并行与分布式计算基础：第十讲

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)					

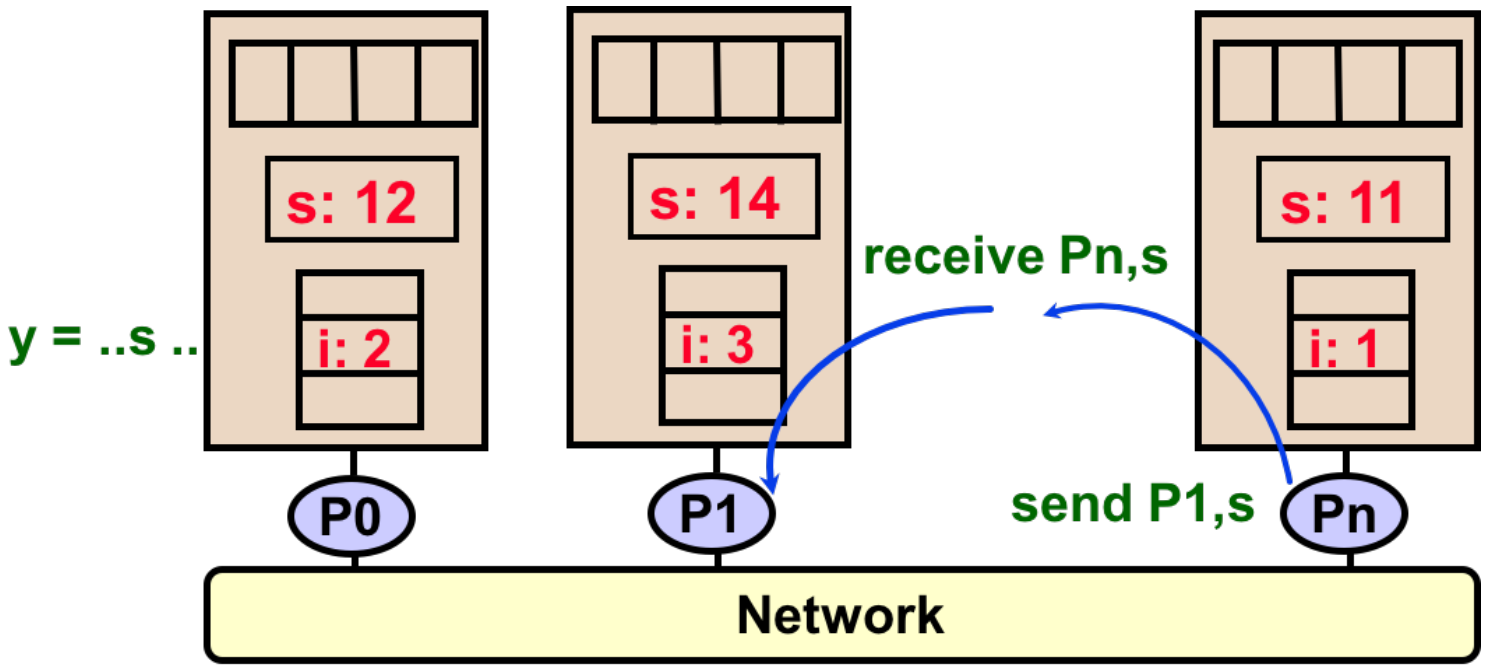
内容提纲

1. OpenMP 编程-1：入门篇
   - 共享内存编程模型
   - 初识 OpenMP
   - 一个例子
   - 并行区制导语句
   - 又一个例子

共享内存编程模型

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?

