

并行与分布式计算基础 III：OpenMP 编程与实践

杨超

chao_yang@pku.edu.cn

2020 秋



内容提纲

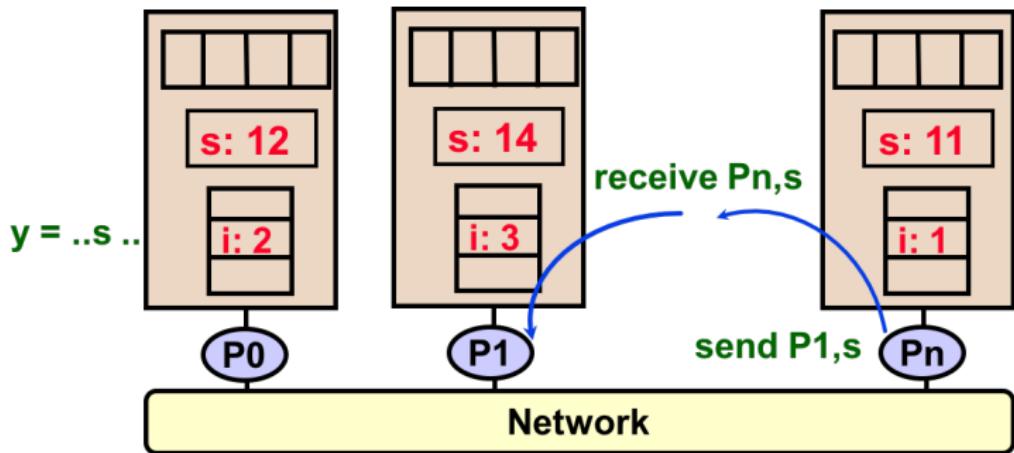
- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

回顾：消息传递

- 机器由若干可以相互传递消息的进程 (process) 构成；
- 每个进程拥有私有的存储空间，进程间无共享存储；
- 进程间的消息传递采用显式的发送/接收 (send/receive) 机制完成；
- 程序往往采用 SPMD (single program multiple data) 方式编写。

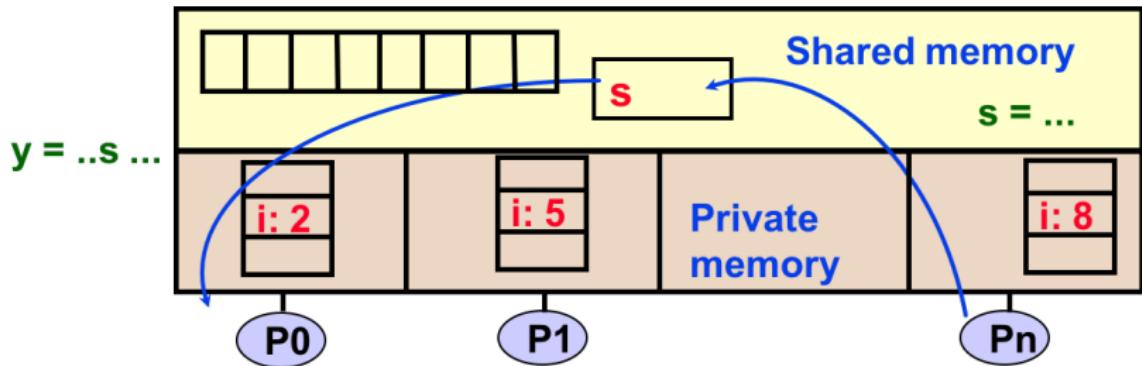


如果仅采用消息传递模式进行编程，合理吗？



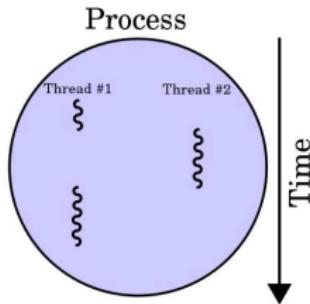
共享内存并行编程

- 程序由一系列线程 (thread) 控制；
- 每个线程都有自己私有的变量；
- 线程之间通过共享变量进行交互：
 - 通过对共享变量的读写进行“隐式”通信；
 - 通过同步机制进行协同。

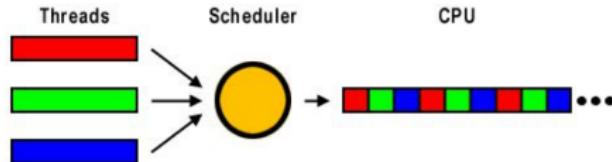


线程与进程的关系

- 线程可以被看作是进程的一部分，一个进程可以开启多个线程；
- 线程继承了进程的资源（比如指令、内存等）；
- 线程相互独立地并发（concurrent）执行；



- 对于单处理器核，多线程可以通过多任务（multi-tasking）亦称为时间切片（time-slicing）的方式由处理器轮流分时执行，此时也称为软件线程（software threads）。



什么是 OpenMP?

OpenMP = Open Multi-Processing

- 是一种支持共享内存并行的应用开发接口 (application programming interface, API) 和规范 (specification);
- 支持多种编程语言、指令集架构和操作系统;
- 由多家计算机厂商组成的非营利组织联合发布，官方网站：
<http://www.openmp.org/> (包含了 OpenMP 相关的接口规范、常见问题、讲座、讨论、发布、日程、会员信息等)。



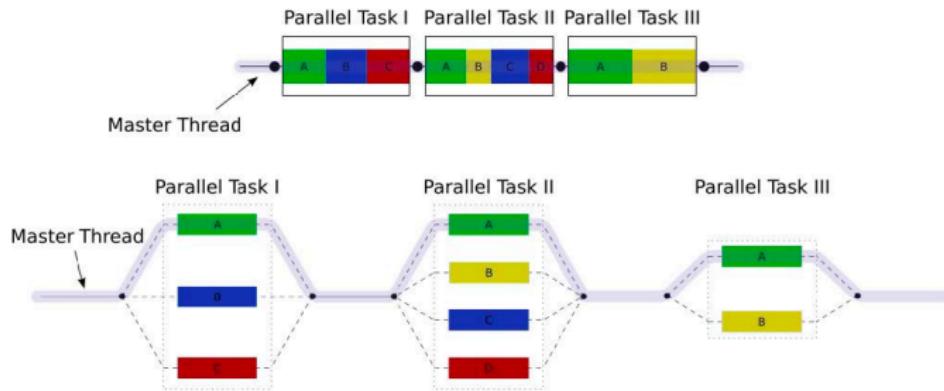
OpenMP 的发展历史

- 1994 年, ANSI X3H5 标准草案 (未采纳);
- 1997 年, 非营利组织 “OpenMP 体系结构审核委员会” (OpenMP Architecture Review Board) 成立, 负责管理和发布;
- 2005 年, C/C++、Fortran 的版本开始合并发布。

Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5

Fork-Join 模式

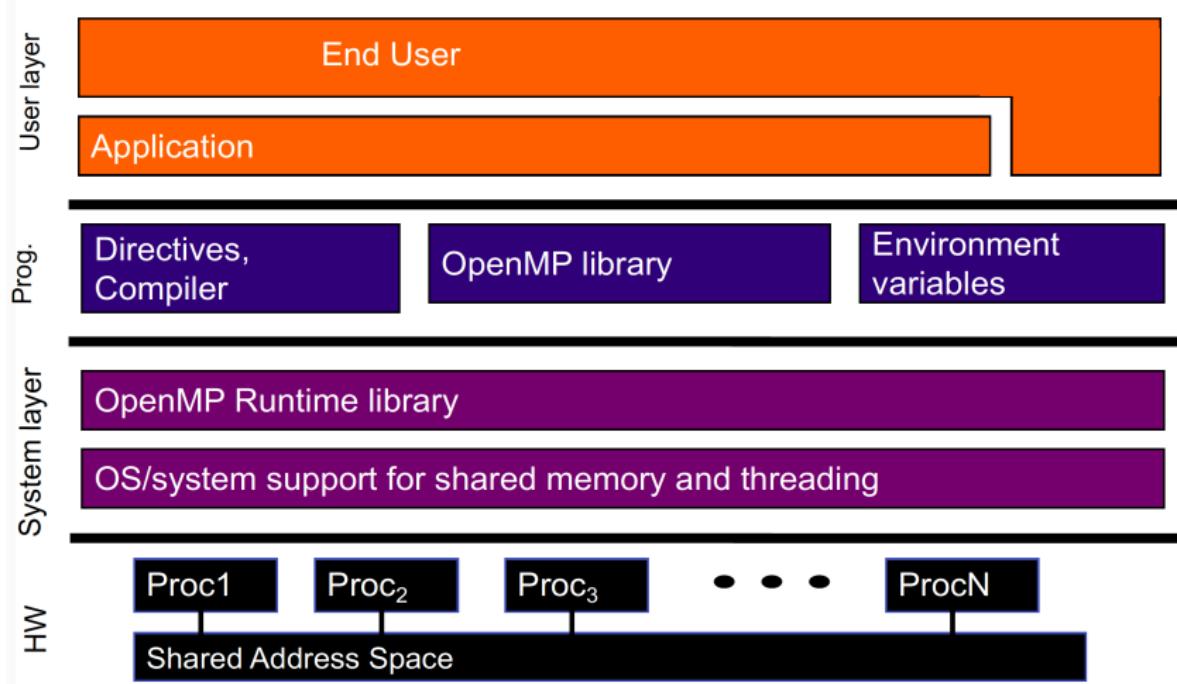
- OpenMP 主要采用 Fork-Join 模式进行并行执行；
- 程序开始时只有一个线程：主线程 (master thread)，编号为 0；
- 主线程仅当进入并行区 (parallel region) 才并行执行：
 - ▶ Fork：主线程创建一组并行线程，编号 $0 \sim n - 1$ ；
 - ▶ 执行：并行线程在并行区中并行执行；
 - ▶ Join：在并行区结尾并行线程进行同步和结束，只保留主线程；
- 并行区的个数，以及每个并行区中的线程数，都可以任意设置。



OpenMP 的特点

- 显式并行 (explicit parallelism):
 - ▶ 由程序员对并行进行控制 (而不是系统自动并行)。
- 数据域 (data scope):
 - ▶ 同一个并行区中的所有线程的数据默认是共享的；
 - ▶ 用户可以指定变量的生存周期以及是否私有。
- 嵌套并行 (nested parallelism):
 - ▶ OpenMP 的 API 允许一个并行区内开启另一个并行区；
 - ▶ 但是部分 OpenMP 库可能不支持。
- 动态线程 (dynamic thread):
 - ▶ 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
 - ▶ 但是部分 OpenMP 库可能不支持。

OpenMP 的基础解决方案



OpenMP API 的三个要素 (1)

运行时库 (run-time library)

主要包括头文件 (omp.h)、库函数的调用和链接。

Fortran	<code>INTEGER FUNCTION OMP_GET_NUM_THREADS()</code>
C/C++	<code>#include <omp.h></code> <code>int omp_get_num_threads(void)</code>

主要用途：

- Setting and querying the number of threads;
- Querying thread ID, ancestor's ID, and thread team size;
- Setting and querying the dynamic threads feature;
- Querying if in a parallel region, and at what level;
- Setting and querying nested parallelism;
- Setting, initializing and terminating locks and nested locks;
- Querying wall clock time and resolution; ...。

OpenMP API 的三个要素 (2)

环境变量 (environment variable)

OpenMP API 预定义了一些环境变量，运行时控制程序的行为。

csh/tcsh	<code>setenv OMP_NUM_THREADS 8</code>
sh/bash	<code>export OMP_NUM_THREADS=8</code>

主要用途：

- Setting the number of threads;
- Specifying how loop iterations are divided;
- Binding threads to processors;
- Enabling/disabling/setting nested parallelism;
- Enabling/disabling dynamic threads;
- Setting thread stack size and wait policy; ...。

OpenMP API 的三个要素 (3)

编译制导语句 (compiler directive)

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 OpenMP 的一些功能，否则，将忽略。

Fortran	<code>!\$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)</code>
C/C++	<code>#pragma omp parallel default(shared) private(beta,pi)</code>

