



Note

介绍

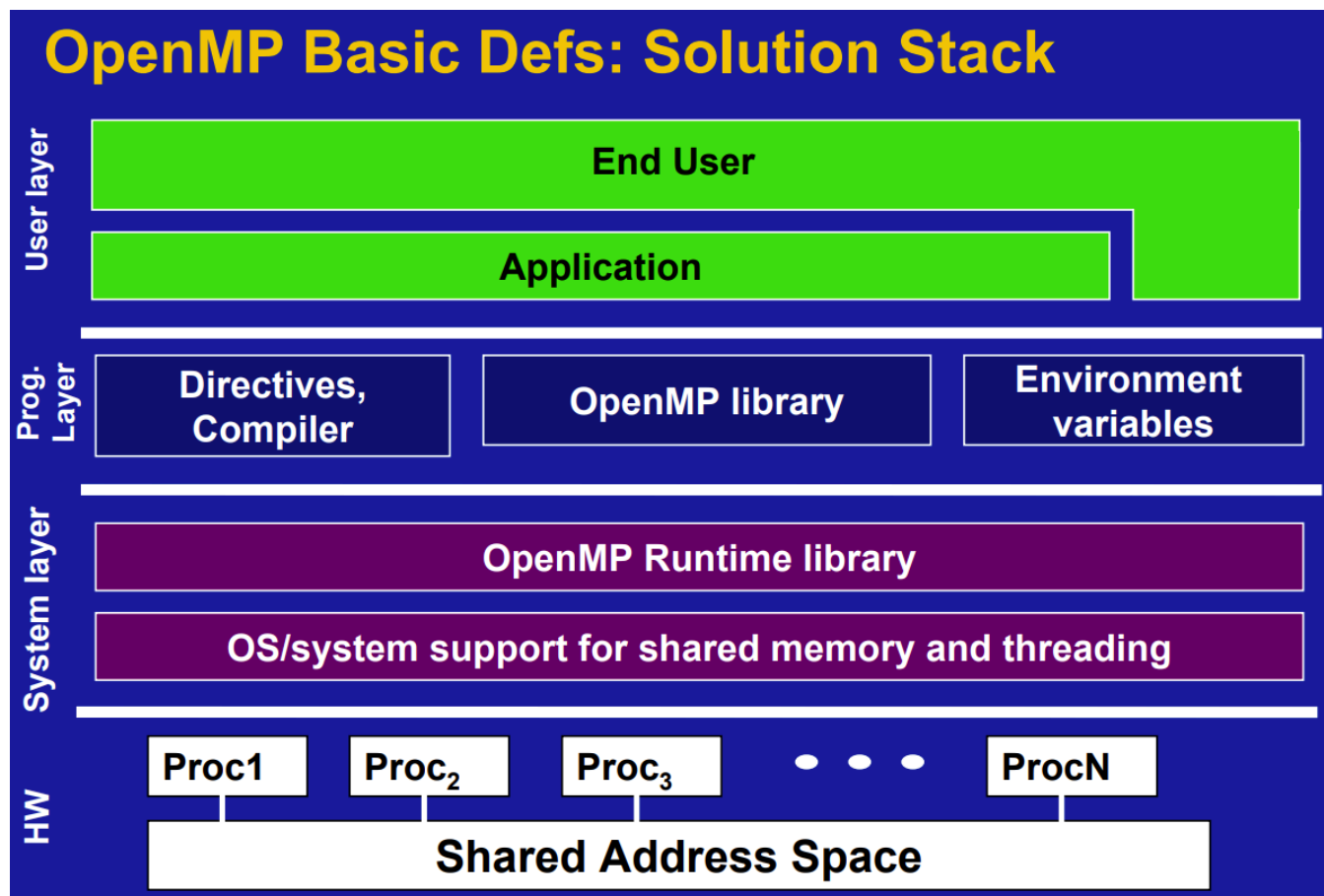
OpenMP是什么

- Open Multi-Processing
- OpenMP是一个应用程序接口（API）
 - 你可以简单理解为它是一个库
- 支持C, C++和Fortran
- 支持多种指令集和操作系统
- 由非营利性组织管理，由多家软硬件厂家参与，包括Arm, AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia等

