并行与分布式计算基础 Part3——openmp

邓喻源 数学科学学院

OpenMP的基础模式与特点

Fork-Join模式

流程：主程序进入并行区，每个并行区中间分多个线程，这些线程并行执行，同步开始和结束。最后只保留主线程

OpenMP的特点

显式并行：由程序员对并行进行控制(而不是系统自动并行)

数据域：同一个并行区中的所有线程的数据默认是共享的，而且用户可以指定变量的生存周期（如是否需要在并行区域结束后保留变量的值）以及是否私有

嵌套并行：OpenMP 的API 允许一个并行区内开启另一个并行区（某些库不允许）

动态线程：OpenMP 的API 允许通过运行时的环境设置动态改变一个并
行区的线程数（某些库不允许）

OpenMP API 的三要素

运行时库：主要是头文件、库函数的调用和链接

环境变量：OpenMP API预定义了一些环境变量，运行时控制程序的行为

编译制导语句：在程序中添加一些特殊格式的注释，如果编译器支持OpenMP，则可以来实现Open的一些功能

OpemMP的21个常用核心：参考ppt第16页（**感觉重要**）

OpenMp的编写和运行

编译与运行

编译命令：$ gcc omp_hello.c -o hello -fopenmp

编写程序主体

进行时间计量

```
#计时程序
double start = omp_get_wtime();
// Your code
double end = omp_get_wtime();
printf("Time taken: %f seconds\n", end - start);
```

线程数

1.通过环境变量：$ export OMP_NUM_THREADS =4

获取线程数：$ echo OMP_NUM_THREADS

2.通过库函数：void omp_set_num_threads(int)

获取线程数：int omp_get_num_threads(void)

获取线程号：int omp_get_thread_num(void)

通过这种方式设置的线程数会更新自己以及之后的所有线程数

线程七问：

Q1：如果没有设置线程数，那么OpenMP 会使用默认的线程数。默认线程数通常是系统可用的物理核心数（或由 OpenMP 实现决定）

Q2：不能在并行区内部直接设置线程数： 无论通过环境变量还是 `omp_set_num_threads`，都需要在并行区域之外完成。

原因是并行区域中的线程数在区域开始时已经确定，不能动态改变。因此，在并行区域内部调用 `omp_set_num_threads` 不会影响当前区域的线程数，只会影响后续的并行区域。

Q3：如果环境变量和程序中设置了不同的线程数怎么办？

- 优先级规则：
  - 如果代码中调用了 `omp_set_num_threads`，那么该设置优先于环境变量 `OMP_NUM_THREADS`。
  - 如果代码中未调用 `omp_set_num_threads`，则使用环境变量中的线程数。

Q4：在串行区域执行 omp_get_num_threads时，会返回1，即主线程

Q5：串行区执行omp_get_thread_num会返回默认值0

Q6：因为是线程号系统自动分配的，因此只能获取，不能设置

Q7：线程编号是运行时动态生成的，它是线程团队中的一个运行时属性，而环境变量是静态的，无法动态反映线程状态，因此不能通过环境变量获取线程号

程序输出

注意：同一个并行区域中间，线程id不是按照固定顺序出现的，而是随机的

从句

控制从句

if 从句：决定是否以并行的方式执行并行区

num_threads 从句：指定并行区的线程数

数据域从句

private 从句：指定私有变量列表；
▶ 每个线程生成一份与该私有变量同类型的数据对象；
▶ 声明为私有变量的数据在并行区中都需要重新进行初始化

firstprivate 从句：指定自动初始化的私有变量列表；
▶ 与private 从句定义的私有变量类似，不同点是
▶ 在并行区执行伊始对该变量根据主线程中的数据进行初始化

shared 从句：指定共享变量列表；
▶ 共享变量在内存中只有一份，所有线程都可以访问；
▶ 编程中要确保多个线程访问同一个公有变量时不会有冲突。

default 从句：指定默认变量类型；
▶ shared：默认为共享变量；
▶ none：无默认变量类型，每个变量都需要另外指定。

reduction 从句：指定规约变量列表；
▶️ 与private 从句定义的私有变量类似，不同点是
▶️ 各个线程对该变量额外进行operator 定义的规约（规约指最后的总结，例如求和求乘）操作。

特殊语句中的循环构造

For 循环

开始语句：必须是“变量= 初值”形式
终止语句：必须明确变量与边界值的大小关系，而且边界值不能变化
计数语句：必须采用规范的等步长累加或者累减
不能使用break、goto、return 等
循环变量必须是整数，初值、边界和增量在循环中固定