主要用途：

- Spawning a parallel region;
- Dividing blocks of code among threads;
- Distributing loop iterations between threads;
- Serializing sections of code;
- Synchronization of work among threads; ...。

OpenMP 的 21 个常用核心 (Common Cores)

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(None)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

第一个 OpenMP 程序：hello world!

omp_hello.c

```
1 #include <omp.h> // omp header file
2 #include <stdio.h> // standard I/O
3 int main(int argc, char *argv[]){
4     int      nthreads, tid;
5     double   t0, t1;
6     omp_set_num_threads(4);
7     t0 = omp_get_wtime();
8 #pragma omp parallel private(tid)
9 {
10     nthreads = omp_get_num_threads(); // get num of threads
11     tid = omp_get_thread_num(); // get my thread id
12     printf("From thread %d out of %d, Hello World!\n", tid,
13           nthreads);
14 }
15 t1 = omp_get_wtime();
16 printf("Time elapsed is %.f.\nThat's all, folks!\n", t1-t0);
17 return 0;
}
```

程序的编译与运行

- 设置环境变量：

```
$ export OMP_NUM_THREADS=4
```

- 编译：

```
$ gcc omp_hello.c -o hello -fopenmp
```

- 运行 (请正确使用 sbatch 或者 salloc)：

```
$ ./hello
```

运行结果

- 运行结果：

```
From thread 1 out of 4, Hello World!
From thread 0 out of 4, Hello World!
From thread 2 out of 4, Hello World!
From thread 3 out of 4, Hello World!
Time elapsed is 0.000005.
That's all, folks!
```

OpenMP 墙钟时间

- 返回当前线程的时钟时间：

```
double omp_get_wtime(void)
```

- 用法：

```
1     ...
2     t0 = omp_get_wtime();
3     ... // do some works
4     t1 = omp_get_wtime();
5     ...
```



- 返回时钟刻度：

```
double omp_get_wtick(void)
```

设置线程数

- 通过环境变量：

```
$ export OMP_NUM_THREADS=4
```

- 通过库函数 (设置此后所有并行区的线程数)：

```
void omp_set_num_threads(int)
```

- Q1：如果不设置线程数呢？(可以用 unset xxx 清除环境变量)
- Q2：是否可以在并行区内部设置线程数？
- Q3：如果环境变量和程序中设置了不同的线程数呢？

获取线程数

- 通过环境变量：

```
$ echo $OMP_NUM_THREADS  
4
```

- 通过库函数：

```
int omp_get_num_threads(void)
```

- ▶ Q1：可以在串行区执行 `omp_get_num_threads` 吗？
- 获取曾使用过的最大线程数：

```
int omp_get_max_threads(void)
```

- ▶ 可以在任意并行区或者串行区执行，返回此前的最大线程数；
- ▶ 受 `OMP_NUM_THREADS` 及 `omp_set_num_threads` 限制。

获取线程号

- 通过库函数：

```
int omp_get_thread_num(void)
```

- Q1：这个函数可以在串行区执行吗？
- Q2：为什么只能获取、不能设置线程号？
- Q3：为什么不能通过环境变量获取线程号？

练习题

修改 `omp_hello.c` 程序：

- 改变线程数的设置方式；
- 改变线程数的获取方式；
- 改变线程号的获取方式；
- 把计时语句放在并行区内；
- 其他感兴趣的修改。

最小侵害性质 (Minimally Invasive Property)

- OpenMP 提供了内置宏 `_OPENMP`, 帮助判断 OpenMP 是否存在;
- 借助条件编译, 我们可以在不支持 OpenMP 的环境下也能编译并正常运行程序, 例如:

```
1 #ifdef _OPENMP
2 #include <omp.h> // omp header file
3 #else
4 #define omp_set_num_threads(x) 0
5 #define omp_get_num_threads() 1
6 #define omp_get_thread_num() 0
7 ...
8 #endif
9 ...
```

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

OpenMP 制导语句的构造和用法

- 同一类 OpenMP 制导语句称为一种构造 (construct):
 - 比如并行区构造、工作共享构造、任务构造、同步构造等。
- OpenMP 制导语句的用法为:
 - 以 `#pragma omp` 开始;
 - 接着是某一个制导名 (directive-name), 比如 `parallel`;
 - 接着是零至多个从句 (clause), 从句出现的顺序不重要。

```
#pragma omp parallel default(shared) private(beta,pi)
```

识别区 制导名 从句1 从句2

- 注意: 如果一行过长, 换行时行末需要加 “\”。

最基本的 OpenMP 构造：并行区构造

- 用途：划定并行区的范围，并做相关设置，格式：

```
#pragma omp parallel [clause1 clause2 ...]
{
    ...
}
```

- 支持的从句：

```
if (scalar_expression)
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer_expression)
```

控制从句

```
if (scalar_expression)
```

- if 从句：决定是否以并行的方式执行并行区；
 - ▶ 表达式为真（非零）：按照并行方式执行并行区；
 - ▶ 否则：主线程串行执行并行区；
 - ▶ 此从句在每个制导语句中最多仅能出现一次。

```
num_threads (integer_expression)
```

- num_threads 从句：指定并行区的线程数；
 - ▶ 此从句在每个制导语句中最多仅能出现一次。

线程数的确定

按照优先级从低到高，并行区中的线程数按照下面的顺序确定：

- 系统默认 (一般是可用的处理器核数);
- OMP_NUM_THREADS 环境变量设定;
- omp_set_num_threads 库函数设定;
- num_threads 从句设定;
- if 从句。

数据域从句

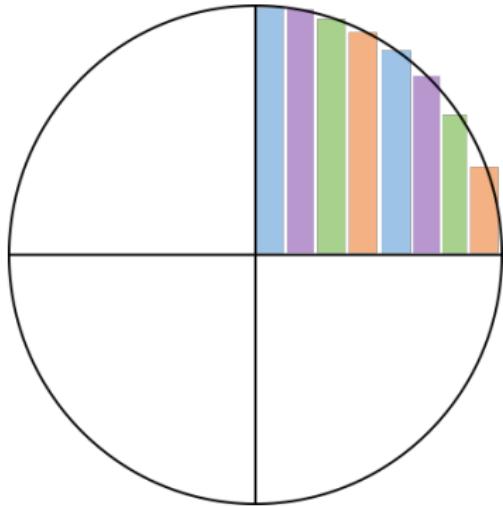
```
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
```

- `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。
- `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` 定义的规约操作。
- `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。
- `copyin` 从句：以后介绍。

示例程序：计算 π

- 计算依据：单位圆的面积。



$$\begin{aligned}\pi &= 4 \int_0^1 \sqrt{1 - x^2} dx \\ &\approx 4h \sum_{i=0}^{N-1} \sqrt{1 - x_i^2},\end{aligned}$$

where

$$x_i = (i + \frac{1}{2})h, \quad h = \frac{1}{N}.$$

- 并行策略：round-robin。

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 #define PI25DT 3.141592653589793238462643
6
7 int main(int argc, char *argv[]){
8     int      nthreads, tid, n, i;
9     double   pi, h, x, t0, t1;
10
11    t0 = omp_get_wtime();
12    n = 10000000;
13    h = 1.0 / (double) n;
14    pi = 0.0;
15 #pragma omp reduction default(shared) \
16     private(tid, i, x) reduction(+,pi)
17 {
18     nthreads = omp_get_num_threads();
19     tid = omp_get_thread_num();
20     for (i = tid + 1; i <= n; i += nthreads) {
21         x = h * ((double)i - 0.5);
```

```
22     pi += 4.0 * h * sqrt(1.-x*x);
23 }
24 }
25 t1 = omp_get_wtime();
26 printf(" Number of threads = %d\n", nthreads);
27 printf(" pi is approximately %.16f\n", pi);
28 printf(" Error is %.16f\n", fabs(pi-PI25DT));
29 printf(" Wall clock time = %f\n", t1-t0);
30
31 return 0;
32 }
```

- 编译:

```
$ gcc omp_cpi.c -o cpi -fopenmp -lm
```

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ ./cpi
Number of threads = 8
pi is approximately 3.1415926536006422
Error is 0.0000000000108491
Wall clock time = 0.036953
```

练习

- 设置不同的线程数，测试、比较和分析结果；
- 设置不同的 n ，测试、比较和分析结果；
- 把 `nthreads` 设置为 `private`；
- 把 `default(shared)` 改为 `default(None)`；
- 把第 14 行 `pi = 0.0` 改为 `pi = 1.0`；
- 把 round-robin 并行策略改为其他策略。

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

回忆计算 π 的示例程序

omp_cpi.c

```
1 ...  
2 pi = 0.0;  
3 #pragma omp parallel default(shared) \  
4     private(tid, i, x) reduction(+:pi)  
5 {  
6     nthreads = omp_get_num_threads();  
7     tid = omp_get_thread_num();  
8     for (i = tid + 1; i <= n; i += nthreads) {  
9         x = h * ((double)i - 0.5);  
10        pi += 4.0 * h * sqrt(1.-x*x);  
11    }  
12 }  
13 ...
```

- 思考：

- ▶ 这种基于循环的“数据”并行十分常见！
- ▶ 是否可以采用更简单的方式实现？

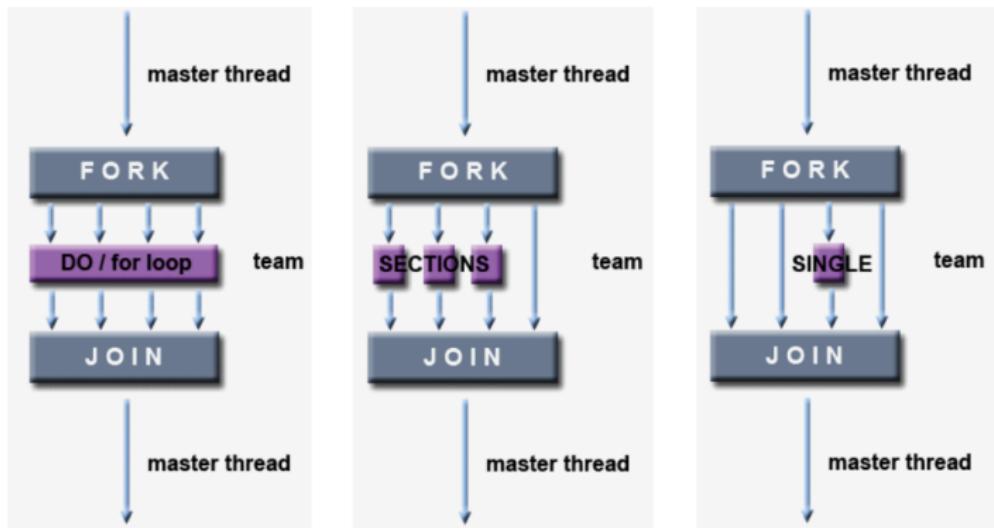
工作共享构造 (work-sharing construct)

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

- 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 循环的格式

- 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。
- 思考：下面哪些用法有问题？

1: `for (; i<=10; ++i)`

2: `for (k=1000; k>8; k--)`

3: `for (i=1; i<=j; j=j-10)`

4: `for (z=1; 100>z; z=2*z)`

for 构造在并行区中的用法

- 如下并行区内的 code1()-code6() 分别怎么执行?

```
#pragma omp parallel
{
    code1();
#pragma omp for
    for (i=1; i<=N; i++) {
        code2();
    }
    for (j=1; j<=M; j++) {
        code3();
    }
    code4();
#pragma omp for
    for (m=L; m>=1; --m) {
        code5();
    }
    code6();
}
```

parallel for 构造

- 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();
}
```

- 练习：

- 把 omp_cpi.c 中的 n 改为 11111111；
- 使用 for 构造修改 omp_cpi.c；
- 使用 parallel for 构造修改 omp_cpi.c；
- 将新程序保存为 omp_cpi2.c.

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。

规约操作的类型和初始值

Valid Operators and Initialization Values			
Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	.true. / 1
Logical OR	.or.		.false. / 0
AND bitwise	iand	&	all bits on / 1
OR bitwise	ior		0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

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。

示例：私有变量

omp_private.c

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]){
5     int i, k;
6
7     printf("Test:\n");
8     k = 100;
9     printf(" first k = %d\n", k);
10    #pragma omp parallel for private(k)
11    for (i = 0; i < 10; i++) {
12        k = i;
13        printf(" private k = %d\n", k);
14    }
15    printf(" last k = %d\n", k);
16    return 0;
17 }
```

