并行与分布式计算基础：第十二讲

杨超
chao_yang@pku.edu.cn

2019 秋

课程基本情况

- 课程名称：并行与分布式计算基础
- 授课教师：杨超（chao_yang@pku.edu.cn，理科 1 号楼 1520）
- 课程助教：尹鹏飞（pengfeiyin@pku.edu.cn）

授课内容（暂定）

- 引言
- 硬件体系架构
- 并行计算模型
- 编程与开发环境
- MPI 编程与实践
- OpenMP 编程与实践
- GPU 编程与实践
- 前沿问题选讲

上课时间（地点：二教 211）

上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)					
第 12 节 (20:40-21:30)					

内容提纲

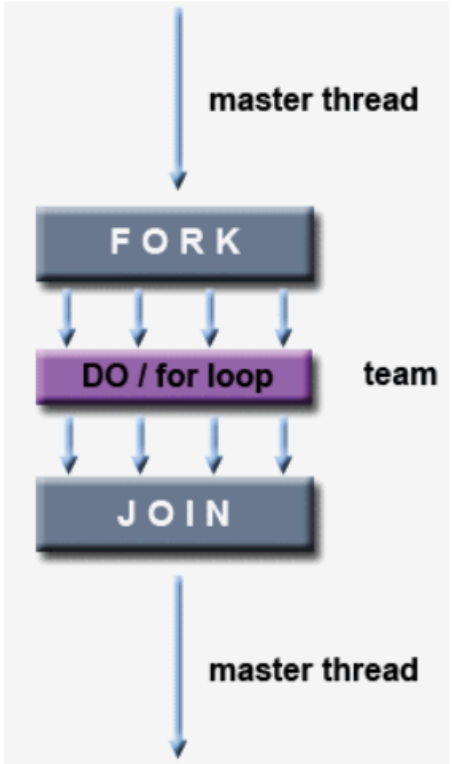
1. OpenMP 编程-3：提高篇
   - 工作共享构造-1 (回顾)
   - 工作共享构造-2
   - 从句汇总
   - 数据依赖

工作共享构造-1 (回顾)

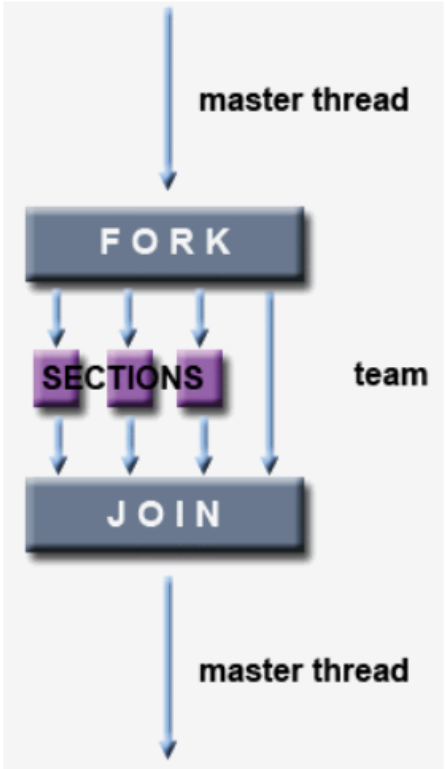
工作共享构造 (work-sharing construct)

用于将代码分配采用某种机制给不同的线程执行：循环、分块、单独
(注：在入口没有同步，但是在出口包含了一个隐含的栅栏同步)。

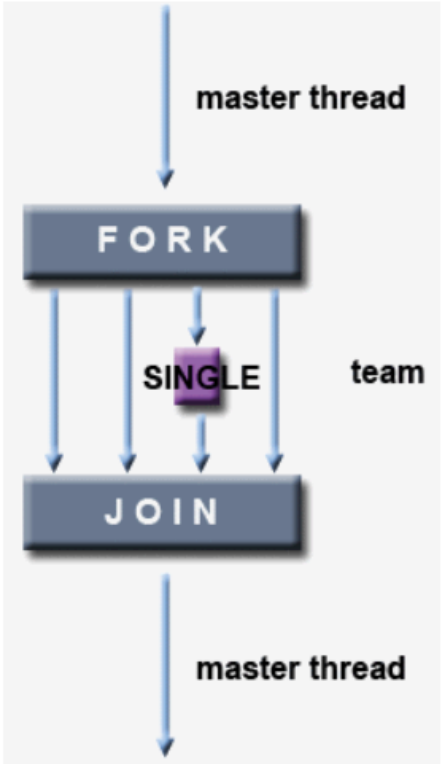
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



for 循环工作共享构造 (1)

