一种非线性静态调度的策略, 能够将负载平均划分到各个迭代块中, 并没有调度开销。此外, 对于递增型循环也可以用同样的方法进行处理。递增型循环动态调度和指导调度虽然负载分配较均衡, 但是需要进行多次调度, 调度开销较大。本文调度策略在处理线性循环时与静态调度的区别如图 2 所示, 迭代块的划分由原来的均分变成了非线性划分, 由于迭代块的变化是非线性的, 因此记为 Nonlinear\_static 调度策略。

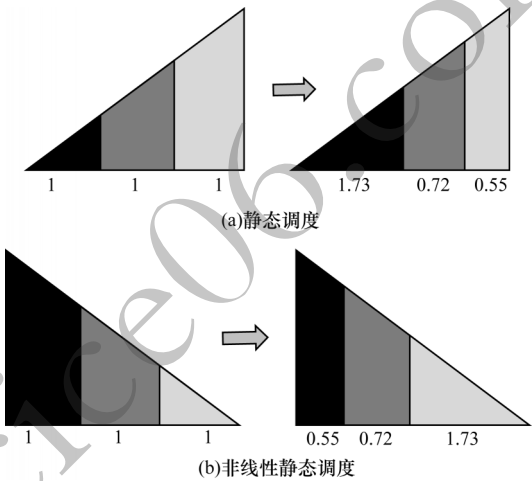


图 2 静态调度和非线性静态调度  
Fig.2 Static scheduling and Nonlinear\_static scheduling

2 Nonlinear\_static 调度策略的设计与实现

2.1 OMPi 编译器

OMP<sub>i</sub> 编译器<sup>[17]</sup> 是由希腊约阿尼纳大学的并行处理小组研发的 OpenMP 编译器, 是一款轻量型、可移植的源到源 C 编译器, 最新开发到了 2.0.0 版本, 支持 OpenMP 4.0 标准。OMP<sub>i</sub> 编译器将程序员编写的带有 OpenMP 指导语句的 C 代码转换为基于 pthreads 等线程库的 C 代码, 然后重新调用指定的 C 编译器进行二次编译并生成可执行文件。由于 OMP<sub>i</sub> 编译器具有轻量、结构性强、易于实现 OpenMP 调度策略等特点, 因此本文调度策略的设计以及实现都是基于 OMP<sub>i</sub> 编译器 2.0.0 版本。

OMP<sub>i</sub> 代码编译示例如图 3 所示, 由于是静态调度, 迭代块值的计算在编译阶段已经完成, 每个线程迭代块的开始位置存放在指针 \*fiter 中, 结束位置存放在指针 \*litter 中, 各个线程按照迭代块并行执行互不干扰并且没有调度开销。



```

1. #pragma omp parallel for schedule(static)
2.   for(i=0;i<1000;i++){
3.       k[i] = i+2;
4.   }

↓

1. for(chid_ = omp_get_thread_num();chid_ +=
TN_){
2.   fiter_ = chid_ * (250);
3.   if(fiter_ >= niters_) break;
4.   liter_ = fiter_ + (250);
5.   if(liter_ > niters_) liter_ = niters_;
6.   for(iter_ = fiter_, i = 0 + fiter_ * 1; iter_
< liter_; iter_++, i += 1)
7.       # 13 "test.c" {
8.         (*k)[i] = i + 2;
9.       }
10.}

```

图3 OMPi代码编译示例

Fig.3 Compilation example of OMPi code

## 2.2 Nonlinear\_static调度模型

Nonlinear\_static调度策略是一种静态调度策略,并程序在编译过程中确定迭代块的划分,没有调度开销。迭代块根据线性递增或线性递减的负载进行平均划分,确保每个线程在并行执行期间都能够处于工作状态,减少线程同步等待时间,提升程序执行效率。模型参数说明如表2所示。