思考和练习

- 循环体中修改 `k` 的值，会影响主线程的结果吗？
- `k = i` 可以改为 `k += i` 吗？
- 将 `private(k)` 改为别的私有变量类型，会怎样？

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。

static 静态调度

- 默认 $\text{chunk} = n/t$, 按循环起止均匀分配;
- 调整 chunk 可以改变静态线程分配的策略;
- $\text{chunk}=1$, 相当于 round-robin。

```
for (i=0; i<=11; i++)
```

schedule(static, 4)

Thread 0 : 0,1,2,3
Thread 1 : 4,5,6,7
Thread 2 : 8,9,10,11

schedule(static, 2)

Thread 0 : 0,1,6,7
Thread 1 : 2,3,8,9
Thread 2 : 4,5,10,11

schedule(static, 1)

Thread 0 : 0,3,6,9
Thread 1 : 1,4,7,10
Thread 2 : 2,5,8,11

示例：线程调度

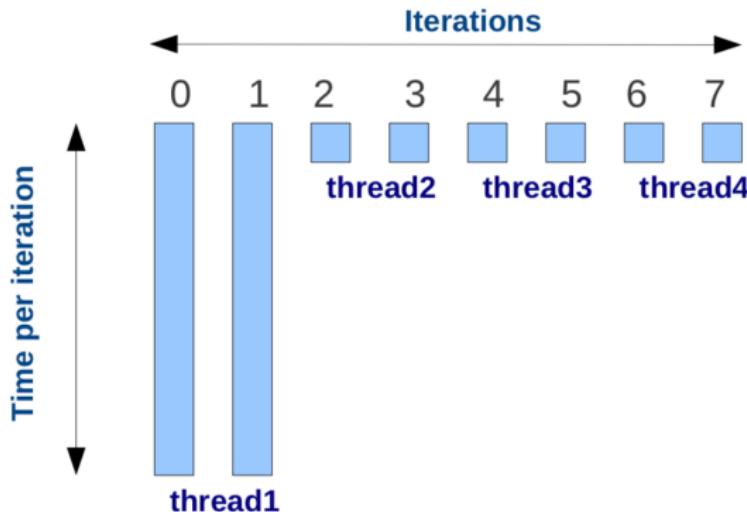
omp_schedule.c

```
1 ...
2 #define n 12
3 ...
4 #pragma omp parallel num_threads(4)
5 {
6 #pragma omp for schedule(static)
7     for (i = 0; i < n; i++) {
8         ...
9     }
10 }
11 ...
```

- 练习：
 - ▶ 尝试给 static 加上不同的 chunk 值。

为什么需要进行动态调度?

- 每个迭代步的耗时可能不平均!



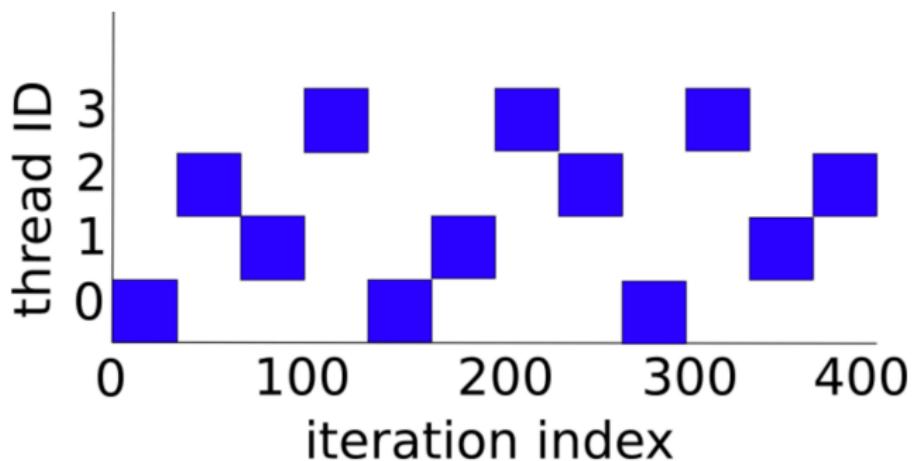
- 比如，计算二重积分：

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

```
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    ...
```

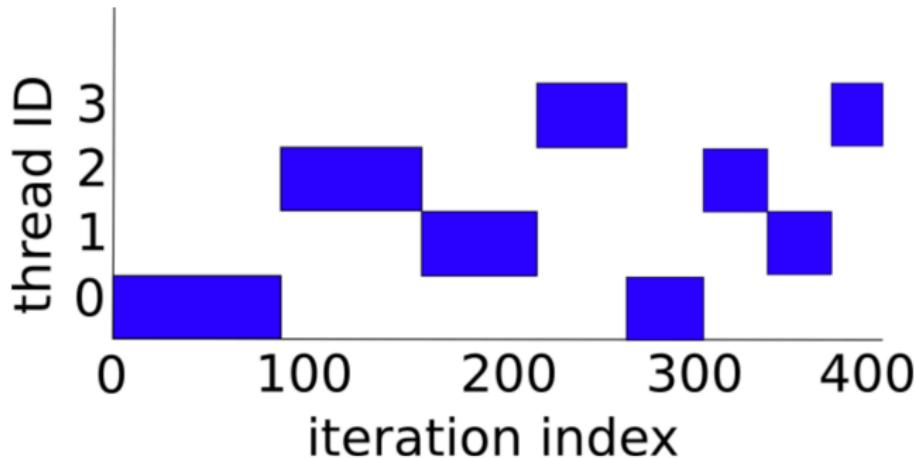
dynamic 动态调度

- 根据线程空闲情况，对工作进行动态分配：
 - 默认 `chunk=1`, 动态分配的任务粒度为 1;
 - 增大 `chunk` 可以增大任务的粒度;
 - 调度开销不容忽视。



guided 动态调度

- 为了减少调度开销，动态分配任务的粒度逐步减小：
 - 调整策略：粒度 = 剩余迭代次数 / 线程数；
 - 最小粒度为 chunk，默认为 1。



练习

- 修改 `omp_schedule.c` 中的 `schedule` 从句：
 - ▶ 测试 `dynamic` 和不同的 `chunk` 值；
 - ▶ 测试 `guided` 和不同的 `chunk` 值；
 - ▶ 测试 `runtime` 并由环境变量 `OMP_SCHEDULE` 确定；
 - ▶ 测试 `auto`，由系统自行确定。

计算 π : 采用并行区构造

omp_cpi.c

```
1   ...
2   t0 = omp_get_wtime();
3   n = 11111111;
4   h = 1.0 / (double) n;
5   pi = 0.0;
6 #pragma omp parallel default(shared) \
7   private(tid, i, x) reduction(+:pi)
8 {
9     nthreads = omp_get_num_threads();
10    tid = omp_get_thread_num();
11    for (i = tid + 1; i <= n; i += nthreads) {
12      x = h * ((double)i - 0.5);
13      pi += 4.0 * h * sqrt(1.-x*x);
14    }
15  }
16  t1 = omp_get_wtime();
17  ...
```

计算 π : 采用循环构造

omp_cpi2.c

```
1   ...
2   t0 = omp_get_wtime();
3   n = 11111111;
4   h = 1.0 / (double) n;
5   pi = 0.0;
6 #pragma omp parallel for default(shared) \
7     private(x) reduction(+:pi)
8   for (i = 1; i <= n; i++) {
9     x = h * ((double)i - 0.5);
10    pi += 4.0 * h * sqrt(1.-x*x);
11  }
12  t1 = omp_get_wtime();
13  ...
```

练习

- 尝试采用不同的调度方式，比较性能：
 - ▶ `static` 静态调度；
 - ▶ `dynamic` 动态调度；
 - ▶ `guided` 动态调度；
 - ▶ `runtime` 运行时设定 (`OMP_SCHEDULE`)。

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 `for` 构造应用于多重循环的第 1 至 n 重。
 - ▶ 涉及的循环间必须没有依赖关系；
 - ▶ 相当于对第 1 至 n 重循环做了合并，当作了一个循环；
 - ▶ 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 }
```

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

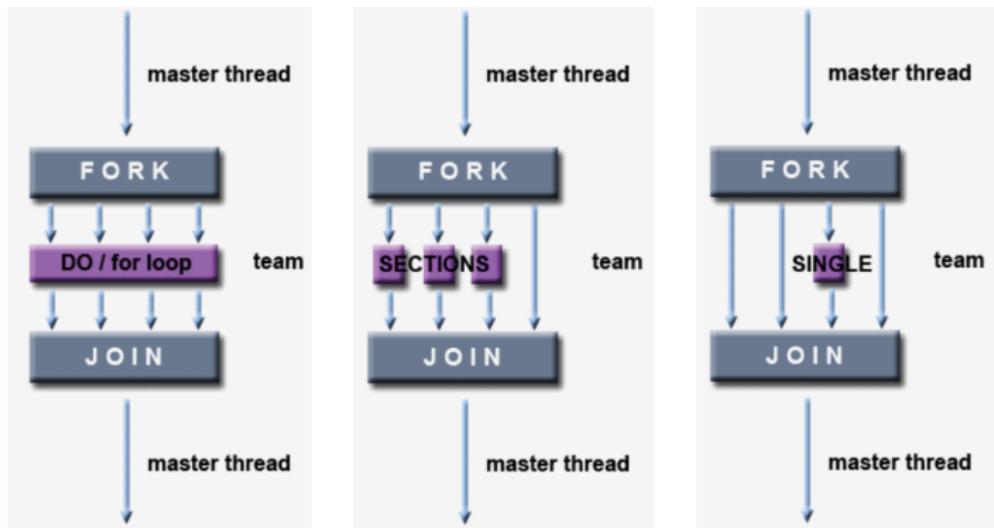
工作共享构造 (work-sharing construct)

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



sections 构造

- 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();
}
```

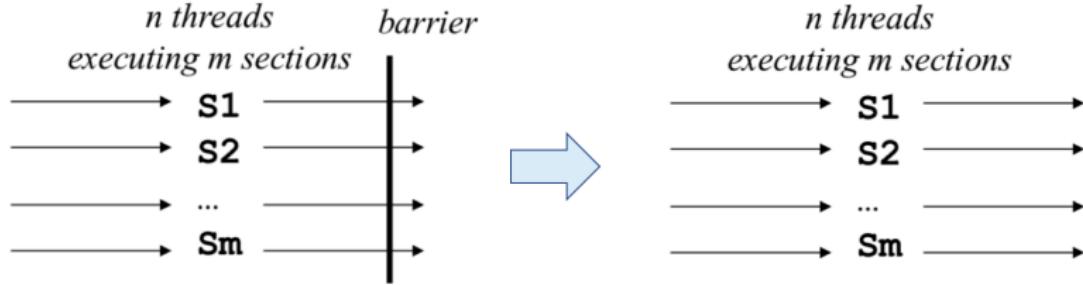
- 思考：为什么 `single` 构造不能与并行区合并？

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

	<i>parallel</i>	<i>for</i>	<i>parallel for</i>	<i>sections</i>	<i>parallel sections</i>	<i>single</i>
<code>if</code>	•		•		•	
<code>num_threads</code>	•		•		•	
<code>default</code>	•		•		•	
<code>copyin</code>	•		•		•	
<code>shared</code>	•	•	•		•	
<code>private</code>	•	•	•	•	•	•
<code>reduction</code>	•	•	•	•	•	
<code>firstprivate</code>	•	•	•	•	•	•
<code>lastprivate</code>	•		•	•	•	
<code>schedule</code>		•	•			
<code>ordered</code>		•	•			
<code>collapse</code>			•			
<code>nowait</code>		•		•		•