- 用于对循环进行多线程并行执行 (前提：已经在并行区内)：

```
#pragma omp for [clause1 clause2 ...]
  for (...) {
    ...
  }
```

- 支持的从句：

```
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait
```

for 循环工作共享构造 (2)

- OpenMP 的 for 循环构造对 for 循环的格式有严格要求：
  - 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；
  - 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。
- 如果并行区中只有一个 for 构造，则可以使用：

```
#pragma omp parallel for
  for (i=1; i<=N; i++) {
    code2();
  }
```

支持的从句概览

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

数据域从句：默认变量、共享变量和规约变量

```
default (shared | none)
shared (list)
reduction (operator: list)
```

- 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。
- 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。
- 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 定义的规约操作。

数据域从句：三种私有变量

```
private (list)
firstprivate (list)
lastprivate (list)
```

- 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。

线程调度：schedule 从句

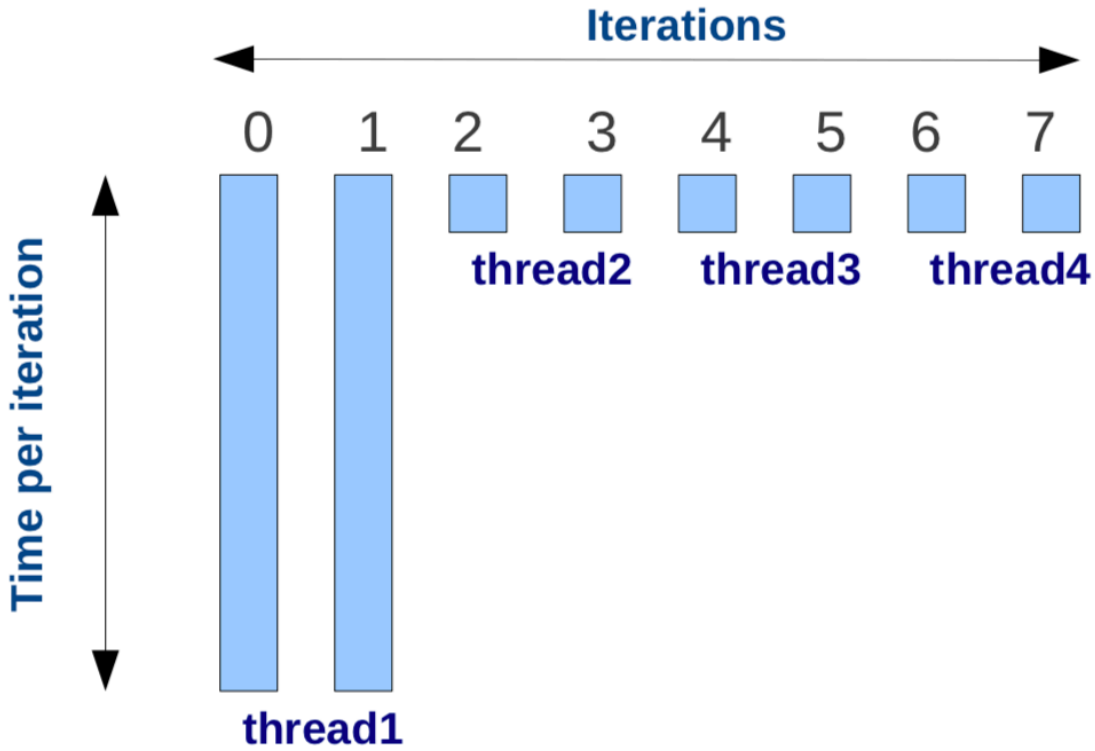
- schedule 从句：主要用于控制调度方式。

```
schedule (type [,chunk])
```

  - type：调度类型，包括：
    - ★ 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)；
    - ★ guided：动态调度，chunk 大小动态缩减；
    - ★ runtime：由环境变量 OMP_SCHEDULE 确定 (上述三种之一)；
    - ★ auto：系统自选。
  - chunk：分块大小，必须是正整数。

为什么需要进行动态调度? (1)

- 每个迭代步的耗时可能不平均!