CPU-MIC异构计算

2个12核CPU
+
3块57核MIC

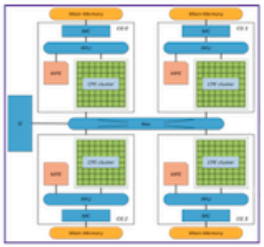
16000x

天河2（312万核）

2013.6~2015.11
世界第一

片上异构计算

4个主核
+
256个从核



40000x

神威太湖之光（1064万核）

2016.6~2017.11
世界第一

CPU-GPU异构计算

2个22核CPU
+
6块80核GPU

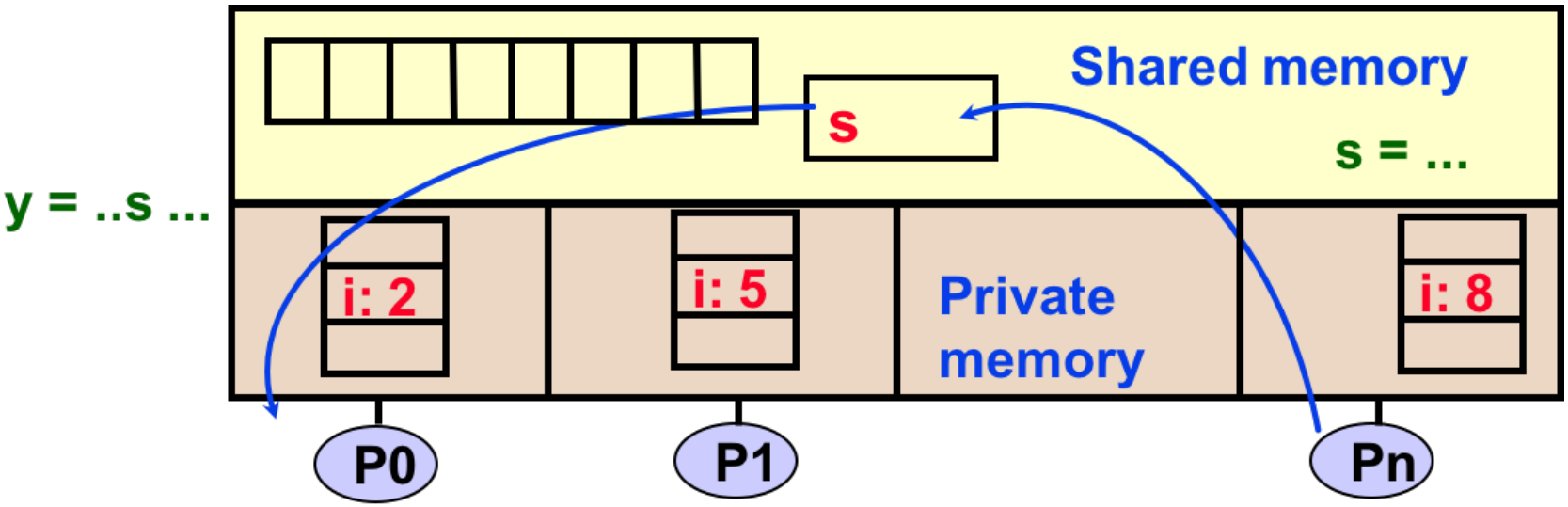
4608x

顶点（241万核）

2018.6~今
世界第一

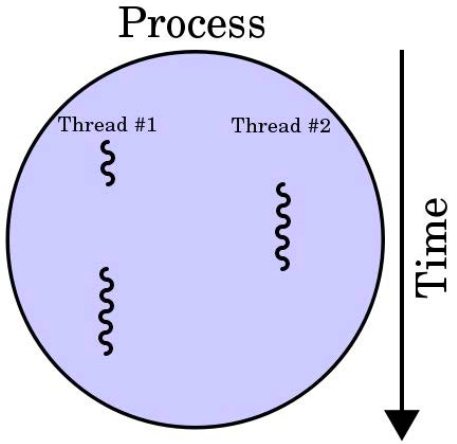
共享内存并行编程

- 程序由一系列线程 (thread) 控制；
- 每个线程有自己私有的变量；
- 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。