表2 Nonlinear\_static调度模型参数说明

Table 2 Parameter description of Nonlinear\_static scheduling model

参数	含义
$n_{niters}$	循环总的迭代数
$p$	开启的线程数
$l_{ld}$	负载的峰值,递减型循环为第一次迭代的负载,递增型循环为最后一次迭代的负载
$l_{load}$	循环总的负载量
$a_{avg\_load}$	平均负载
$l_{load}(i)$	线程 <i>i</i> 分配到迭代块的负载
$k$	负载线性变化的斜率
$h(i)$	线程 <i>i</i> 分配到迭代块开始处的负载
$f(i)$	线程 <i>i</i> 分配到迭代块的长度

递减循环和递增循环负载分块示意图如图4和图5所示,不同灰度的色块表示不同线程分配到的迭代块。

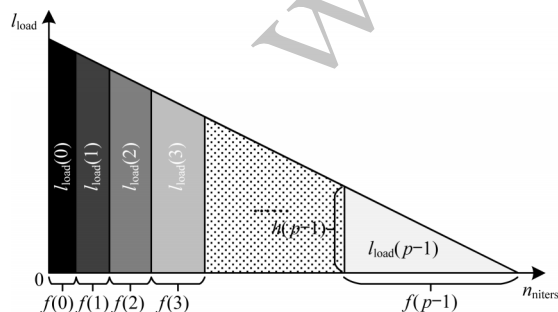


图4 递减循环负载分块示意图

Fig.4 Schematic diagram of decreasing loop load block

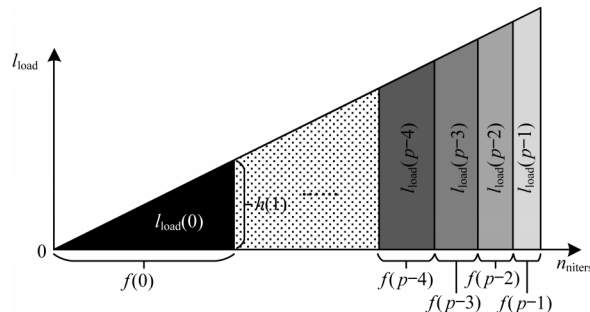


图5 递增循环负载分块示意图

Fig.5 Schematic diagram of incremental loop load block

Nonlinear\_static调度策略的目的是使得每个线程的负载量保持一致,对于递减型循环*f(i)*的推导主要分为以下5个步骤。

1)计算参与的常量(总负载、平均负载、斜率),如式(1)~式(3)所示:

$$l_{load} = \frac{l_{ld} \times n_{niters}}{2} \quad (1)$$

$$a_{avg\_load} = \frac{l_{load}}{p} = \frac{l_{ld} \times n_{niters}}{2p} \quad (2)$$

$$k = \frac{l_{ld}}{n_{niters}} \quad (3)$$

2)列出恒等式,如式(4)、式(5)所示:

$$l_{load}(i) = a_{avg\_load}, i \in \{x | x < p, x \in N\} \quad (4)$$

$$\sum_{i=0}^{p-1} f(i) = n_{niters} \quad (5)$$

3)计算*h(i)*,如式(6)所示:

$$h(i) = k \times \sum_{i=0}^{p-1} f(i) \quad (6)$$

4)计算*l\_load(i)*如式(7)所示:

$$l_{load}(i) = f(i) \times \frac{h(i) + h(i+1)}{2} \quad (7)$$

5)从以上公式推理可得:

$$f(i) = \frac{n_{niters}}{\sqrt{p}} \times (\sqrt{p-i} - \sqrt{p-i-1}) \quad (8)$$

同理可得当循环为线性递增型循环时,*f(i)*如式(9)所示:

$$f(i) = \frac{n_{niters}}{\sqrt{p}} \times (\sqrt{i+1} - \sqrt{i}) \quad (9)$$

式(4)表明每个线程的负载*l\_load(i)*需要等于理论上的平均负载,式(7)用于计算线程*i*根据本文提出的调度策略分配到的负载值。通过以上计算就可以得到*f(i)*,*f(i)*是Nonlinear\_static调度策略划分迭代块的依据,不同的线程获得不同大小的*f(i)*,但对应到负载上每个线程是严格均衡的。在代码实现中,当计算结果不是整数时,处理方法是向下取整。

