并行复习5 OpenMP编程

> ## OpenMP编程
>
> **OpenMP基础API、归约、parallel for、数据依赖\重排转换、 循环调度**
>
> 静态、动态、指数 三种调度方式
>
> 静态的调度方式有几个任务，采用静态的调度方式 写上参数得知道具体第几轮哪一个任务分到哪一个线程上，怎么分的

- 通过少量编译指示出并行部分和数据共享，即可实现很多串行程序的并行化。
- 是Pthread的常见替代。更简单，但限制也更多。
- OpenMP能：

☒ 程序员只需将程序分为串行和并行区域，而不是构建并发执行的多线程

☒ 隐藏栈管理

☒ 提供同步机制

- OpenMP不能：

☒ 自动并行化

☒ 确保加速

☒ 避免数据竞争 （数据竞争是指在多线程或多进程的并发程序中，两个或多个并发执行的线程或进程同时访问共享的数据，并且至少有一个线程对共享数据进行了写操作，而且这些访问没有进行适当的同步控制。）

OpenMP执行模型

☒ Fork-join并行执行模型

☒执行伊始是单进程（主线程）

☒并行结构开始

☒ 主线程创建一组线程（工作线程）

☒并行结构结束

☒ 线程组同步——隐式barrier

☒只有主线程继续执行

☒实现优化

☒ 工作线程等待下一次fork

编译指示

```
#pragma omp parallel [clause[ [,] clause] ...]
{
    // 并行执行的代码块
}
其中，clause 是一些可选的指令修饰符，用于控制并行区域的一些特性。
一些常见的 clause 包括：
num_threads(n)：指定并行区域中的线程数目。
private(list)：指定私有变量，每个线程都有自己的一份。
shared(list)：指定共享变量，所有线程共享一份。
```

```
//条件编译
#ifdef _OPENMP
…
printf(“%d avail.processors\n”,omp_get_num_procs());
#endif
```

大小写敏感

```
使用库函数需要包含头文件
#ifdef _OPENMP
#include <omp.h>
#endif
```

- 临界区指令

所有线程都执行，但每个时刻限制只有一个线程执行

```
被临界区包围的代码
#pragma omp critical
语句块
线程在临界区开始位置等待，直至组中没有其他线程在同名临界区中执行
所有未命名临界区指示都映射到相同的未指定名字
```

编程模型

数据共享

- 共享数据：所有线程可见
- 私有数据：单线程可见

```
shared(bigdata)
private(tid)
默认是shared
```

查询函数

```
int omp_get_num_threads(void);
返回执行当前并行区域的线程组中的线程数
int omp_get_thread_num(void);
返回当前线程在线程组中的编号，值在0和omp_get_num_threads()-1之间。
主线程的编号为0
```

OpenMP归约

归约就是将相同的归约操作符重复地应用到操作数序列来得到一个结果的计算。

对for循环自动并行化，默认块状划分。

```
sum=0;
#pragma omp parallel for reduction(+:sum)
for(i=0;i<100;i++)
{
   sum+=array[i];
}
```

OpenMP数据并行：并行循环

parallel for

编译器为每个线程计算负责的循环范围

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    // 并行执行的代码块
```

```
4 }
```

数据依赖

- 竞争条件：执行结果依赖于两个或更多事件的时序，则存在竞争条件

竞争条件发生在多个线程或进程尝试同时修改共享资源，而最终的结果取决于执行的顺序。

- 数据依赖：两个内存操作的序，为了保证结果的正确性，必须保持这个序

数据依赖发生在一个操作依赖于另一个操作的结果，而这两个操作被分配给不同的线程。

数据依赖可能导致程序无法正确地执行，因为某些操作可能在依赖的操作完成之前就开始执行。

☒ 如果两个内存访问指向相同的内存位置且其中一个是写操作，则它们产生数据依赖

重排转换

同步点之间并行执行的计算可能重排执行顺序。如果它能保持代码中的依赖关系，则它是安全的。

循环调度

在并行循环（parallel for）中将迭代分配给不同的线程以进行并行执行的方法。

静态划分

先分配好

在静态调度中，循环的迭代被静态地分配给不同的线程。每个线程被分配一定数量的连续迭代，这样在编译时就已经确定了每个线程要执行的迭代范围。静态调度适用于迭代次数已知且相对均匀的情况。

```
1 #pragma omp parallel for schedule(static)
2 for (int i = 0; i < N; ++i) {
3     // 并行执行的代码块
4 }
```

static([chunk])静态划分 分配给每个线程chunk步迭代，所有线程都分配完后继续循环分配，直至所有迭代步分配完毕。默认chunk为ceil=iterations/threads

dynamic([chunk])动态划分 分给每个线程chunk步迭代，一个线程完成任务后再为其分布chunk步迭代

默认chunk为1

动态划分

看情况分配

在动态调度中，迭代的分配是动态进行的。每个线程完成一次迭代后，会从迭代剩余的集合中获取下一个迭代。动态调度适用于迭代次数不均匀的情况，因为它可以更灵活地分配工作负载。

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; ++i) {
    // 并行执行的代码块
}
```

调度决策对性能的影响

负载均衡

静态划分可能会导致负载不均衡，因为无法预测每个迭代的执行时间

动态划分可以根据执行时的情况动态分配工作，更有可能实现负载均衡

调度开销

动态调度通常会引入更多的调度开销，因为它需要在运行时进行决策，而静态调度在编译时就已经确定了执行计划。因此，对于小型任务，静态调度可能更有效。

数据局部性

静态调度可能会有助于维持较好的数据局部性，因为在编译时就已经确定了执行顺序。相反，动态调度可能会引入不可预测的执行顺序，可能会降低数据局部性。

总结

动态划分对负载均衡更有利 但会增加调度开销

静态划分有助于维持较好的数据局部性 但可能导致负载不均衡

总之：静态划分适合小规模问题 动态划分适合大规模问题