为什么需要进行动态调度？ (2)

- 比如，计算二重积分：

$$\int_0^1 \int_0^y f(x,y)dxdy.$$

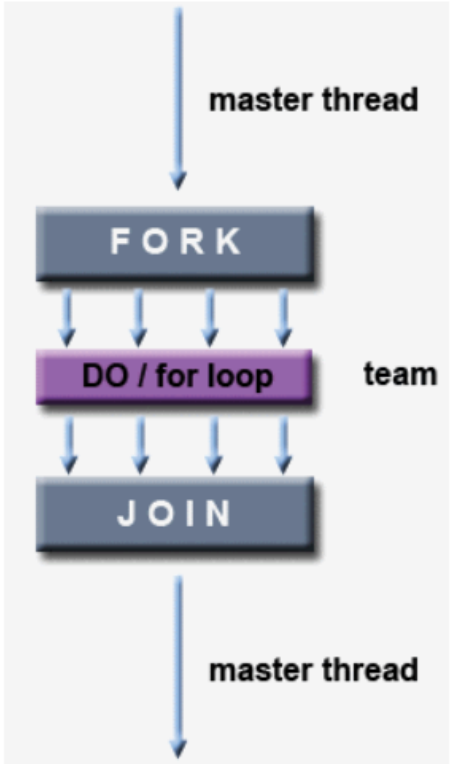
```
1   ...
2   sum = 0.0;
3 #pragma omp parallel for reduction(+:sum)
4   for (i = 0; i < n; i++) {
5     for (j = 0; j < i; j++) {
6       ...
7       sum += ...;
8     }
9   }
10  ...
```

工作共享构造-2

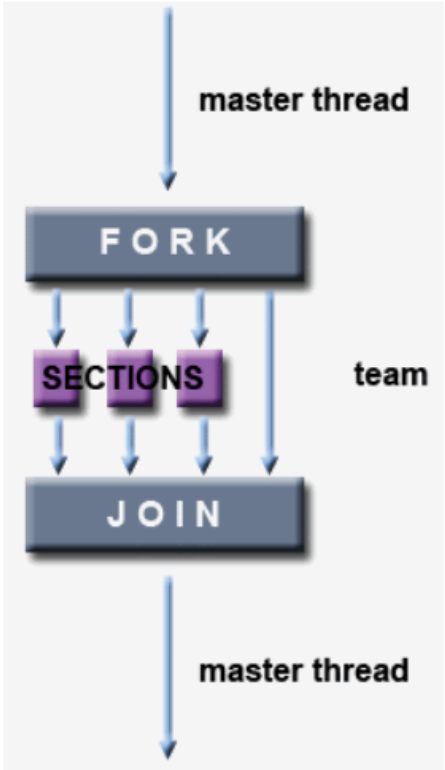
工作共享构造 (work-sharing construct)

用于将代码分配采用某种机制给不同的线程执行：循环、分块、单独
(注：在入口没有同步，但是在出口包含了一个隐含的栅栏同步)。

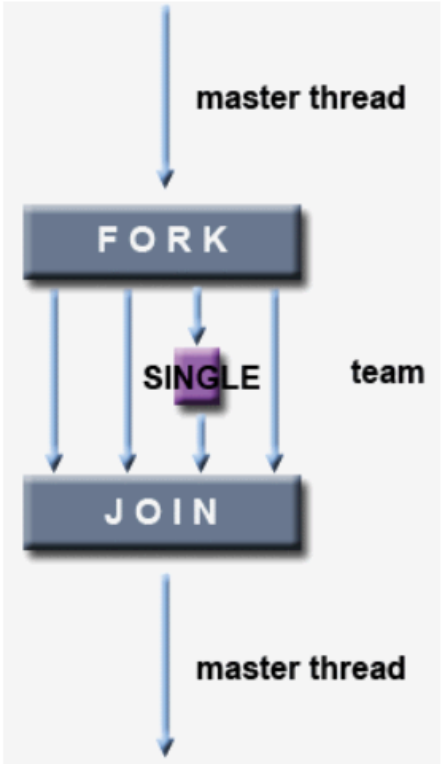
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



for 循环构造

- 对并行区内的循环多线程并行执行:

```
#pragma omp for [clause1 clause2 ...]
  for (...) {
    ...
  }
```

- 支持的从句:

```
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait
```

ordered 从句与 ordered 构造 (1)

- ordered 从句：声明 for 循环中有潜在的顺序执行部分

```
ordered
```

- ordered 构造：声明循环中的顺序执行代码区

```
#pragma omp ordered
```

- 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；
- 注意 3：为了提升并行度，需要合理调整循环的 schedule 方式.