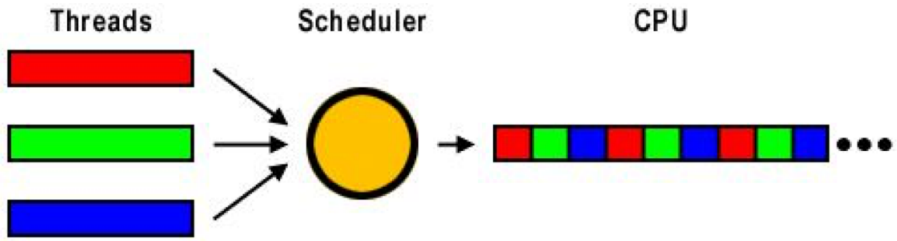


线程与进程的关系

- 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)。



初识 OpenMP

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)。

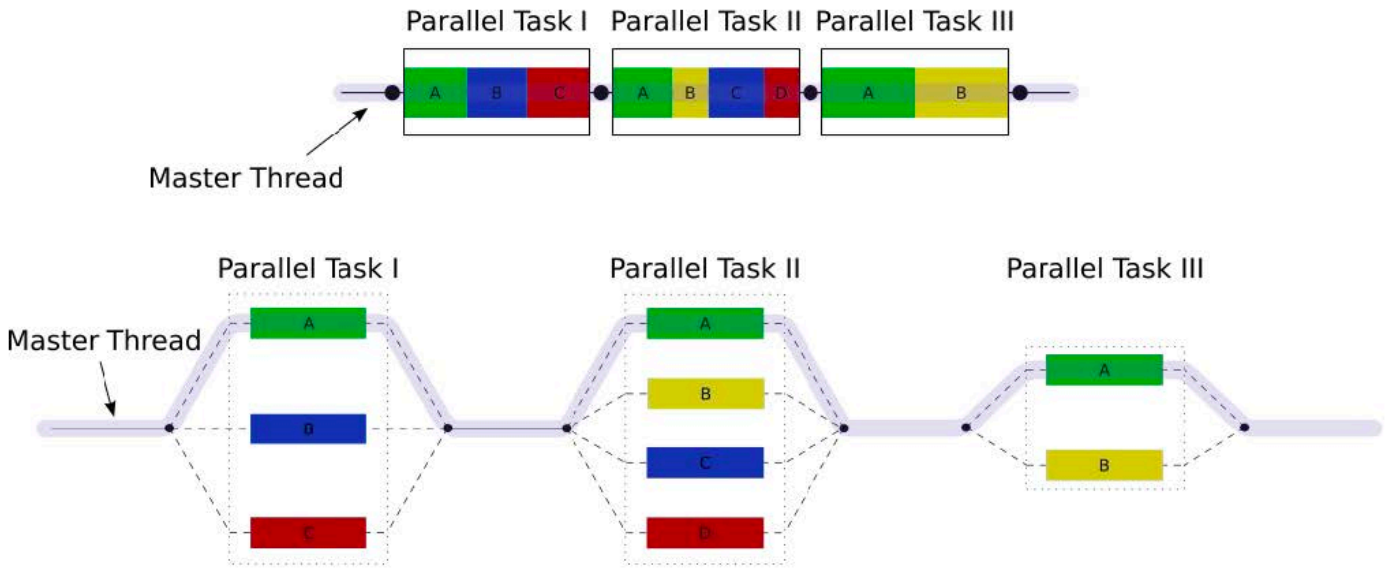
OpenMP 的发展历史

- 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

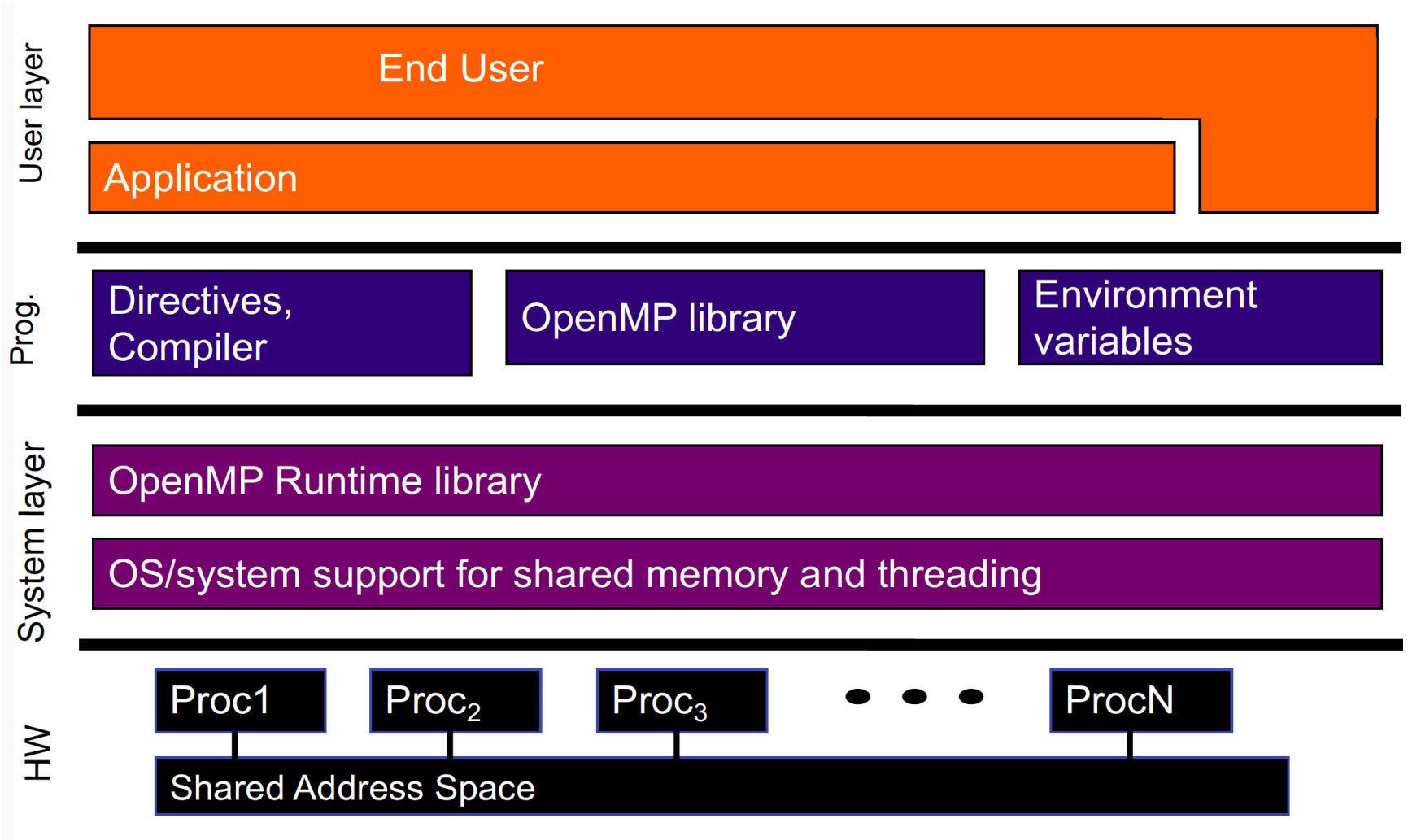
- OpenMP 主要采用 Fork-Join 模式进行并行执行；
- 程序开始时只有一个线程：主线程 (master thread)，编号为 0；
- 主线程仅当进入并行区 (parallel region) 才并行执行：
  - Fork：主线程创建一组并行线程，编号 0 ～ $n-1$；
  - 执行：并行线程在并行区中并行执行；
  - Join：在并行区结尾并行线程进行同步和结束，只保留主线程；
- 并行区的个数，以及每个并行区中的线程数，都可以任意设置。



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

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	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

主要用途：

- Setting the number of threads；
- Specifying how loop interations 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	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

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 19 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
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting 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(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers. The flush concept (but not the concept)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