内容提纲

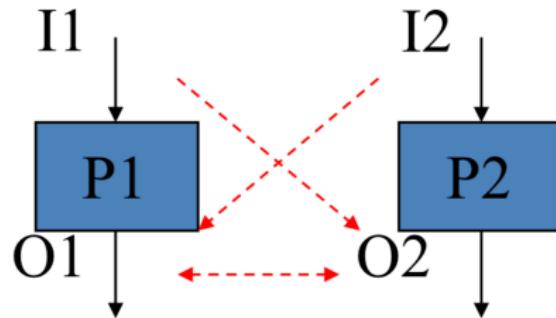
- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

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)

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

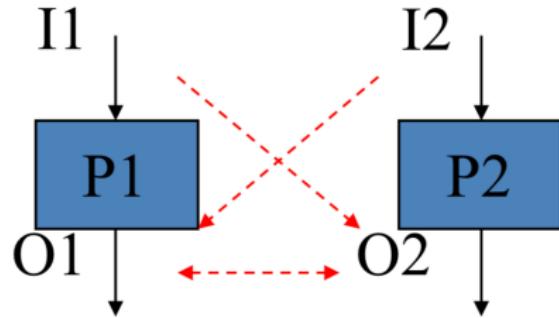
- 推论：当任两个程序 P_i 和 P_j 都满足 Bernstein 条件时，一定有

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

- 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

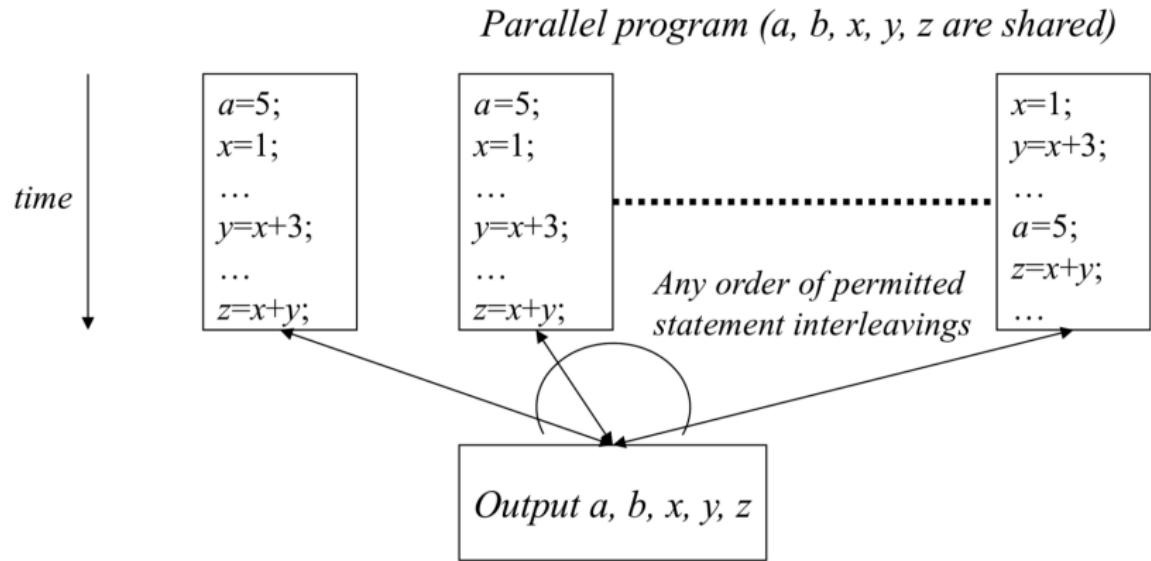
- 竞争条件 (race condition): 指的是并行程序的执行结果具有随机性，依赖于某些事件的发生顺序。
- 数据依赖 (data dependency): 指的是两个或两个以上的程序访问同一片内存，且至少有一个程序执行了写操作。



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： $variable = variable \text{ op} ...$, 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): $RAW = \text{Read After Write}$;
- 2. 反依赖 (anti-dependence): $WAR = \text{Write After Read}$;
- 3. 输出依赖 (output dependence): $WAW = \text{Write After Write}$ 。

<i>independent</i>	<i>RAW</i>	<i>WAR</i>	<i>WAW</i>
$P_1: \mathbf{A} = \mathbf{x} + \mathbf{y};$ $P_2: \mathbf{B} = \mathbf{x} + \mathbf{z};$	$P_1: \mathbf{A} = \mathbf{x} + \mathbf{y};$ $P_2: \mathbf{B} = \mathbf{x} + \mathbf{A};$	$P_1: \mathbf{A} = \mathbf{x} + \mathbf{B};$ $P_2: \mathbf{B} = \mathbf{x} + \mathbf{z};$	$P_1: \mathbf{A} = \mathbf{x} + \mathbf{y};$ $P_2: \mathbf{B} = \mathbf{x} + \mathbf{z};$
$I_1 \cap O_2 = \emptyset$	$I_1 \cap O_2 = \emptyset$	$I_1 \cap O_2 = \{\mathbf{B}\}$	$I_1 \cap O_2 = \emptyset$
$I_2 \cap O_1 = \emptyset$	$I_2 \cap O_1 = \{\mathbf{A}\}$	$I_2 \cap O_1 = \emptyset$	$I_2 \cap O_1 = \emptyset$
$O_1 \cap O_2 = \emptyset$	$O_1 \cap O_2 = \emptyset$	$O_1 \cap O_2 = \emptyset$	$O_1 \cap O_2 = \{\mathbf{A}\}$

- 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 ...
```

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

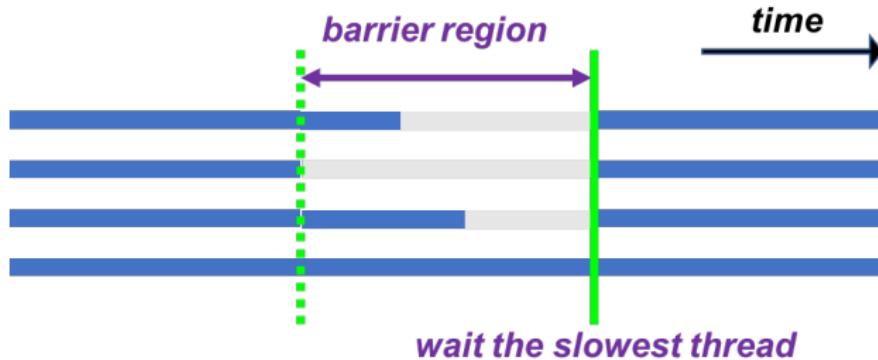
同步构造 (synchronization construct)

- `barrier` 构造: 栅栏同步;
- `single` 构造: 单线程执行, 有同步;
- `master` 构造: 主线程执行, 无同步;
- `ordered` 构造: 按循环顺序执行;
- `critical` 构造: 各线程依次执行, 程序片段;
- `atomic` 构造: 各线程依次执行, 单一指令.

barrier 构造

- 在并行区中特定位置显式加入栅栏同步：

```
#pragma omp barrier
```



- 例如：

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

wait !

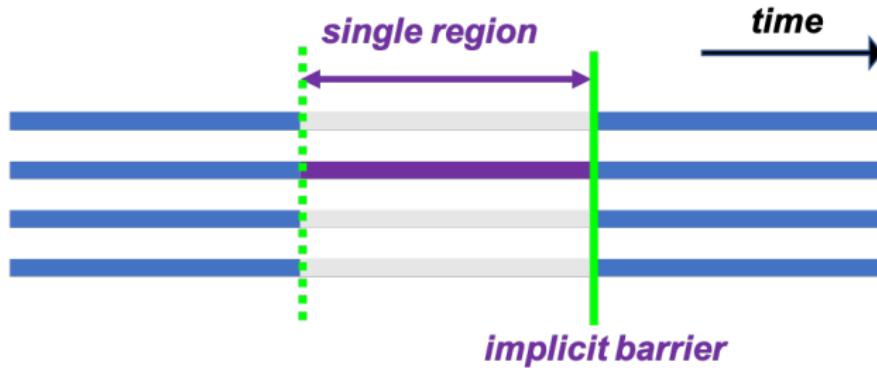
barrier

```
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```

single 构造

- 对并行区内的一段代码单线程执行，有同步（可用 nowait 去掉）：

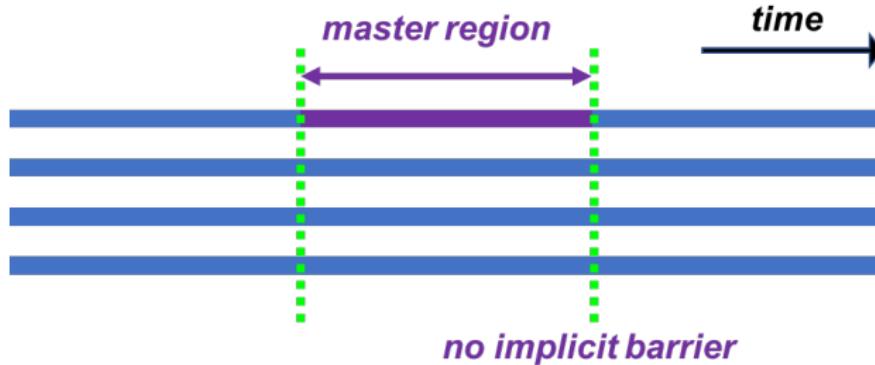
```
#pragma omp single [clause1 clause2 ...]  
{  
    code();  
}
```



master 构造

- 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)

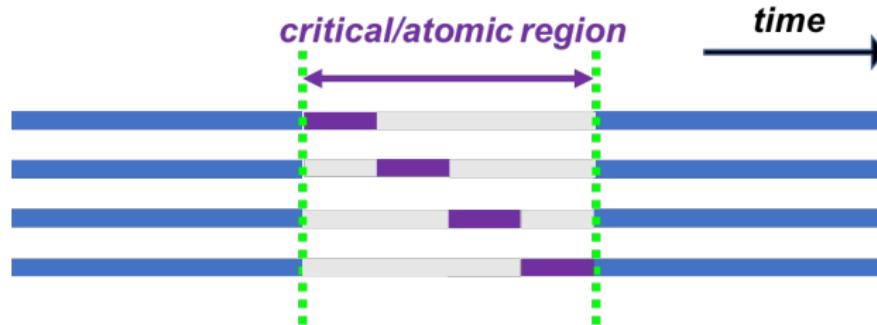
```
#pragma omp master
{
    code();
}
```



critical 构造

- 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();  
}
```



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)；
- 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；
- 在 OpenMP 中，避免竞争条件发生的主要手段有：
 - ▶ 使用 `critical` 构造；
 - ▶ 使用 `atomic` 构造；
 - ▶ 使用 `reduction` 从句等。

- 例如，如下操作不是线程安全的：

```
1 #pragma omp parallel for shared(sum,a)
2   for (int i=0; i<n; i++) {
3     a[i] = ...
4     sum += a[i];
5 }
```

- 对此，可做下述修改，避免竞争条件发生：

```
1 #pragma omp parallel for shared(sum,a)
2   for (int i=0; i<n; i++) {
3     a[i] = ...
4     #pragma omp critical
5       sum += a[i];
6 }
```

atomic 构造

- 可认为是一种特殊的 critical 构造，对单个特定格式的语句或语句组中某个变量进行原子操作，用法（如对 x 原子操作）：

```
#pragma omp atomic read | capture
    something = x;
#pragma omp atomic write | capture
    x = something;
#pragma omp atomic [ update | capture ]
    x = x binop something;
#pragma omp atomic capture
{ x = x binop something; anotherthing = x; }
```

- 详细用法可参见 OpenMP 相关技术手册。

示例：同步构造

omp_sync.c

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]){
5     int nt, tid;
6     int i, a[10], sum = 0;
7
8 #pragma omp parallel private(i, tid)
9 {
10     nt = omp_get_num_threads();
11     tid = omp_get_thread_num();
12     for (i = tid; i < 10; i += nt) {
13         a[i] = i;
14     }
15 #pragma omp barrier
16 #pragma omp single
17     for (i = 0; i < 10; i++) {
18         printf(" tid = %d/%d, a[%d] = %d\n", tid, nt, i, a[i]);
```

```
19     }
20 #pragma omp for
21     for (i = 0; i < 10; i++) {
22 #pragma omp critical (summation)
23     sum += a[i];
24 }
25 #pragma omp master
26     printf(" sum = %d\n", sum);
27 }
28 return 0;
29 }
```

- 运行结果：

```
tid = 3/4, a[0] = 0
tid = 3/4, a[1] = 1
tid = 3/4, a[2] = 2
tid = 3/4, a[3] = 3
tid = 3/4, a[4] = 4
tid = 3/4, a[5] = 5
tid = 3/4, a[6] = 6
tid = 3/4, a[7] = 7
tid = 3/4, a[8] = 8
tid = 3/4, a[9] = 9
sum = 45
```

- 练习：尝试修改代码中的各种同步构造，测试结果。

程序的遗孤 (orphaning)

- 并行区的作用范围
 - ▶ 静态范围 (static extent): 并行区直接影响的代码段;
 - ▶ 动态范围 (dynamic extent): 并行区间接影响的代码，例如在并行区内被调用的函数.
- 遗孤 (orphaning): 工作共享和同步构造被放在并行区静态范围外
 - ▶ 如果在动态范围之内，等同于非遗孤情况；
 - ▶ 否则，制导语句不起作用.

```
(void) dowork(); !- Sequential FOR  
  
#pragma omp parallel  
{  
    (void) dowork(); !- Parallel FOR  
}
```

```
void dowork()  
{  
#pragma omp for  
    for (i=0;.....)  
    {  
        :  
    }  
}
```

- sections 构造不支持遗孤.