## 2.3 Nonlinear\_static调度实现

本文在OMP编译器中编码实现一种不同于传统静态调度的非线性静态调度策略,具有没有调度开销以及高数据局部性的优点。Nonlinear\_static调

度按照 $f(i)$ 进行迭代块的划分使得每个线程的负载总是等于 $\text{avg\_load}$ , 保证负载的均衡性。

从代码层面分析, OMPi 编译器模块化清晰明了, 能够更好地进行扩展, 为实现 OpenMP 调度策略提供便利, 这也是在 GCC 编译器和可执行文件之间选择一个源到源编译器的原因。通过 OMPi 编译器更改其中关于 OpenMP 调度的代码, 进而实现自己的调度策略。在 OMPi 编译器中, 本文实现 3 种基本调度策略(静态、动态、指导)的代码在文件/ompi-2.0.0/runtime/host/worksharing.c 中, 不同的调度策略对应各自的函数调用, 其中 `ort_get_static_default_chunk()` 函数用于计算静态调度迭代块的大小 $f(i)$ , 并把 $f(i)$ 的开始迭代索引存放在指针 $*fiter$ 中, 把迭代块结束索引存放在指针 $*litter$ 中。

在 Nonlinear\_static 调度模型中计算每个线程迭代块的大小 $f(i)$ , 作为分块的依据, 但在调度策略代码实现中, 还需要知道迭代块首尾索引的值, 即为 $*fiter$ 、 $*litter$ 赋值。 $s_{\text{start}}(i)$ 为线程 $i$ 分配到迭代块的开始索引, 推导主要用到 2.2 节中的变量。

1) 当循环为线性递减循环时,  $s_{\text{start}}(i)$  如式(10)所示:

$$s_{\text{start}}(i) = n_{\text{iters}} - \frac{f(i)}{2} - \frac{n_{\text{iters}}^2}{2p \times f(i)} \quad (10)$$

2) 当循环为线性递增循环时,  $s_{\text{start}}(i)$  如式(11)所示:

$$s_{\text{start}}(i) = \frac{n_{\text{iters}}^2}{2p \times f(i)} - \frac{f(i)}{2} \quad (11)$$

从 $f(i)$ 以及 $s_{\text{start}}(i)$ 的计算结果得到 $*fiter$ 和 $*litter$ 的值,  $*fiter = \text{start}(i)$ ,  $*litter = *fiter + f(i)$ 。Nonlinear\_static 调度策略是一种静态调度, 本文在 OpenMP 标准静态调度的基础上进行实现的, 总迭代数的获取如图 6 所示, Nonlinear\_static 伪代码如图 7 所示, 其中 `myid` 对应线程 $i$ , `fi` 对应 $f(i)$ , `start` 对应开始索引 $s_{\text{start}}(i)$ 。为保证计算的精确度, 中间变量都定义为浮点型数据。

```
int ort_num_iters(int num, long specs[][2], int
*itp[]){
    int i, totaliters = 1, iters;
    ldiv_t deters;
    for (i = 0; i < num; i++){
        deters = ldiv(specs[i][0], specs[i][1]);
        //总迭代数
        iters = (int) deters.quot;
        if (deters.rem != 0) iters++;
        if (itp[i]) *(itp[i]) = iters;
        totaliters *= iters;
    }
    return (totaliters);
}
```

图6 总迭代数的获取

Fig.6 Obtaining the total number of iterations

```
1. int ort_get_static_default_chunk(int niters, int *fiter,
    int *litter)
2. {
3.     ... //省略初始化和变量定义
4.     ... //省略出错处理
5.     double nit = (double)niters;
        //类型转换, 保证精度
6.     double fi = nit*(sqrt(p-myid) -
        sqrt(p-myid-1))/sqrt(p); //计算f(i), 即迭代块的大小
7.     double start = nit - (nit * nit)/(2*p*fi) - fi/2;
        //计算迭代块 的开始索引
8.     *fiter = start; //开始索引
9.     *litter = *fiter + fi; //结束索引
10.    if(myid == p-1) *litter = niters;
11.    return (*fiter != *litter);
12. }
```