ordered 从句与 ordered 构造 (2)

- 举例:

```
1 #pragma omp parallel for private(myval) ordered
2   for(i=0; i<n; i++){
3     myval = do_lots_of_work(i);
4 #pragma omp ordered
5     {
6       printf("%d %d\n", i, myval);
7     }
8   }
```

- 思考: 如果线程数为 3，迭代次数 $n = 9$，不同 schedule 方式对执行结果的影响是什么?
- 提示: 循环的默认 schedule 方式 (static) 导致整个循环几乎完全串行执行!

collapse 从句

```
collapse (n)
```

- collapse 从句：将多重循环展开到第 n 重。
  - 待展开的循环间必须没有依赖关系；
  - 展开后的顺序与串行迭代顺序一致；
  - 相当于增大外层循环次数，从而有助于 schedule。
- 举例：

```
1 #pragma omp parallel for collapse(2)
2   for (i = 0; i < 10; i++) {
3     for (j = 0; j < 100; j++) {
4       ...
5     }
6   }
```

sections 构造 (1)

- 对非循环任务多线程并行执行 (前提：已在并行区内)：

```
#pragma omp sections [clause1 clause2 ...]
{
#pragma omp section
  code1();
#pragma omp section
  code2();
  ...
}
```

- 支持的从句：

```
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait
```

sections 构造 (2)

- sections 构造内由 section 划分出不同的程序段;
- 各个 section 程序段分别并发执行;
- 每个程序段由一个线程执行:
  - 线程数等于 section 数: 线程与程序段一一对应;
  - 线程数大于 section 数: 个别线程空闲;
  - 线程数小于 section 数: 个别线程执行多于一个程序段;
- 无法提前得知哪个线程执行哪个程序段;
- 唯一知道的是每个程序段被执行且只被执行了一次。

single 构造

- 对并行区内的一段代码单线程执行：

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

- 支持的从句：

```
private (list)
firstprivate (list)
nowait
```

  - 无法提前得知是哪个线程执行 single 标记的代码；
  - 其他线程等待该线程执行完毕后进行同步；
  - 一般用于处理非线程安全 (thread safe) 的任务，
    如 I/O、对共享变量赋值等。

与并行区合并

- 如果并行区中只有一个工作共享构造，则可以合并：

```
#pragma omp parallel for
  for (i=1; i<=N; i++) {
    code();
  }
```

```
#pragma omp parallel sections
{
#pragma omp section
  code1();
#pragma omp section
  code2();
}
```

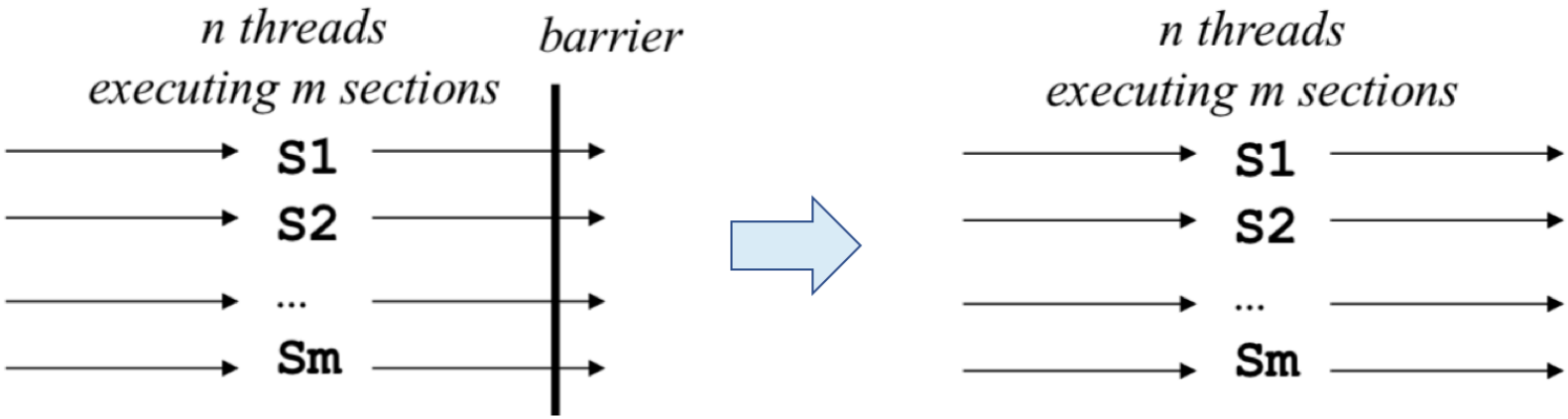
- 思考：为什么 `single` 构造不能与并行区合并?