练习

- 修改 `omp_sync.c`, 求和部分的代码用如下函数:

omp_sync.c

```
1 int compute_sum(int *pt, int m) {  
2     int i, sum = 0;  
3 #pragma omp for  
4     for (i = 0; i < m; i++) {  
5         sum += pt[i];  
6     }  
7     return sum;  
8 }
```

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

动态线程

- 动态线程：系统动态选择并行区的线程数（默认：一般为关闭）。
- 打开/关闭动态线程
 - ▶ 库函数：

```
void omp_set_dynamic(int flag)
```

- ▶ 环境变量：
- 检查动态线程是否打开

```
export OMP_DYNAMIC=true
```

- ▶ 库函数：

```
int omp_get_dynamic (void)
```

- 一个小例子：

- ▶ flag 为 0：并行区开启 10 个线程；
- ▶ flag 非 0：并行区开启 1-10 个线程（系统决定）。

```
1   ...
2   omp_set_dynamic(flag);
3 #pragma omp parallel num_threads(10)
4 {
5     /* do work here */
6 }
7 ...
```

嵌套并行 (nested parallelism)

- 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)
```

- 思考：下面的例子运行结果是什么？

omp_nested.c

```
1 ...
2     omp_set_dynamic(0);
3 #pragma omp parallel num_threads(2)
4 {
5
6     omp_set_nested(1);
7 #pragma omp parallel num_threads(3)
8 {
9 #pragma omp single
10    printf ("Inner: num_thds=%d\n", omp_get_num_threads());
11 }
12
13 #pragma omp barrier
14     omp_set_nested(0);
15 #pragma omp parallel num_threads(3)
16 {
17 #pragma omp single
```

```
18     printf ("Inner: num_thds=%d\n", omp_get_num_threads());
19 }
20
21 #pragma omp barrier
22 #pragma omp single
23     printf ("Outer: num_thds=%d\n", omp_get_num_threads());
24
25 }
26 ...
```

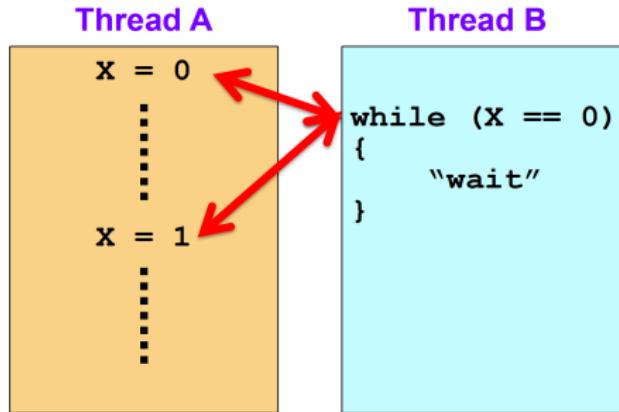
flush 构造

- OpenMP 的松弛一致性 (relaxed consistency):
 - ▶ 数据不仅在内存，还在缓存 (以及寄存器等) 中有多份拷贝；
 - ▶ 实际上 OpenMP 的共享变量在本地缓存中并不随时更新。
- 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)
```

- OpenMP 的一些同步操作隐含包含了 flush，比如：
 - ▶ 并行区入口，critical/ordered 区的入口、出口；
(注意：工作共享构造的入口/出口是不隐含包含 flush 的)
 - ▶ 显式、隐式的 barrier 操作等。

- 举例：



If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)

- 若确需 flush，一般置于共享变量的写操作后，或读操作前；
- 合理的算法设计一般不需要显式的 flush (因为容易出错).

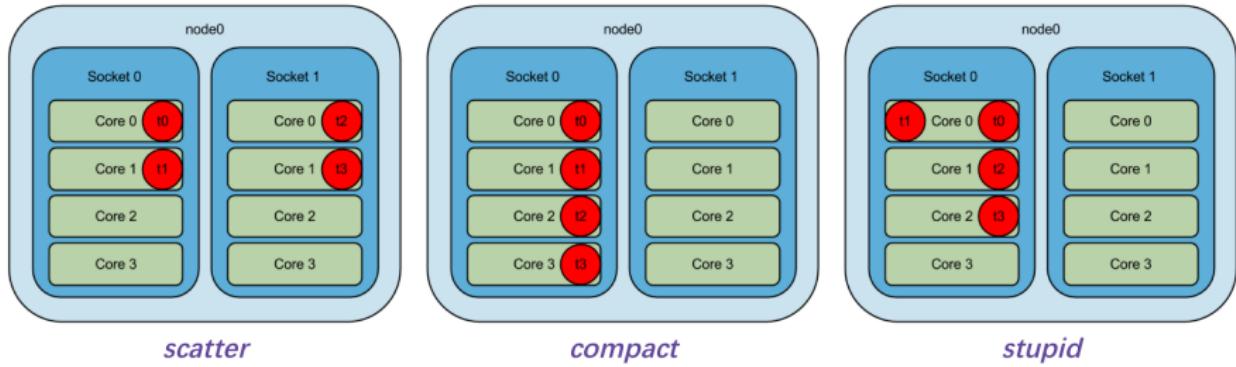
堆栈大小 (stack size)

- 除了主线程，OpenMP 的每个线程的私有变量存储空间受线程的堆栈大小控制。
- OpenMP 标准并不规定具体的堆栈大小，依赖于具体实现：
 - ▶ Intel 编译器：默认大小一般为 4MB；
 - ▶ gcc/gfortran 编译器：默认大小一般为 2MB。
- 如果超出堆栈大小，程序的行为不可控；
- 可以通过环境变量修改默认堆栈大小，比如：

```
export OMP_STACKSIZE=32M  
export OMP_STACKSIZE=8192K
```

线程亲和性 (affinity) 和线程绑定 (binding)

- 线程亲和性：决定了 NUMA 架构的系统上线程在物理计算核心的映射策略。



- 线程绑定：显式确定线程与物理计算核心的对应关系，以提升性能。

- OpenMP3.1 标准仅提供了非常有限的支持，可通过如下方式开启：

```
export OMP_PROC_BIND=TRUE
```

- 如果使用 Intel 编译器，可通过如下方式设置线程亲和性：

```
export KMP_AFFINITY={scatter,compact..}
```

(参考网上教程：<https://software.intel.com/en-us/node/522691>)

- OpenMP 4.5 开始对上述功能提供了较好支持 (参阅相关手册)；
- numactl 工具有时候可以发挥重要作用 (参阅网上教程，如：<http://www.glennclockwood.com/hpc-howtos/process-affinity.html>)。

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

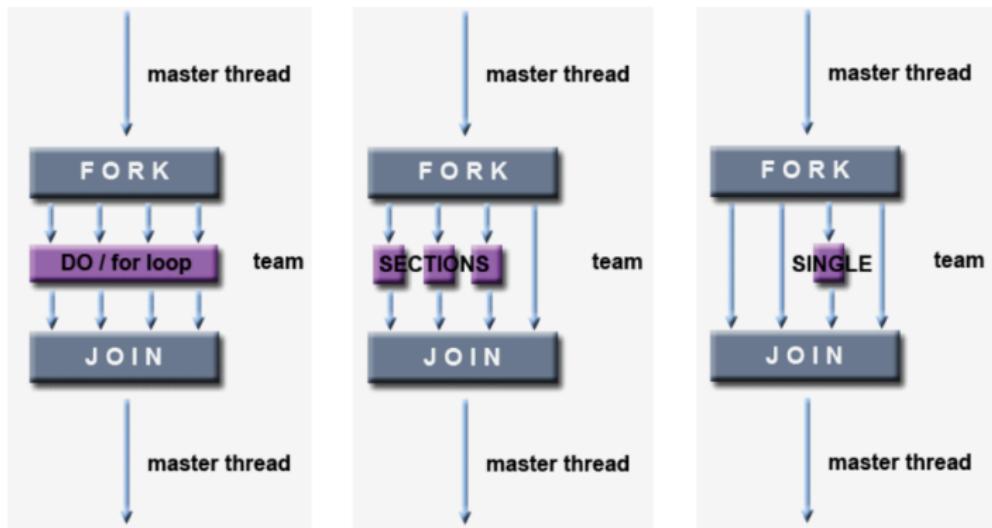
回忆：工作共享构造 (work-sharing construct)

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



OpenMP 工作共享构造的缺陷

- 任务必须可数，比如，下面的任务（如链表、递归等）无法支持：

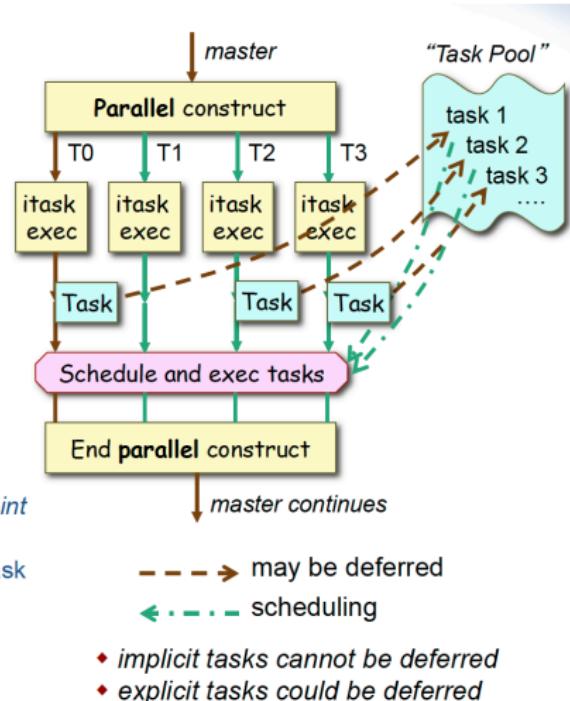
```
#pragma omp parallel
{
    ...
    while (my_pointer != NULL) {
        do_independent_work(my_pointer);
        my_pointer = my_pointer->next;
    } // End of while loop
    ...
}
```

- 只支持任务可数的情况（`for` 循环或者 `section` 区块）；
- 如果不能转换为可数任务，缺乏灵活的任务处理机制；
- OpenMP3.0 开始支持的任务并行是一个有益补充。

OpenMP 的任务并行 (task parallelism)

- 显式定义一系列可执行的任务及其相互依赖关系，通过任务调度的方式多线程动态执行，支持任务的延迟执行 (deferred execution)。

- Starts with the *master* thread
- Encounters a **parallel** construct
 - Creates a team of threads, *id 0* for the *master* thread
 - Generates implicit tasks, one per thread
 - Threads in the team executes implicit tasks
- Encounters a worksharing construct
 - Distributes work among threads (or implicit tasks)
- Encounters a **task** construct
 - Generates an explicit task
 - Execution of the task could be deferred
- Execution of explicit tasks
 - Threads execute tasks at a *task scheduling point* (such as **task**, **taskwait**, **barrier**)
 - Thread may switch from one task to another task
- At the end of a **parallel** construct
 - All tasks complete their execution
 - Only the *master* thread continues afterwards