图7 Nonlinear\_static调度的伪代码

Fig.7 Pseudocode of Nonlinear\_static scheduling

### 3 实验与结果分析

#### 3.1 实验环境

本文采用戴尔 Power Edge T640 服务器作为实验平台, 具有 2 个 Intel® Xeon® Silver 4110 CPU, 2 个 CPU 有 8 个核心, 共 16 核心, 主频为 2.10 GHz, 内存为 128 GB。操作系统使用 Ubuntu 16.04, 编译器采用 GCC 7.1.0 版本, 优化选项 -O3, OMPi 编译器使用最新的 2.0.0 版本。

针对 1.1 节提到的 4 种基本循环结构, 本文选取典型的应用程序分别进行调度策略测试。AC (Adjoint Convolution) 是一个递减型循环结构, 用于计算 2 个向量的卷积, 实验规模为  $600 \times 600$ 。CP (Compute Pots)<sup>[18]</sup> 是一个递增型循环结构, 用于更新三维空间中点与点之间的数值关系, 类似于模板计算, 实验规模为 10 000, 源代码发布于 2007 年 Intel 多核平台编码优化赛事。MM (Matrix Multiplication)<sup>[19]</sup> 是一个规则循环结构, 用于计算浮点矩阵乘法, 实验规模为  $2\,500 \times 2\,500$ 。MS (Mandelbrot Set) 是一个随机循环结构, 能够生成复平面上分形图的点的集合, 实验规模为  $15\,000 \times 15\,000$ 。为充分验证测试用例的性能表现, 本文选取的实验规模在保证加速效果来自于调度策略而不是随机因素的前提下, 尽可能进行多次实验来排除随机因素, 以验证调度策略的可靠性以及扩展性。CP 程序与 AC 程序的负载分布如图 8 所示, 负载的单位是核心计算的次数。

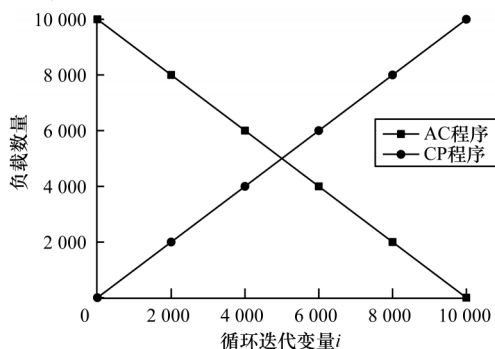


图8 CP和AC程序负载分布

Fig.8 Load distribution of CP and AC programs

本文实验主要研究4种测试用例中的热点循环部分,针对整个热点循环从开始执行到执行完毕的时间,计时操作使用OpenMP提供的函数omp\_get\_wtime(),该函数返回1个双精度浮点值,等于经过的时间(以s为单位)。

3.2 实验结果分析

测试用例串行执行时间如表3所示。为保证数据的全面性,本文分别测试线程数为4、8、12、16时不同调度策略的并行执行时间。针对同一种调度策略,本文还测试了不同迭代块值(由OpenMP指导语句指定)下的并行时间,经过测试迭代块值的选取除了对静态调度有一定的影响外,对于其他调度策略影响较小。因此,本文只列举chunksize=100的情况下,对比不同调度策略的执行时间。

表3 测试用例串行执行时间

Table 3 Serial execution time of test case s	
测试用例	执行时间
MM 程序	114
MS 程序	103
AC 程序	109
CP 程序	75

3.2.1 AC程序并行执行结果

AC是线性循环中的递减型循环。AC程序并行执行时间如表4所示,不同调度策略的AC程序并行执行时间对比如图9所示。从表4和图9可以看出,表现最差的是静态调度和指导调度。静态调度把循环迭代均匀分割在各个线程,造成负载在最先分配的线程上最多,最后分配的线程上最少。指导调度由于迭代块是指数下降的,负载的不均衡度更大。表现较优的是动态调度和new\_guided调度,线程动态申请迭代块,减少了线程的等待,使得线程一直处于工作状态。根据文献[20]的实验数据,梯式调度的效果与new\_guided调度相当。表现最优的是Nonlinear\_static调度,与动态调度和指导调度相比,Nonlinear\_static调度具有更优的负载均衡且没有调度开销。