可以在[官网页面](#)查询到openmp的历史版本和发布日期

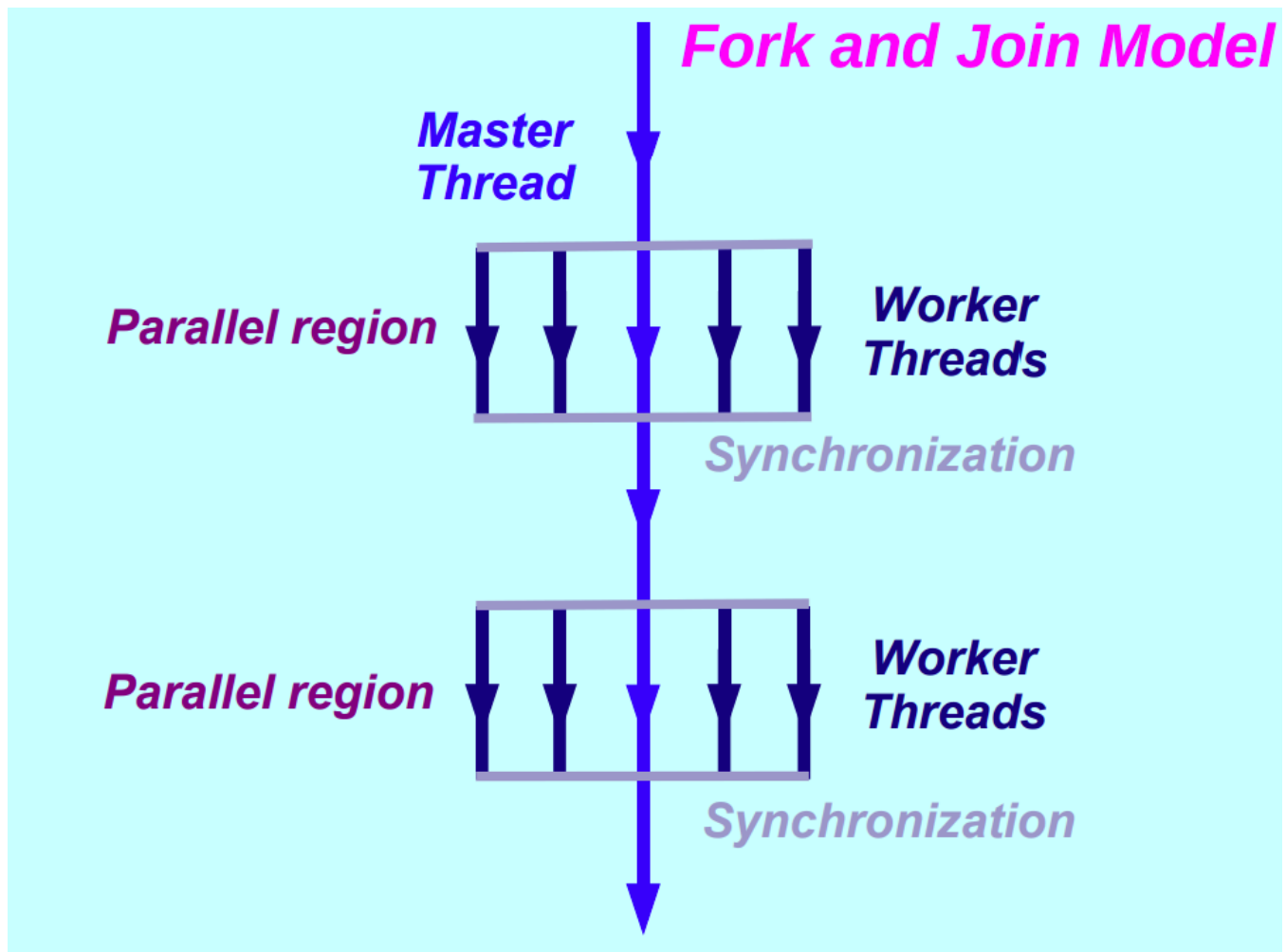
日期	版本
October 1997	Fortran 1.0
October 1998	C/C++ 1.0
March 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
July 2011	OpenMP 3.1
July 2013	OpenMP 4.0
November 2015	OpenMP 4.5
November 2018	OpenMP 5.0
November 2020	OpenMP 5.1
November 2021	OpenMP 5.2