nowait 从句

nowait

- 去掉工作共享构造末尾的隐式栅栏同步;
- 可以用于 for、sections、single，比如:

```
#pragma omp parallel
#pragma omp sections nowait
...
```



从句汇总

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		●		●		●

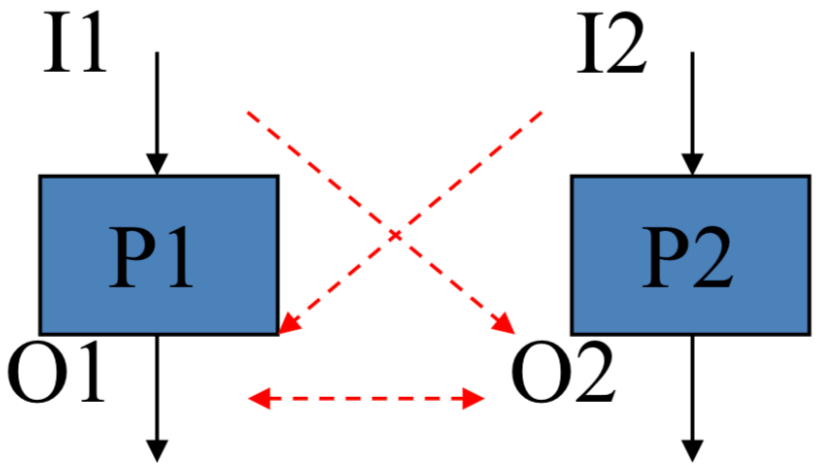
数据依赖

什么情况下两个程序可以并行执行?

一个关键问题

任给两个程序 P_1, P_2，什么情况下它们可以并行执行呢?

思路 记 $I(P)$ 为某个程序 P 读取的数据集合，$O(P)$ 为 P 写出的数据集合。考虑：集合 $I(P_1), I(P_2), O(P_1), O(P_2)$ 之间的关系!



Bernstein 条件

Bernstein's Condition (1966)

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

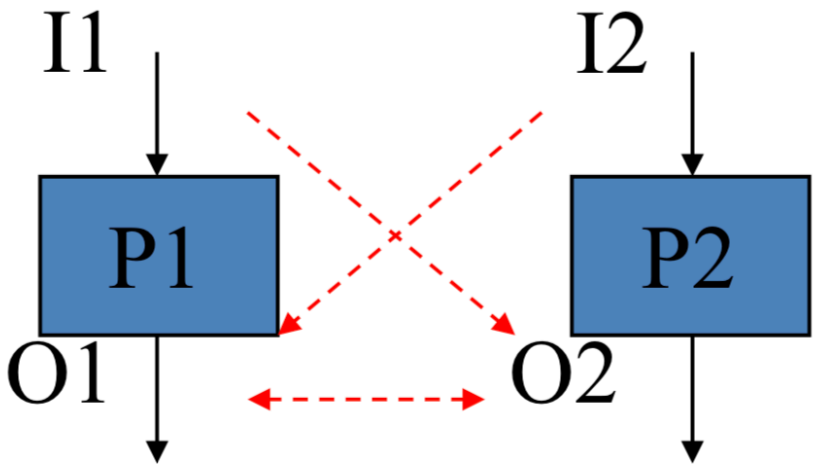
- 推论：当任两个程序 P_i 和 P_j 都满足 Bernstein 条件时，一定有

$$P_1; P_2; \cdots; P_n \Rightarrow P_1 \parallel P_2 \parallel \cdots \parallel P_n.$$

- 注 1：Bernstein 条件是充分条件而非必要条件，事实上，找不到一个算法可以确定任意两个程序是否可以并行！
- 注 2：这里的 Bernstein 不是那个俄罗斯数学家！