表4 AC程序并行执行时间

Table 4 Parallel execution time of AC program s				
调度策略	4线程	8线程	12线程	16线程
静态调度	47.74	25.73	17.73	13.58
静态调度,100	30.27	15.61	10.78	8.37
动态调度	30.19	16.19	11.83	8.50
动态调度,100	28.61	14.26	9.53	7.19
指导调度	47.80	25.68	17.88	13.60
指导调度,100	47.72	25.75	17.71	13.59
new_guided调度	30.23	15.73	10.80	8.41
new_guided调度,100	30.25	15.77	10.80	8.42
Nonlinear_static调度	28.67	14.34	9.57	7.20

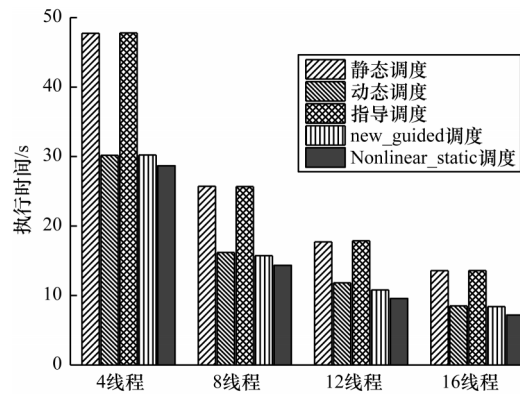


图9 不同调度策略的AC程序并行执行时间对比

Fig.9 Parallel execution time comparison of AC program among different scheduling strategies

3.2.2 CP程序并行执行结果

CP程序并行执行时间如表5所示,不同调度策略的CP程序并行执行时间对比如图10所示。从表5和图10可以看出,对于CP的递增型循环结构,静态调度表现最差,指导调度略弱于动态调度和new\_guided调度,根据文献[20]的实验数据,梯式调度的效果与动态调度相当,Nonlinear\_static调度表现最优。

表5 CP程序并行执行时间

Table 5 Parallel execution time of CP program s				
调度策略	4线程	8线程	12线程	16线程
静态调度	35.67	20.04	14.21	10.29
静态调度,100	27.29	14.70	10.16	7.66
动态调度	27.22	13.18	9.20	6.93
动态调度,100	27.26	13.12	9.29	7.03
指导调度	29.06	15.58	11.15	7.37
指导调度,100	28.29	15.20	10.47	7.26
new_guided调度	27.05	13.78	9.69	7.72
new_guided调度,100	27.01	13.75	9.63	7.80
Nonlinear_static调度	25.79	12.88	8.63	6.53

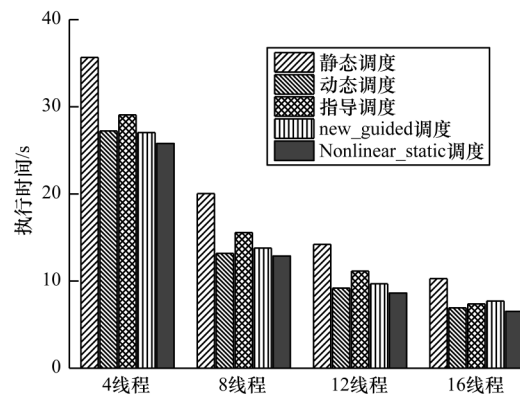


图10 不同调度策略的CP程序并行执行时间对比

Fig.10 Parallel execution time comparison of CP program among different scheduling strategies

3.2.3 MS与MM程序并行执行时间

MS随机循环结构的并行执行时间如表6所示,MM



规则循环结构并行执行时间如表7所示。从表6和表7可以看出,表现最优的是动态调度、指导调度以及new\_guided调度。随机循环结构负载分布是随机变化,因此,需要动态分配迭代块才能实现负载均衡,静态调度以及Nonlinear\_static调度的迭代块分配在编译过程中已经确定了,所以表现最差。在面对规则循环时,Nonlinear\_static调度的表现也是最差的。根据文献[20]的实验数据,梯式调度在面对随机循环结构时效果优于new\_guided调度,在处理规则循环结构时表现出与动态调度相同的效果。