技术框架



包含OpenMP library和OpenMP Runtime library

执行模型：Fork-Join Model



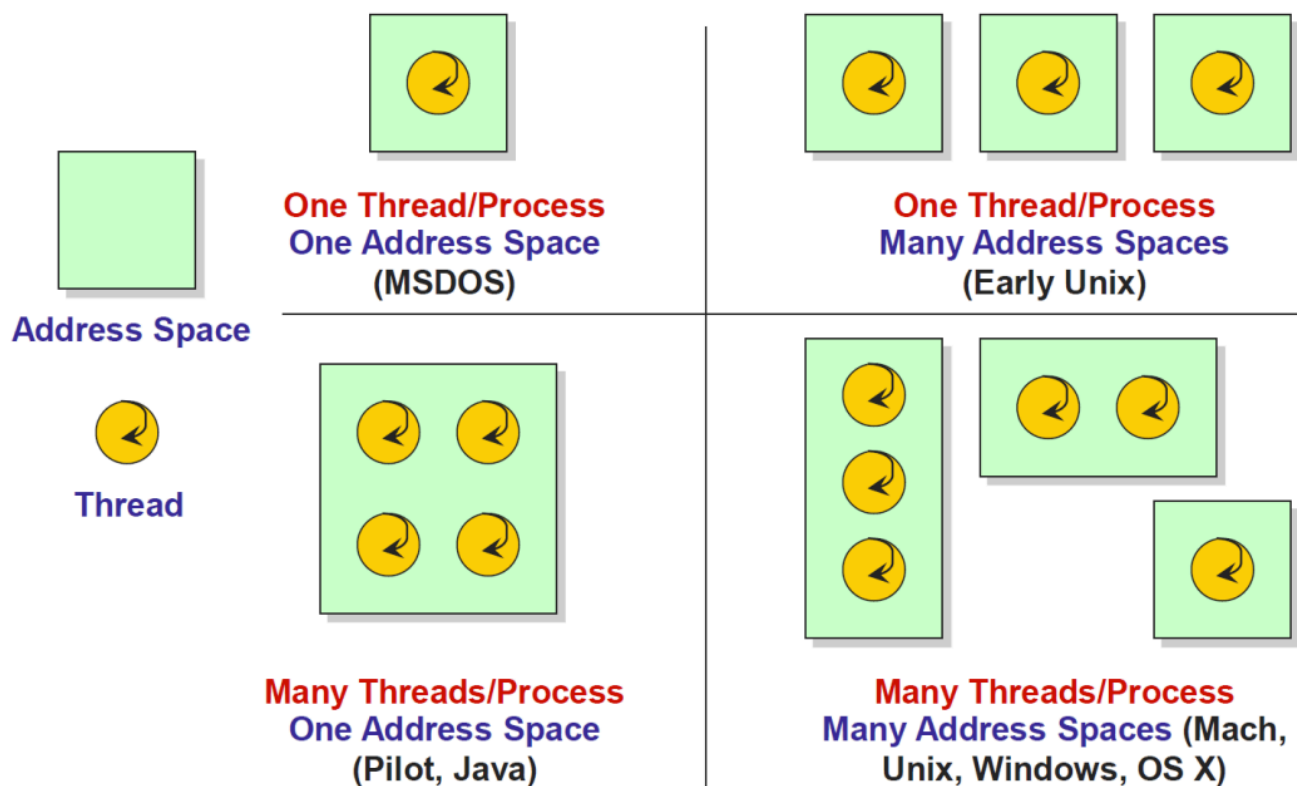
- 单线程（initial thread）开始执行
- 进入并行区（parallel region）开始并行执行
- 在并行区结尾进行同步和结束线程，继续单线程执行程序