#pragma omp parallel for只能管到一个（也就是第一个）for循环

三种私有变量

private 从句：
▶️ 每个线程生成一份与该私有变量同类型的数据对象；
▶️ 声明为私有变量的数据在并行区中都需要重新进行初始化。
firstprivate 从句：
▶️ 与private 从句定义的私有变量类似，不同点是在并行区执行伊始，对该变量根据主线程中的数据进行初始化。
lastprivate 从句：
▶️ 与private 从句定义的私有变量类似，不同点是在并行区执行结束，将执行最后一个循环的线程的私有数据取出。

schedule从句

schedule (type [,chunk])

type是调度类型，chunk是迭代块大小

⋆ static：静态调度，chunk 大小固定(默认：n/t)
⋆ dynamic：动态调度，chunk 大小固定(默认：1)（在迭代时间不一致时使用）
⋆ guided：动态调度，chunk 大小动态缩减（优于dynamic，动态分配任务的粒度会逐步减小）
⋆ runtime：由环境变量OMP_SCHEDULE 确定(上述三种之一)
⋆ auto：系统自选

order从句

order从句是为了保证for中的代码按照顺序执行而不并行

注意1：ordered 从句和构造必须同时存在才起作用
注意2：ordered 区内的语句任意时刻仅由最多一个线程执行

示例代码：

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

※特别需要注意schedule对order分配的影响

collapse从句

collapse 从句的作用：将多层嵌套循环合并为一个大的循环，提升 OpenMP 并行化能力。相当于磨小任务粒度

适用场景：嵌套循环任务中，当外层循环迭代次数不足时，`collapse` 可以显著提高线程利用率。

性能注意：选择合适的 `collapse(n)` 参数，避免任务粒度过小或调度开销过高。

最佳实践：用于 2-3 层嵌套循环的并行化，且任务负载均匀时效果最佳。

特殊语句中的分块构造

sections构造

对非循环任务多线程并行执行(前提：已在并行区内)

每个section段分别并发执行，也就是手动分区并行

single构造

对并行区内的一段代码单线程执行

用于确保某段代码仅由一个线程执行，而其他线程跳过该代码

OpenMP 会自动选择一个线程来执行 single 块中的代码。

Q：为什么single 构造不能与并行区合并?

A：如果 single 与 parallel 合并，难以理解或解释合并后的语义。

nowait

去掉工作共享构造末尾的隐式栅栏同步

支持的从句

从句汇总

	parallel	for	parallel for	sections	parallel sections	single
if	●		●		●	
num_threads	●		●		●	
default	●		●		●	
copyin	●		●		●	
shared	●	●	●		●	
private	●	●	●	●	●	●
reduction	●	●	●	●	●	
firstprivate	●	●	●	●	●	●
lastprivate		●	●	●	●	
schedule		●	●			
ordered		●	●			
collapse		●	●			
nowait		●		●		●

并行的条件

Bernstein条件(充分)

设 P_1 和 P_2 是两个程序，若满足以下三个条件，则 P_1 和 P_2 可以并行执行：

$$P_1 \parallel P_2 \quad \text{if} \quad \begin{cases} I(P_1) \cap O(P_2) = \emptyset, \\ O(P_1) \cap I(P_2) = \emptyset, \\ O(P_1) \cap O(P_2) = \emptyset. \end{cases}$$

符号解释：

1. $I(P)$：表示程序 P 的输入集合（Input set），即程序需要读取的数据。
2. $O(P)$：表示程序 P 的输出集合（Output set），即程序将要写入的数据。

数据依赖

竞争条件(race condition)：指的是并行程序的执行结果具有随机性，依赖于某些事件的发生顺序。
数据依赖(data dependency)：指的是两个或两个以上的程序访问同一片内存，且至少有一个程序执行了写操作。