表6 MS程序并行执行时间  
Table 6 Parallel execution time of MS program s

调度策略	4线程	8线程	12线程	16线程
静态调度	49.29	26.81	18.27	13.84
静态调度,100	27.43	13.74	9.71	6.92
动态调度	27.69	14.10	9.22	7.05
动态调度,100	27.50	13.74	9.16	6.99
指导调度	33.89	16.27	10.67	8.30
指导调度,100	33.89	14.94	10.67	8.10
new_guided调度	28.72	14.58	9.46	8.82
new_guided调度,100	28.30	14.34	9.47	7.53
Nonlinear_static调度	53.93	27.00	20.03	14.97

表7 MM程序并行执行时间  
Table 7 Parallel execution time of MM program s

调度策略	3线程	5线程	7线程	9线程
静态调度	22.77	16.04	11.18	7.62
静态调度,100	35.49	18.18	13.57	8.94
动态调度	25.55	13.27	8.97	6.82
动态调度,100	24.68	14.26	9.89	6.65
指导调度	22.81	11.75	8.24	7.03
指导调度,100	22.80	14.92	9.37	7.51
new_guided调度	22.84	13.69	8.29	6.75
new_guided调度,100	23.01	12.60	9.86	7.29
Nonlinear_static调度	40.90	25.76	20.41	17.76

不同调度策略的本质是对chunksize划分不同,循环结构的负载分布是变化的,因此会造成线程分配到的负载量难以明确,反映到OpenMP调度模型上使得先执行完任务的线程,并等待未执行完任务的线程进行同步操作,而这一步骤会降低程序整体的执行效率。从表4~表7可以看出,对于不同的测试用例,除了静态调度在指定chunksize时有明显的加速(对于规则循环是负加速)外,其他的调度策略在指定chunksize时基本没有大的变化。

从整体上分析,静态调度不能有效地平衡负载,除了规则循环结构,一般情况下效果都是最差的。

指导调度的特性导致其不适合处理递减型循环。Nonlinear\_static调度由于静态的迭代块划分规则,无论是哪种类型的循环,其迭代块的划分是固定

的,所以不适合随机循环以及规则循环结构。在其他情况下,动态调度和指导调度性能较优,符合其动态分配迭代块的设计原理,但额外的调度开销使得它们在线性循环时相较于Nonlinear\_static调度效果较差。Nonlinear\_static调度相较于其他调度策略能够更好地处理线性循环,理论与实验结果相一致。

4 结束语

本文提出静态调度策略Nonlinear\_static用于消除调度开销。将线性循环负载均匀变化参数与总负载、负载峰值、线程数相结合构建Nonlinear\_static调度策略的模型,并在OMP编译器中进行编码。实验结果表明,该策略在负载均衡的基础上实现了负载的静态分配,与静态调度、动态调度、指导调度等策略相比,其总体执行时间缩短了5%~10%。后续将在处理非线性循环结构时采用迭代编译方法获得关键信息,同时重新划分迭代块的大小进行二次编译,进一步提高负载的均衡性。

参考文献

[1] OpenMP architecture review board. OpenMP application programming interface, version5.0[EB/OL]. [2020-12-10]. <https://www.openmp.org/resources/>.

[2] 刘晓娟. 面向共享存储结构的并行编译优化技术研究[D]. 郑州:解放军信息工程大学,2013.

LIU X X. Research on parallel compilation optimization technology for shared storage structure[D]. Zhengzhou: PLA Information Engineering University, 2013. (in Chinese)

[3] TZEN T H, NI L M. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers[J]. IEEE Transactions on Parallel and Distributed Systems, 1993, 4(1):87-98.

[4] 张延红,史永昌,朱晓璐. 非规则循环的OpenMP调度算法[J]. 计算机工程,2011,37(6):74-76.

ZHANG Y H, SHI Y C, ZHU X J. OpenMP scheduling algorithm with irregular loop[J]. Computer Engineering, 2011, 37(6):74-76. (in Chinese)

[5] CIORBA F M, IWAINSKY C, BUDER P. OpenMP loop scheduling revisited: making a case for more schedules: evolving OpenMP for evolving architectures[C]// Proceedings of the 14th International Workshop on OpenMP. Berlin, Germany: Springer, 2018:21-36.