竞争条件和数据依赖

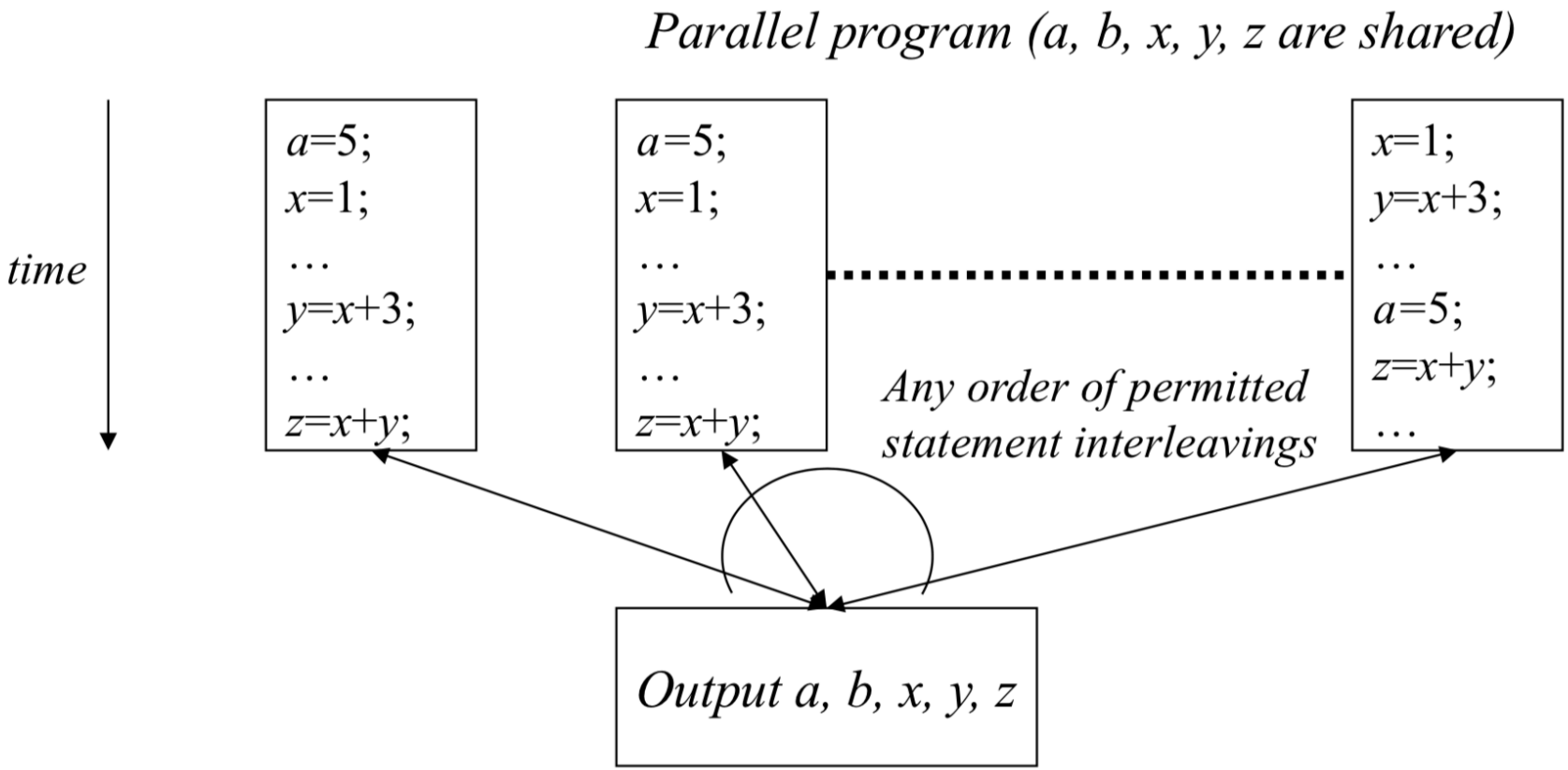
- 竞争条件 (race condition)：指的是并行程序的执行结果具有随机性，依赖于某些事件的发生顺序。
- 数据依赖 (data dependency)：指的是两个或两个以上的程序访问同一片内存，且至少有一个程序执行了写操作。



并行程序的串行一致性

串行一致性 (sequential consistency)

输入不变的情况下，调整并行程序的语句顺序，输出维持不变，则称为这种调整满足了串行一致性。



基本依赖定理

基本依赖定理 (Fundamental Theorem of Dependence)

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。

- 一个需要排除的特例：规约 (reduction)
  - (1). 操作类型：$variable = variable$ **op**..., 且
  - (2). **op** 满足交换律。
- 什么叫不可消除的数据依赖?