基本依赖定理

当且仅当程序中所有不可消除的数据依赖都得以满足的条件下，并行程序的执行满足串行一致性。

基本依赖定理 的核心在于：
对于程序中的任何两条指令 S_1 和 S_2，如果存在数据依赖关系，则 S_1 和 S_2 必须按特定顺序执行，不能并行化。

下面我们讨论具体的数据依赖的分类

(1) 流依赖（Flow Dependence，RAW: Read After Write）

定义：程序的后续操作需要使用之前操作写入的数据。

特性：

必须按照写 → 读的顺序执行，否则后续操作无法正确获取数据。

又称为**真实依赖**。

示例：

```
P1: A = x + y;  // 写 A
P2: B = A + z;  // 读 A
```

- P2的执行依赖于 P1 写入的 A。

---

(2) 反依赖（Anti-Dependence，WAR: Write After Read）

定义：程序的后续操作需要写入数据，而之前的操作读取了相同的数据。

特性：

必须按照读 → 写的顺序执行，否则后续操作可能会覆盖未完成的读取操作。

通常由于**变量重用**引起。

示例：

```
P1: B = x + y;  // 读 B
P2: A = B + z;  // 写 B
```

- P2 的写入操作必须等到 P1 完成对 B 的读取。

---

(3) 输出依赖（Output Dependence，WAW: Write After Write）

定义：程序的多个操作试图写入相同的数据。

特性：

必须按照写 → 写的顺序执行，否则后续操作可能会覆盖之前的写入结果。

示例：

```
P1: A = x + y;  // 写 A
P2: A = x + z;  // 写 A
```

P2 的写入操作必须等到 P1的写入完成。

RAW（流依赖）

$O_1 \cap I_2 \neq \emptyset$

程序 P_1 的输出是程序 P_2 的输入，必须顺序执行。

WAR（反依赖）

$I_1 \cap O_2 \neq \emptyset$

程序 P_1 先读取某变量，程序 P_2 写入相同变量，必须顺序执行。

WAW（输出依赖）

$O_1 \cap O_2 \neq \emptyset$

程序 P1P_1P1 和 P2P_2P2 同时写入相同变量，必须顺序执行。

---

3. 哪些依赖可以消除？哪些不能消除？

(1) 可消除的依赖

反依赖（WAR）和 输出依赖（WAW）可以通过变量重命名消除。

反依赖（WAR）：

- 通过分配新的变量名，避免写入覆盖读取的数据。

输出依赖（WAW）：

- 通过分配新的变量名，避免多个写入操作冲突。

(2) 不可消除的依赖

流依赖（RAW）

通常不可消除，因为它反映了程序的逻辑正确性。

- 例如，后续操作需要依赖前一操作的结果，这是程序执行顺序的核心要求。

---

4. 规约属于哪种数据依赖？

规约（Reduction）的特点

- 在规约操作中（如求和、最大值、乘积等），多个线程会并行计算部分结果，最后归约为一个总结果。
- 规约可能涉及 **WAW（输出依赖）** 或 **RAW（流依赖）**，因为最终结果依赖于多个线程的中间计算结果。

同步构造(synchronization construct)

barrier 构造：栅栏同步

在并行区中特定位置显式加入栅栏同步：

#pragma omp barrier

single 构造：单线程执行，有同步

对并行区内的一段代码单线程执行，有同步(可用nowait 去掉)：

```
#pragma omp single [clause1 clause2 ...]
{
code();
}
```

master 构造：主线程执行，无同步

对并行区内的一段代码采用主线程执行，无同步：
(可以看作是一种加上nowait 的特殊版的single 构造)

```
#pragma omp master
{
code();
}
```

ordered 构造：按循环顺序执行

critical 构造：各线程依次执行，程序片段

对并行区内的一段代码依次互斥(mutual exclusion, mutex) 执行：

```
#pragma omp critical [(name) hint(..)]
{
code();
}
```

atomic 构造：各线程依次执行，单一指令

一些概念

竞争条件(race condition) 和线程安全(thread safe)

竞争条件(race condition) 指的是并行程序的执行结果具有随机性，依赖于某些事件的发生顺序，在OpenMP 中竞争条件的产生往往是由于多个线程同时更新同一片内存地址空间(如共享变量)

可以使用critical、atomic、reduction从句等避免竞争

称程序为线程安全(thread safe)，一般指竞争条件可以完全避免，如I/O 操作、OS 操作、通用库函数等均有可能不是线程安全的，需要使用单线程调用

程序的遗孤(orphaning)

线程控制

动态线程

打开/关闭动态线程

库函数：void omp_set_dynamic(int flag)（flag为0就是全开启，否则开启一部分，并由系统决定开启哪一部分）

环境变量：export OMP_DYNAMIC=true

检查动态线程是否打开

库函数：int omp_get_dynamic (void)

嵌套并行

嵌套并行：指在并行区之内开启并行区(默认：一般为开启)。

打开/关闭嵌套并行

库函数：void omp_set_nested(int flag)

环境变量：
export OMP_NESTED=true
export OMP_NUM_THREADS=n1 ,n2 ,n3 ,...

检查嵌套并行是否打开

库函数：int omp_get_nested (void)

flush构造

flush 构造：手动更新当前线程本地缓存中的数据。

#pragma omp flush [acq_rel | release | acquire] (list)

ps：合理的算法设计一般不需要显式的flush (因为容易出错)

OpenMP 任务构造

定义：任务构造让程序可以显式地创建**任务**，这些任务由 OpenMP 的运行时系统动态分配给线程。

定义任务：

#pragma omp task [clause1 clause2]
...

注意：过度分治会导致性能下降

向量化

向量化是一种利用 **SIMD**（Single Instruction Multiple Data，单指令多数据）硬件功能的技术，通过一次执行一条指令来处理多个数据，从而加速程序的执行。

SIMD 向量化的效果与数据是否对齐(aligned) 有很大关系