基础知识

进程和线程

- 进程
 - 每个进程有自己独立的地址空间
 - CPU在进程之间切换需要进行上下文切换
- 线程
 - 一个进程下的线程共享地址空间

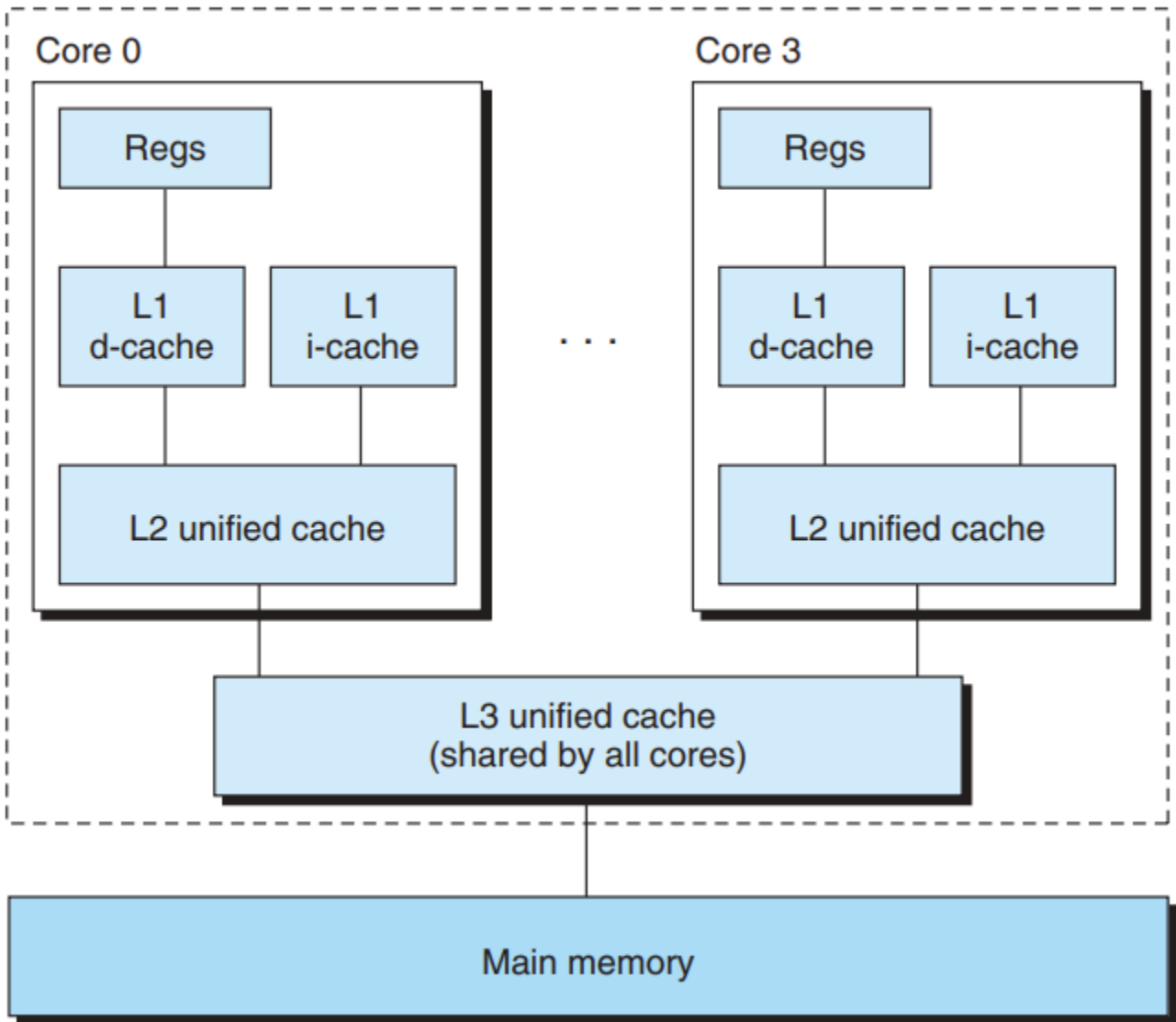
- CPU在线程之间切换开销较小