[6] 范会敏,李滋田. 基于OpenMP的多线程负载均衡调度策略[J]. 计算机与现代化,2013(12):192-195.

FAN H M, LI Z T. Multi-thread load balancing scheduling strategy based on OpenMP[J]. Computer and Modernization, 2013(12):192-195. (in Chinese)

[7] YANG C T, HUANG C W, CHEN S T. Improvement of workload balancing using parallel loop self-scheduling on Intel Xeon Phi[J]. Journal of Supercomputing, 2017, 73(11):4981-5005.

[8] MOHAMMED A, ELELIEMY A, CIORBA F M, et al. Experimental verification and analysis of dynamic loop

- scheduling in scientific applications[C]//Proceedings of the 17th International Symposium on Parallel and Distributed Computing. Washington D. C. , USA:IEEE Press,2018: 141-148.
- [ 9 ] POLYCHRONOPOULOS C D, KUCK D J. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers [J]. IEEE Transactions on Computers, 1987, 36(12): 1425-1439.
- [ 10 ] HUMMEL S F, HONBERG E, FLYNN L E. Factoring: a method scheme for scheduling parallel loops [J]. Communications of the ACM, 1992, 35(8): 90-101.
- [ 11 ] YANG C T, CHANG S C. A parallel loop self-scheduling on extremely heterogeneous PC clusters [J]. Journal of Information Science and Engineering, 2004, 20 (2) : 263-273.
- [ 12 ] 刘胜飞, 张云泉. 一种改进的 OpenMP 指导调度策略研究[J]. 计算机研究与发展, 2010, 47(4): 687-694.
- LIU S F, ZHANG Y Q. Research on an improved OpenMP guided scheduling strategy [J]. Journal of Computer Research and Development, 2010, 47(4) : 687-694. (in Chinese)
- [ 13 ] KALE V, IWAINSKY C, KLEMM M, et al. Toward a standard interface for user-defined scheduling in OpenMP [C]//Proceedings of the 1st International Workshop on OpenMP. Berlin, Germany: Springer, 2019: 186-200.
- [ 14 ] BAK S, GUO Y, BALAJI P, et al. Optimized execution of parallel loops via user-defined scheduling policies [C]// Proceedings of the 48th International Conference on Parallel Processing. New York, USA: ACM Press, 2019: 1-10.
- [ 15 ] KASIELKE F, TSCHUTER R, IWAINSKY C, et al. Exploring loop scheduling enhancements in OpenMP: an LLVM case study [C]//Proceedings of the 18th International Symposium on Parallel and Distributed Computing. Washington D. C. , USA: IEEE Press, 2019: 131-138.
- [ 16 ] JYOTHI K V S, BALACHANDRAN S. Compiler enhanced scheduling for OpenMP for heterogeneous multiprocessors [EB/OL]. [2020-12-10]. <https://arxiv.org/pdf/1808.06074.pdf>.
- [ 17 ] DIMAKOPOULOS V V, LEONTIADIS E, TZOUMAS G. A portable C compiler for OpenMP V. 2.0 [C]// Proceedings of the 5th European Workshop on OpenMP. Stockholm, Sweden: [s. n. ], 2003: 5-11.
- [ 18 ] Intel. Compute Potts (CP) [EB/OL]. [2020-12-10]. <https://zhuanlan.zhihu.com/p/129331749>.
- [ 19 ] QUN N H, KHALIB Z I A, RAO F R A A. Performance analysis of OpenMP scheduling type on embarrassingly parallel matrix multiplication algorithm [C]//Proceedings of International Conference on Reliable Information and Communication Technology. Berlin, Germany: Springer, 2018: 917-925.
- [ 20 ] 任小西, 唐玲, 李仁发. OpenMP 多线程负载均衡调度策略研究与实现 [J]. 计算机科学, 2010, 37(11): 148-151.
- REN X X, TANG L, LI R F. Research and implementation of OpenMP multi-threading load balancing scheduling strategy [J]. Computer Science, 2010, 37(11): 148-151. (in Chinese)

编辑 薛晋栋