OpenMP 任务构造

- 定义任务：

```
#pragma omp task [clause1 clause2]
```

```
...
```

- 支持的从句：

```
if (scalar expression)
final (scalar expression)
untied
default (shared | none)
mergeable
private (list)
firstprivate (list)
shared (list)
```

- 完成任务 (含子任务)

- 自动完成：在程序的显式或者隐式同步点；
- 手动完成：

```
#pragma omp taskwait
```

```
...
```

- 变量的数据域

- 并行区中的共享变量：在 task 区中默认也为共享；
- 并行区中的私有变量：在 task 区中默认为 firstprivate；
- task 区中的其他变量：默认为私有。

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private

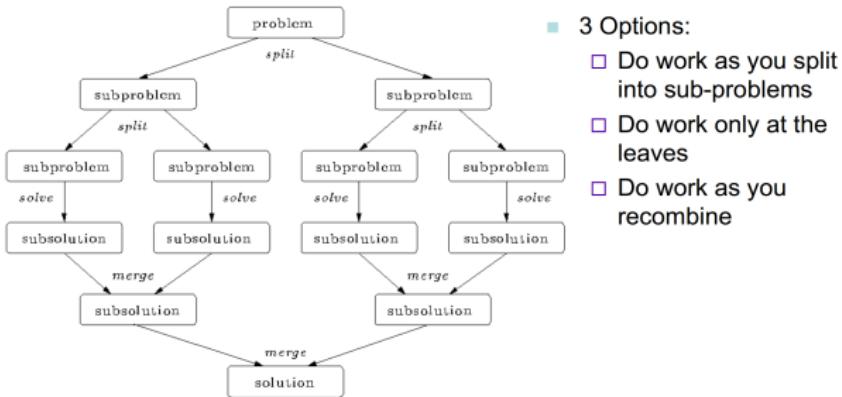
例子：计算斐波那契数

- 斐波那契数 (Fibonacci Number):

$$F(n) = \begin{cases} n, & \text{if } n = 0, 1, \\ F(n - 1) + F(n - 2), & \text{otherwise.} \end{cases}$$

- 分而治之 (Divide and Conquer):

Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



- 递归实现：创建一个二叉任务树，从叶子结点开始进行并行执行。

```
int main()
{
    int res, n=45;
    #pragma omp parallel
    {
        #pragma omp single
        res = fib(n);
    }
    printf("fib(%d)=%d\n",
           n,res);
}
```

```
int fib(int n)
{
    int x, y;
    if (n < 2) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return(x+y);
}
```

Explicit tasks with proper data sharing attributes

Ensure calculations for x and y are done and storage does not disappear

Single thread generates tasks, but multiple threads execute tasks

- 修改一下要求：求第 0 至 n 个数。

omp_fib2.c

```
1 uint64_t fib(int n) {  
2     uint64_t x, y, res;  
3     if      (n < 2)  res = n;  
4     else {  
5         #pragma omp task shared(x)  
6             x = fib(n-1);  
7         #pragma omp task shared(y)  
8             y = fib(n-2);  
9         #pragma omp taskwait  
10            res = x + y;  
11    }  
12    a[n] = res;  
13    return a[n];  
14 }
```

- 思考：为什么性能不好？

- 任务并行的额外开销 (overhead): 任务调度 (task scheduling)。
- 任务粒度 (granularity): 每个任务的“大小”
 - ▶ 粒度太大: 负载不平衡;
 - ▶ 粒度太小: 调度开销过大。
- 思考 1: 如何判断调度开销是否过大?
- 答案: 试一下串行运行, 比较时间.
- 思考 2: 如何减少上述程序的调度开销?
- 答案: 增大任务粒度, 在 n 过小时不再生产任务.
- 调整了任务粒度后, 感觉还是太慢?

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

持久 (persistent) 变量的线程私有化

- 持久 (persistent) 变量：一般指生存周期为整个程序的变量数据，例如全局变量、静态变量等。
- 如果希望每个线程拥有自己的持久变量，并且可以随心所欲地在不同线程间传递各自持久变量的值，怎么办？
- OpenMP 的线程私有型变量提供了上述机制：
 - ▶ `threadprivate` 构造提供了持久变量的私有化机制；
 - ▶ `copyin` 从句提供了将主线程的 `threadprivate` 变量的值传递给其他线程的机制；
 - ▶ `copyprivate` 从句提供了将某线程的 `threadprivate` 变量的值广播给其他线程的机制。

threadprivate 型变量

- `threadprivate` 构造：将持久变量置为线程私有类型

```
#pragma omp threadprivate (list)
```

- 对全局变量：必须置于全局变量声明列表之后并在被首次使用之前，否则不起作用；
- 对静态变量：必须置于 `static` 变量声明列表之后并在被首次使用之前，否则不起作用.
- 注意：与 `private` 类型变量的最大差别是，`threadprivate` 型变量的值可以跨并行区有效 (前提是动态线程关闭，并且每个并行区线程数一致).
- 思考：下面的程序运行结果是什么？

omp_threadprivate.c

```
1 ...
2 int a = 0, b = 0;
3 #pragma omp threadprivate(a)
4
5 int compare(int x) {
6     static int n = 2;
7 #pragma omp threadprivate(n)
8     if (n < x) n = x;
9     return n;
10}
11
12 int main(int argc, char *argv[]){
13     int tid, c, d;
14     omp_set_dynamic(0);
15     printf("1st Parallel Region:\n");
16 #pragma omp parallel num_threads(4) private(tid,b,c,d)
17     {
```

```
18     tid = omp_get_thread_num();
19     a = tid + 1;
20     b = tid + 2;
21     c = compare(a);
22     d = compare(b);
23     printf("Thread %d: a,b,c,d = %d %d %d %d\n",tid,a,b,c,d);
24 }
25 printf("Serial Region: a,b = %d %d\n",a,b);
26 printf("2nd Parallel Region:\n");
27 #pragma omp parallel num_threads(4) private(tid,c,d)
28 {
29     tid = omp_get_thread_num();
30     c = compare(a + 2);
31     d = compare(b + 3);
32     printf("Thread %d: a,b,c,d = %d %d %d %d\n",tid,a,b,c,d);
33 }
34 ...
```

- 运行结果 (请正确使用 sbatch 或者 salloc):

```
$ ./threadprivate
1st Parallel Region:
Thread 0: a,b,c,d = 1 2 2 2
Thread 1: a,b,c,d = 2 3 2 3
Thread 2: a,b,c,d = 3 4 3 4
Thread 3: a,b,c,d = 4 5 4 5
Serial Region: a,b = 1 0
2nd Parallel Region:
Thread 2: a,b,c,d = 3 0 5 5
Thread 1: a,b,c,d = 2 0 4 4
Thread 0: a,b,c,d = 1 0 3 3
Thread 3: a,b,c,d = 4 0 6 6
```

全局或静态变量的传递与广播

- copyin 从句用于将主线程的 threadprivate 变量的值传递给其他线程，仅能用于并行区构造的初始化：

```
#pragma omp parallel [for | sections] copyin(list)
{ ... }
```

- copyprivate 从句用于将某线程的 threadprivate 变量的值广播给其他线程，仅能用于 single 构造，并在其出口处起作用：

```
#pragma omp single copyprivate(list)
{ ... }
```

- 思考：下面的程序运行结果是什么？

omp_copyprivate.c

```
1 ...
2 int counter = 0;
3 #pragma omp threadprivate(counter)
4
5 int increment_counter(){
6     return(++counter);
7 }
8
9 int main(int argc, char *argv[]){
10    int tid, c;
11    omp_set_dynamic(0);
12    omp_set_num_threads(4);
13    printf("1st Parallel Region:\n");
14 #pragma omp parallel private(tid,c)
15 {
16     tid = omp_get_thread_num();
17 #pragma omp single copyprivate(counter)
```

```
18     counter = 50 + tid;
19     c = increment_counter();
20     printf("ThreadId: %d, count = %d\n", tid, c);
21 #pragma omp barrier
22     counter = 100 + tid;
23     c = increment_counter();
24     printf("ThreadId: %d, count = %d\n", tid, c);
25 }
26 printf("2nd Parallel Region:\n");
27 #pragma omp parallel private(tid,c) copyin(counter)
28 {
29     tid = omp_get_thread_num();
30     c = increment_counter();
31     printf("ThreadId: %d, count = %d\n", tid, c);
32 }
33 ...
```

- 运行结果 (请正确使用 sbatch 或者 salloc):

```
$ ./copyprivate
1st Parallel Region:
ThreadId: 3, count = 51
ThreadId: 0, count = 51
ThreadId: 1, count = 51
ThreadId: 2, count = 51
ThreadId: 3, count = 104
ThreadId: 0, count = 101
ThreadId: 1, count = 102
ThreadId: 2, count = 103
2nd Parallel Region:
ThreadId: 1, count = 102
ThreadId: 3, count = 102
ThreadId: 0, count = 102
ThreadId: 2, count = 102
```

并行区与工作共享构造的从句汇总

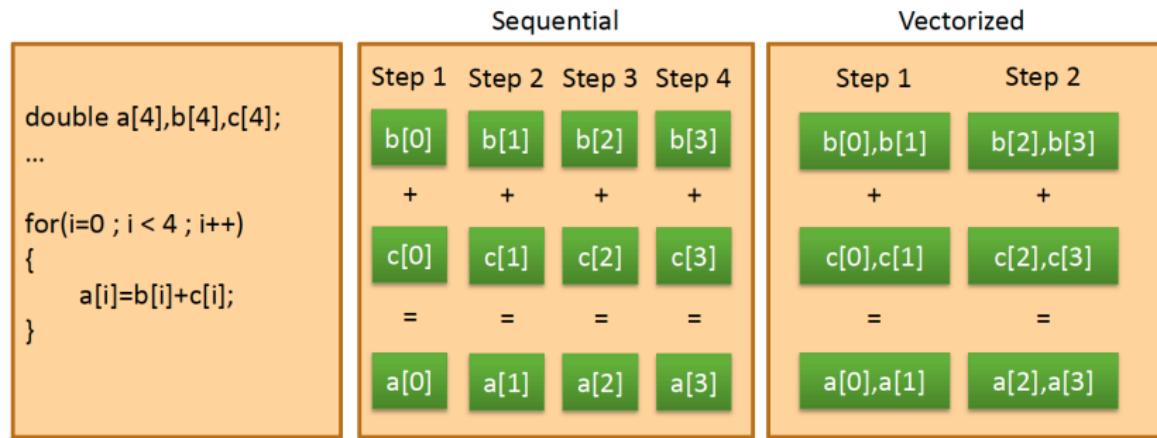
	<i>parallel</i>	<i>for</i>	<i>parallel for</i>	<i>sections</i>	<i>parallel sections</i>	<i>single</i>
<code>if</code>	●		●		●	
<code>num_threads</code>	●		●		●	
<code>default</code>	●		●		●	
<code>shared</code>	●	●	●		●	
<code>private</code>	●	●	●	●	●	●
<code>reduction</code>	●	●	●	●	●	
<code>firstprivate</code>	●	●	●	●	●	●
<code>lastprivate</code>		●	●	●	●	
<code>copyin</code>	●		●		●	
<code>copyprivate</code>						●
<code>schedule</code>		●	●			
<code>ordered</code>		●	●			
<code>collapse</code>		●	●			
<code>nowait</code>		●		●		●

内容提纲

- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

向量化 (vectorization)

- SIMD = Single Instruction Multiple Data
- 大多数处理器均提供具有 SIMD 向量化功能的硬件指令；
- 这些 SIMD 指令一般作用于向量化寄存器中；
- 通过 SIMD 向量化，可以加速计算，例如：