数据依赖的分类

三种基本数据依赖关系

- 1. 流依赖 (flow dependence)：RAW = Read After Write；
- 2. 反依赖 (anti-dependence)：WAR = Write After Read；
- 3. 输出依赖 (output dependence)：WAW = Write After Write。

independent	*RAW*	*WAR*	*WAW*
P_1: `A = x + y;`	P_1: `A = x + y;`	P_1: `A = x + B;`	P_1: `A = x + y;`
P_2: `B = x + z;`	P_2: `B = x + A;`	P_2: `B = x + z;`	P_2: `A = x + z;`
$I_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \{\mathbf{B}\}$	$I_1 \cap O_2 = \varnothing$
$I_2 \cap O_1 = \varnothing$	$I_2 \cap O_1 = \{\mathbf{A}\}$	$I_2 \cap O_1 = \varnothing$	$I_2 \cap O_1 = \varnothing$
$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \{\mathbf{A}\}$

- 思考 1：哪些为可消除的数据依赖？哪些不可消除？
- 思考 2：规约属于哪种数据依赖？

不可消除的依赖

- 流依赖又称真依赖 (true dependence)，是唯一一种不可消除的依赖!
- 其他依赖类型均可以通过某些方式消除;
- 规约依赖是一种特殊的存在，当操作满足交换律时不必消除;
- 好的编译器能够帮助程序员消除一些可以消除的数据依赖;
- 好的程序员不太需要编译器帮助做这样的事情;-)

回到 OpenMP

循环携带的 (loop-carried) 数据依赖:

```
1   ...
2 #pragma omp parallel for
3   for (i = 0; i < 99; i++) {
4     x = b[i] + c[i];
5     a[i] = a[i+1] + x;
6   }
7   ...
```

- 思考 1: 这段代码有几种数据依赖?
- 思考 2: 如何消除其中循环携带的数据依赖?

消除数据依赖 (1)

- 方法 1：变量消去

```
1   ...
2 #pragma omp parallel for
3   for (i = 0; i < 99; i++) {
4     // x = b[i] + c[i];
5     // a[i] = a[i+1] + x;
6     a[i] = a[i+1] + b[i] + c[i];
7   }
8   ...
```

  - 只能去掉一些比较容易解决的数据依赖。

消除数据依赖 (2)

- 方法 2：变量私有化

```
1   ...
2  #pragma omp parallel for lastprivate(x)
3    for (i = 0; i < 99; i++) {
4      x = b[i] + c[i];
5      a[i] = a[i+1] + x;
6    }
7    ...
```

  - 思考：为什么用 `lastprivate` 而不是 `private`?

消除数据依赖 (3)

- 方法 3：变量替换

```
1   ...
2  #pragma omp parallel for
3    for (i = 0; i < 99; i++)
4      a2[i] = a[i+1];
5  #pragma omp parallel for lastprivate(x)
6    for (i = 0; i < 99; i++) {
7      x = b[i] + c[i];
8      // a[i] = a[i+1] + x;
9      a[i] = a2[i] + x;
10   }
11   ...
```

再举一例

例如：

```
1   ...
2 #pragma omp parallel for
3   for (i = 1; i < n; i++) {
4     b[i] = b[i] + a[i-1];
5     a[i] = a[i] + c[i];
6   }
7   ...
```

- 思考 1：这段代码有哪种数据依赖?
- 思考 2：如何消除?

消除数据依赖 (4)

- 方法 4：循环倾斜 (loop skewing)

```
1   ...
2   b[1] = b[1] + a[0];
3 #pragma omp parallel for
4   for (i = 1; i < n-1; i++) {
5     a[i] = a[i] + c[i];
6     b[i+1] = b[i+1] + a[i];
7   }
8   a[n-1] = a[n-1] + c[n-1];
9   ...
```

预告：翻转课堂

- 本学期课时较为充足，我打算尝试给有兴趣的同学提供机会上台，也就是目前流行的“翻转课堂”(Flipped Classroom)。
- 具体要求：
  - 1. 提前一周向助教发申请并提供大纲或初步的 slides 供审核；
  - 2. 视情况共安排约 4 人上台，每人讲 50 分钟 (含问答)；
  - 3. 内容要与课程有相关性，可围绕教学，也可围绕研究课题；
  - 4. 讲稿需保持自我完整性，易于理解，没有突兀或戛然而止。
- Benefit：视情况对参与翻转课堂讲解的同学奖励 1-10 分平时分。