一个例子

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,
       nthreads);
13   }
14   t1 = omp_get_wtime();
15   printf("Time elapsed is %f.\nThat's all, folks!\n", t1-t0);
16   return 0;
17 }
```

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

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)

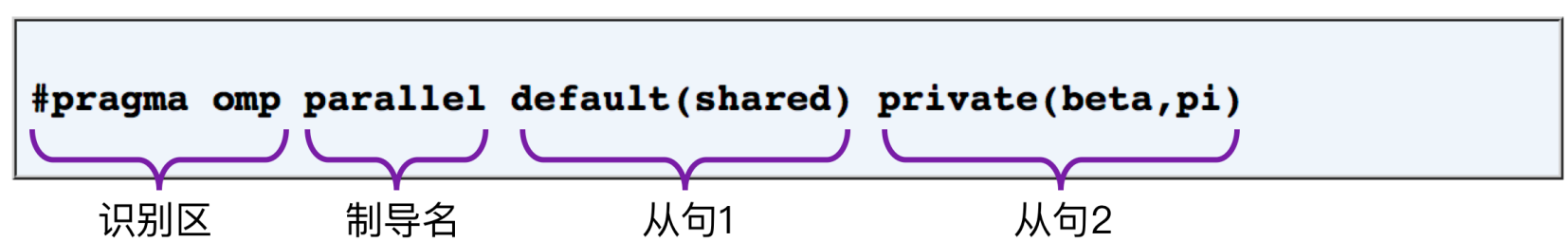
- OpenMP 提供了内置宏 _OPENMP，帮助判断 OpenMP 是否存在；
- 借助条件编译，我们可以在不支持 OpenMP 的环境下也能编译并正常运行程序，例如：

```
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   ...
```

并行区制导语句

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。



  - 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 (1)

```
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。

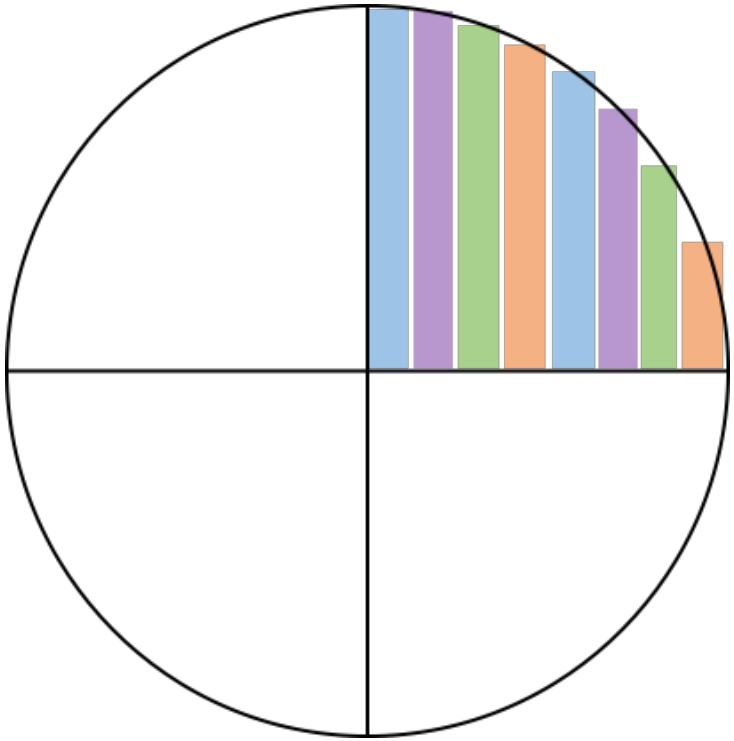
数据域从句 (2)

- 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。

又一个例子

示例程序：计算 π (1)

- 计算依据：单位圆的面积。



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

where

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

- 并行策略：round-robin。

示例程序：计算 π (2)

omp_cpi.c

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  #define PI25DT 3.14159265358979323846264 3
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 parallel default(shared) \
16   private(tid, i, x) reduction(+:pi)
17   {
18     nthreads = omp_get_num_threads();
```

示例程序：计算 π (3)

```
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  }
```

示例程序：计算 π (4)

- 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。