- 不同处理器支持的 SIMD 向量化宽度依赖于硬件本身：

Vector lengths on Intel architectures

→ 128 bit: SSE = Streaming SIMD Extensions



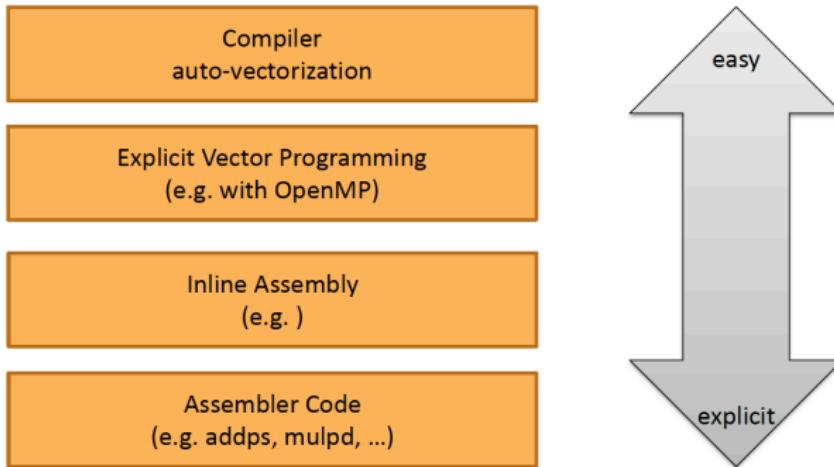
→ 256 bit: AVX = Advanced Vector Extensions



→ 512 bit: AVX-512



- 实现 SIMD 向量化有几种不同的手段：



OpenFOAM 的 SIMD 构造

- OpenFOAM 提供 SIMD 构造对循环进行向量化计算：

```
#pragma omp simd [clause1 | clause2 | ...]  
for_loops
```

- 支持的从句：

```
private (list)  
lastprivate (list)  
reduction (op:list)  
collapse (n)  
linear (list[:step])  
aligned (list[:step])  
safelen (length)
```

- collapse 从句：先对多重循环进行合并，然后进行向量化

```
collapse (n)
```

- linear 从句：列出与迭代变量有线性关系的变量

```
linear (list[:step])
```

- aligned 从句：列出内存地址对齐的数组或指针

```
aligned (list[:step])
```

- safelen 从句：给出没有循环间数据依赖的最大步长

```
safelen (length)
```

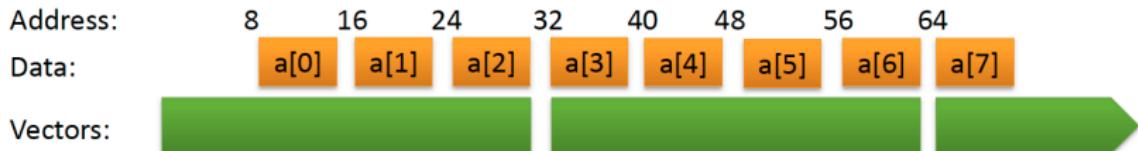
数据对齐

- SIMD 向量化的效果与数据是否对齐 (aligned) 有很大关系，例如：

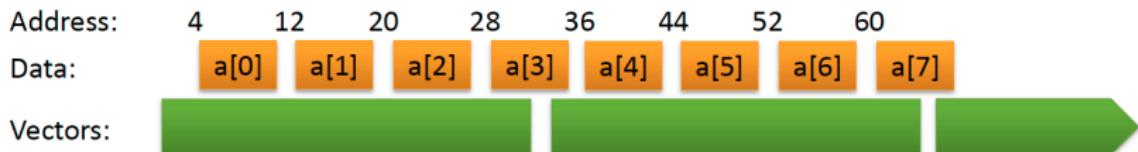
Good alignment



Bad alignment

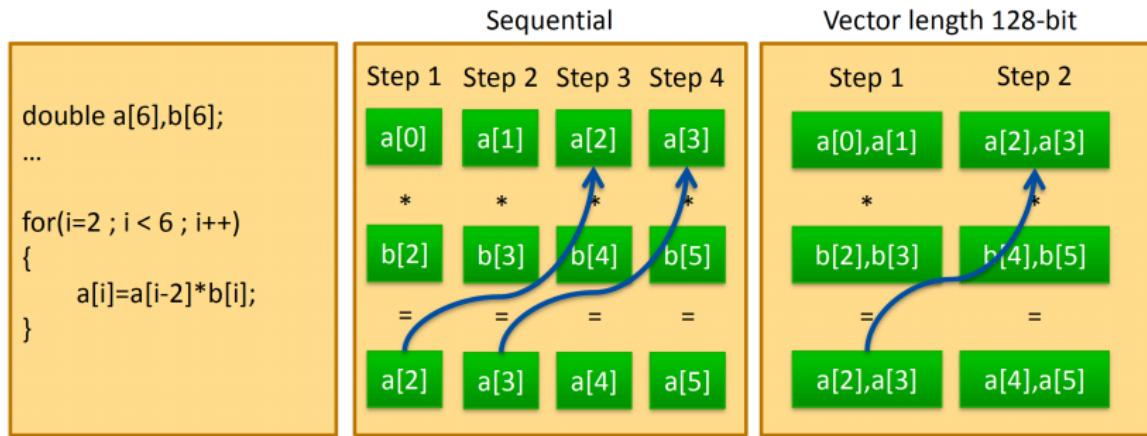


Very bad alignment



向量化的 safelen

- 进行向量化时需要注意不要破坏循环携带的数据依赖；
- safelen 从句用于给出没有循环间数据依赖的最大步长；
- 与 for 构造不同，simd 构造向量化的循环仍然按照顺序依次执行；
- 下例中，safelen=1：



for simd 构造

- simd 构造可与 for 构造合并:

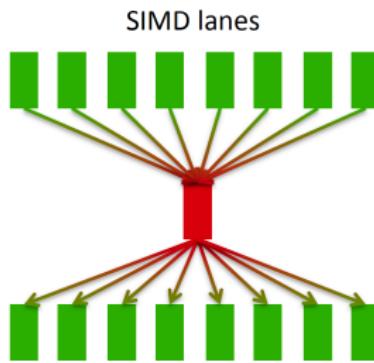
```
#pragma omp for simd [clause1 | clause2 | ...]  
for_loops
```

- 此时，循环任务将先按照线程进行分配，每个线程分配的任务将进一步按照 SIMD 向量化进行二次分配；
- 对于双方共有的从句，将同时起作用；
- 对于各自专有的从句，将各自起作用.

向量化与函数调用

- SIMD 向量化的循环中，如果有外部函数调用，有可能带来严重的性能瓶颈，因为此时函数的执行是完全串行的，例如：

```
for(i=0 ; i < N ; i++)  
{  
    a[i]=b[i]+c[i];  
  
    d[i]=sin(a[i]);  
  
    e[i]=5*d[i];  
}
```



Solutions:

- avoid or inline functions
- create functions which work on vectors instead of scalars

- 上述问题的解决手段包括：使用内联函数或者创建向量化的函数.

declare simd 构造

- declare simd 构造：用于提示编译器根据需要生成一个至多个具有 SIMD 向量化功能的函数

```
#pragma omp declare simd [clause1 | clause2 | ...]  
function_definition/declaration
```

- 支持的从句：

```
linear (list[:step])  
aligned (list[:step])  
uniform (list)  
simdlen (length)  
inbranch | notinbranch
```

- `linear` 从句：列出与迭代变量有线性关系的变量

```
linear (list[:step])
```

- `aligned` 从句：列出内存地址对齐的变量

```
aligned (list[:step])
```

- `uniform` 从句：列出不变量

```
uniform (list)
```

- `simdlen` 从句：给出需要同时向量化计算的变量个数

```
simdlen (length)
```

- `inbranch/notinbranch` 从句：声明在/不在分支判断中被调用

```
inbranch | notinbranch
```

编译器的自动向量化

- 事实上，不少编译器都提供了较为不错的自动向量化功能；
- 下表总结了几种主流编译器中开启和关闭自动向量化的选项：

Compiler	Compilers Options	Disabling Vectorizer
Intel C/C++ 17.0	-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed	-no-vec
GCC C/C++ 6.3.0	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed	-fno-tree-vectorize
LLVM/Clang 3.9.1	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize	-fno-vectorize
PGI C/C++ 16.10	-O3 -Mvect -Minfo=loop,vect -Mneginfo=loop,vect	-Mnovect

示例程序：计算 π

omp_cpi2.c

```
1 ...
2 #pragma omp declare simd
3 double f(double x) {
4     return (16.0*(x-1.0)/(x*x*x*x-2.0*x*x*x+4.0*x-4.0));
5 }
6 int main(int argc, char *argv[]){
7     ...
8 #pragma omp parallel for simd private(x) linear(i) reduction(+: pi)
9     for (i = 0; i < n; i++) {
10         x = h * ((double)i + 0.5);
11         pi += h * f(x);
12     }
13     ...
14 }
```

- 注：这里选择了一个计算量更大的积分公式用来检验 SIMD 的效果
- 编译方式

```
$ module load gcc # load gcc 9.2.0
$ gcc -o cpi2 omp_cpi2.c -O3 -Wall -fopenmp \
-fopt-info-vec
(gcc 显示成功进行向量化的信息)
$ gcc -o cpi2nosimd omp_cpi2nosimd.c -O3 -Wall \
-fopenmp -fopt-info-vec -fno-tree-vectorize
(gcc 没有显示成功进行向量化的信息)
```

- 测试结果

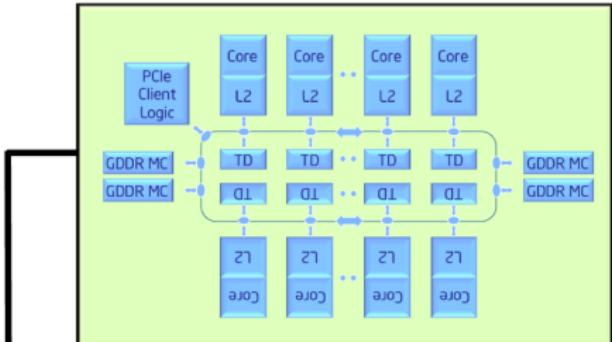
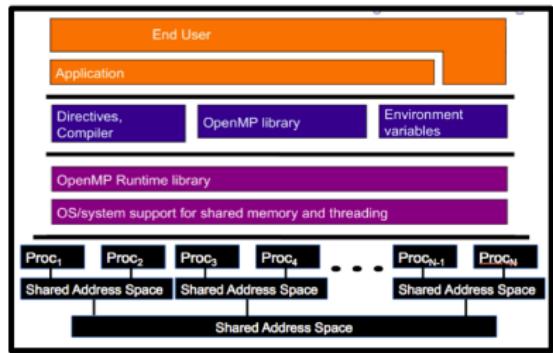
线程数	1	2	4	8
无向量化	3.92s	2.07s	1.17s	0.65s
向量化	1.98s	1.08s	0.63s	0.33s

内容提纲

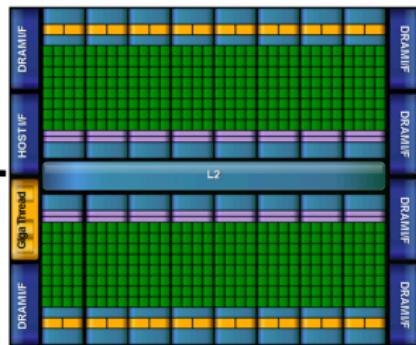
- ① 入门知识
- ② 从 Hello World 谈起
- ③ 并行区制导语句
- ④ 循环工作共享构造
- ⑤ 其他工作共享构造
- ⑥ 数据依赖
- ⑦ 同步构造
- ⑧ 线程控制
- ⑨ 任务构造
- ⑩ 持久变量
- ⑪ 向量化
- ⑫ 补遗与新特性

OpenMP 对加速卡 (accelerator) 的支持

Supported (since OpenMP 4.0)
with target, teams, distribute,
and other constructs

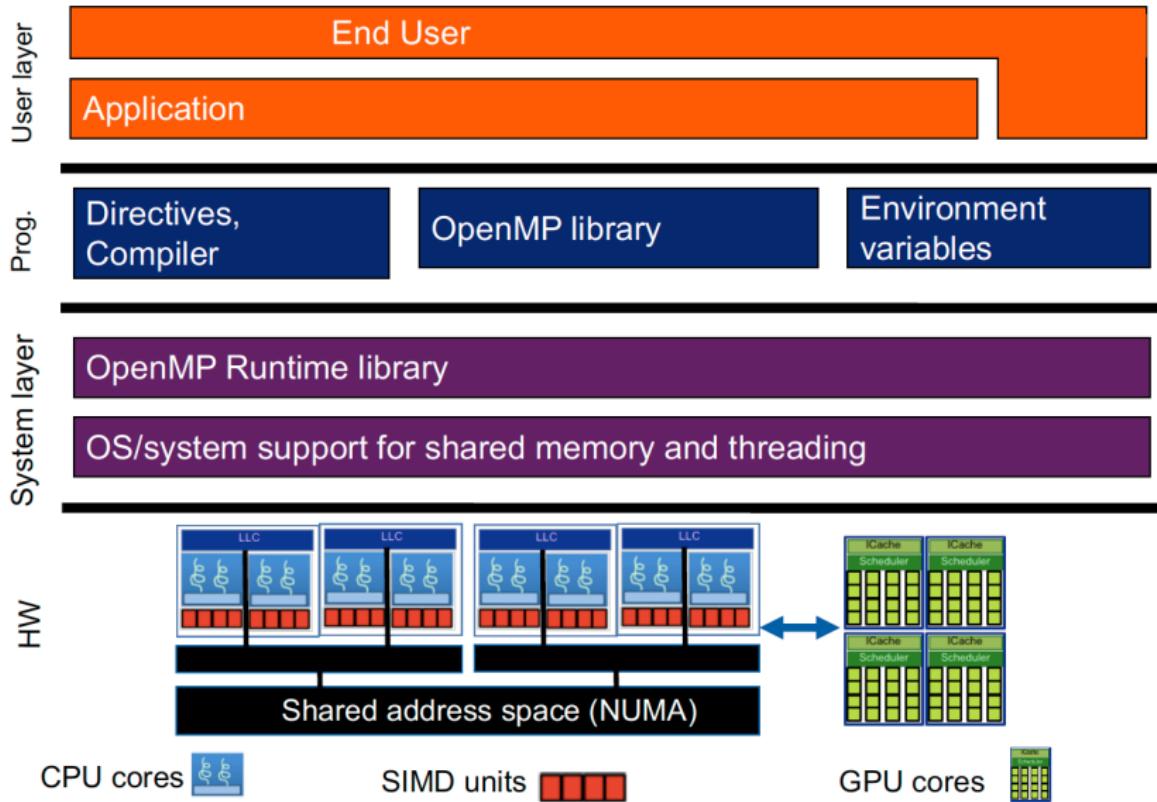


Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

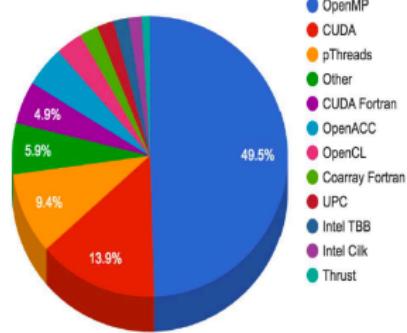
更新后的 OpenMP 基础解决方案



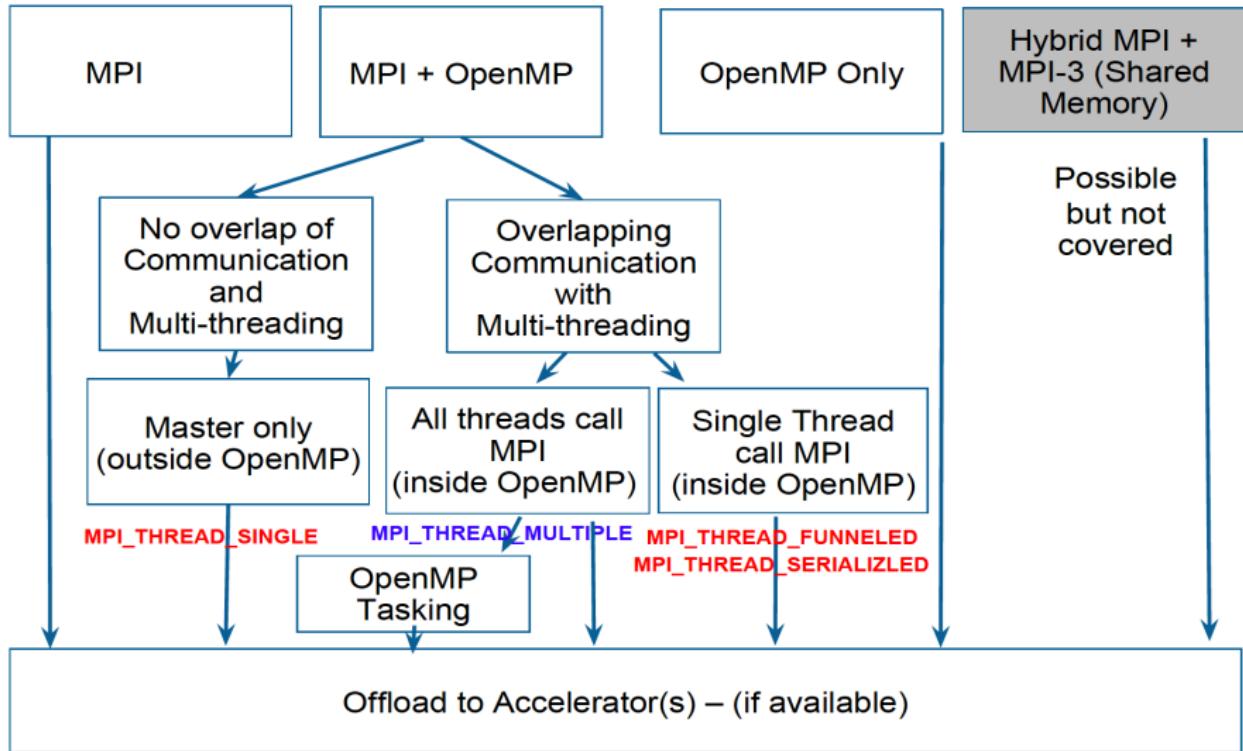
各种并行编程模型汇总

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for shared memory or intra-node, and now supports accelerators as well (intra means within)
- Hybrid Programming (MPI+X) is when we use a solution with different programming models for inter vs. intra-node parallelism
- Several solutions including
 - Pure MPI
 - MPI + Shared Memory (OpenMP)
 - MPI + Accelerator programming
 - OpenMP 4.5 shared memory + offload, OpenACC, CUDA, etc
 - MPI message passing + MPI shared memory
 - PGAS: UPC/UPC++, Fortran 2008 coarrays, GA, OpenSHMEM, etc
 - Runtime tasks (Legion, HPX, HiHat (draft), etc)
 - Other hybrid based on Kokkos, Raja, SYCL, C++17 (C++20 draft)

NERSC data from 2015:
When asked: If you use MPI + X,
what is X ?



MPI+OpenMP 混合编程



其他未讨论的话题

- 假共享 (false sharing);
- 线程锁;
- 自定义规约;
- 高级任务并行 (OpenMP4.0);
- ...

OpenMP 5.0 发布

2018 年 11 月 8 日，在美国 Dallas 举办的 SC18 大会上，OpenMP 5.0 正式发布，在功能方面的主要增强/增加包括：全面支持众核加速设备（Intel Xeon Phi, GPU 等）、增强的调试和性能分析接口、支持最新版 C/C++/Fortran、增强的可移植性等。



Press Release

[More press releases»](#)

Nov 8, 2018 17:00 UTC

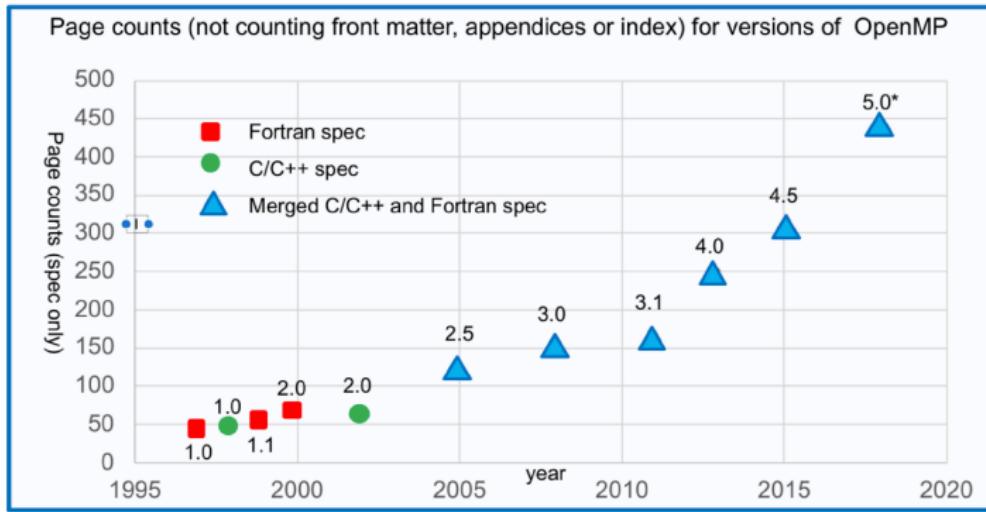
OpenMP® 5.0 is a Major Leap Forward

DALLAS--(Business Wire)--#SC18

The OpenMP Architecture Review Board (ARB) is pleased to announce Version 5.0 of the OpenMP API Specification, a major upgrade of the OpenMP language. The OpenMP

OpenMP 发展历史一览

- 从 1997 年诞生开始，OpenMP 经历了从仅适用于科学计算的简单接口到更为通用的、复杂的 API 的发展历程。



* Does not include the tools interface added with OpenMP 5.0 which pushes the page count to 618

- 为了降低 OpenMP 的学习复杂性，可先学习主要的 OpenMP 常用核心 (Common Cores)，在此基础上学习向量化等高级技能。

OpenMP Common Core Reference Guide

OpenMP API Common Core

Page 1



Common Core

OpenMP API Reference Guide: Common Core

OpenMP® is an API for writing parallel applications in C/C++ and Fortran. OpenMP is suitable for programming multicore chips, NUMA systems, and devices attached to a CPU (such as a GPU).

The OpenMP Common Core consists of the 19 most commonly used items from the full OpenMP API. It is often best to master the Common Core and then proceed to the full OpenMP API as needed.

C/C++ content

Fortran content

[n.n.n] 5.0 spec. [n.n.n] 4.5 spec.

Directives and Constructs

parallel construct

parallel [2.6] [2.5]

Forms a team of threads and starts parallel execution.

	#pragma omp parallel [clause[,]clause] ...) structured-block
	!\$omp parallel [clause[,]clause] ...) structured-block !\$omp end parallel

clause: reduction(op : list), private(list), firstprivate(list), shared(list)

Worksharing constructs

single [2.8.2] [2.7.3]

Specifies that the associated structured block is executed by only one of the threads in the team. Has implied barrier unless turned off with nowait.

	#pragma omp single [clause[,]clause] ...) structured-block
	!\$omp single [clause[,]clause] ...) structured-block

Tasking constructs

task [2.10.1] [2.9.1]

Defines an explicit task.

	#pragma omp task [clause[,]clause] ...) structured-block
	!\$omp task [clause[,]clause] ...) structured-block !\$omp end task

clause: private(list), firstprivate(list), shared(list)

Synchronization constructs

critical [2.17.1] [2.13.2]

Restricts execution of the associated structured block to a single thread at a time.

	#pragma omp critical structured-block
	!\$omp critical structured-block

barrier

[2.17.2] [2.13.3]

Specifies an explicit barrier at the point at which the construct appears.

	#pragma omp barrier
	!\$omp barrier

taskwait

[2.17.5] [2.13.4]

Specifies a wait on the completion of child tasks of the current task.

	#pragma omp taskwait
	!\$omp taskwait

Memory consistency

Rules that define which values are allowed to be observed when a variable shared between one or more threads is read. A thread uses a flush to make its variables consistent with memory. A flush is implied at the following boundaries:

OpenMP 官方网址

- 为了更好的了解 OpenMP 的技术动态和进展，可关注官网：
<https://www.openmp.org>