你可以简单理解，操作系统会自行将线程调度到CPU核心运行。所以当线程数目超过核心数，会出现多个线程抢占一个CPU核心，导致性能下降

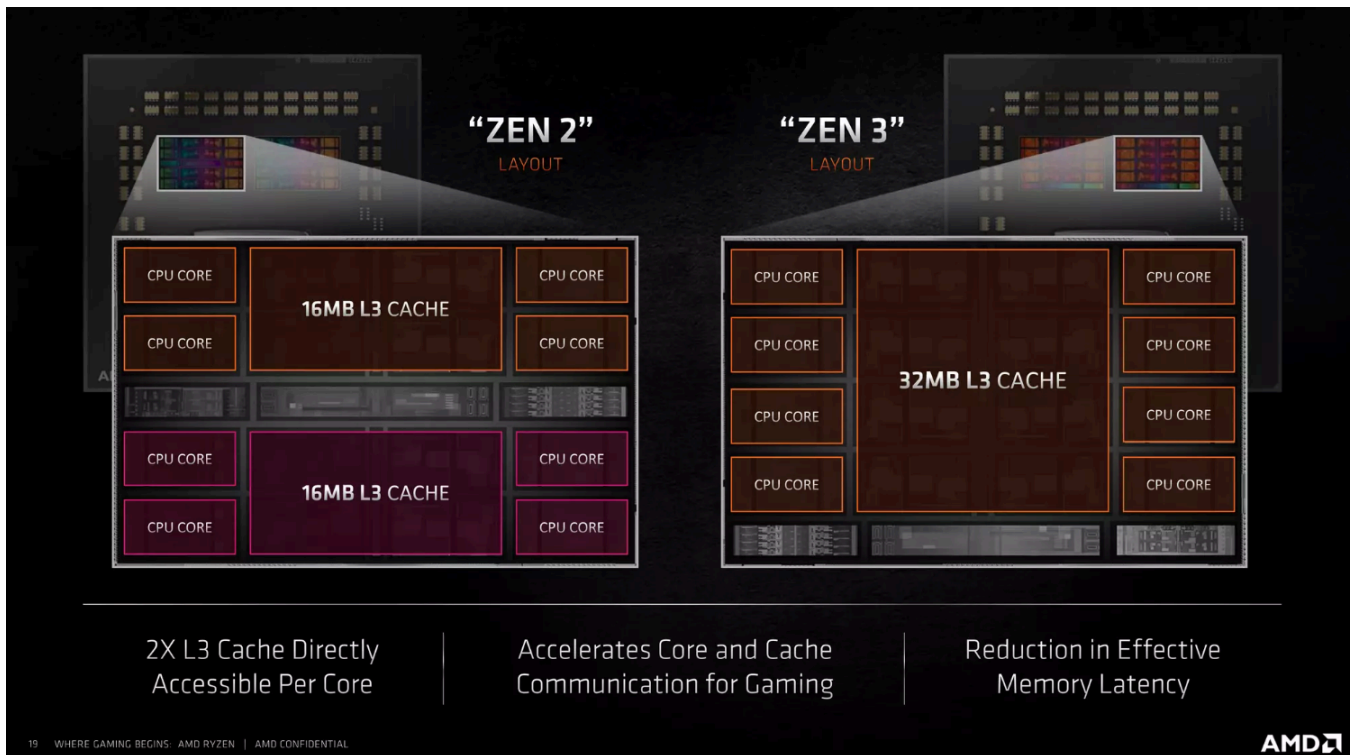
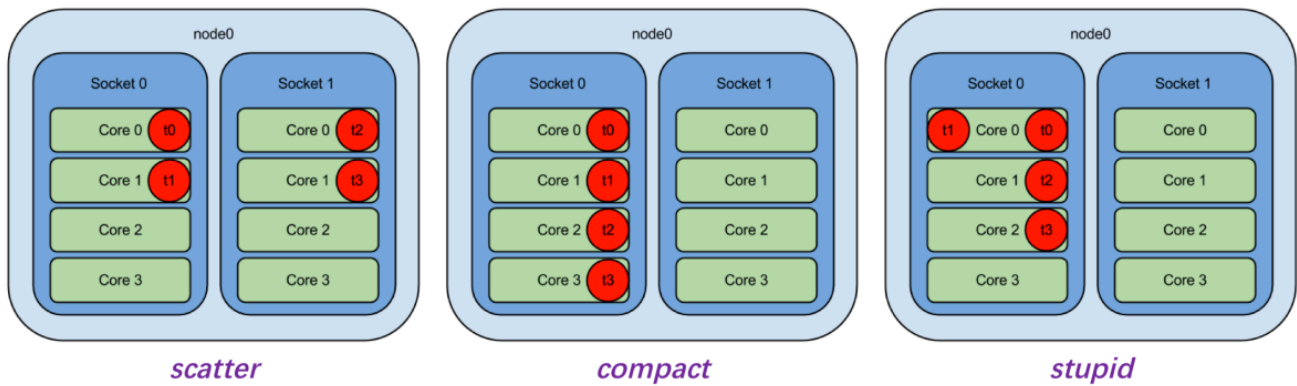
硬件内存模型

Processor package



- CPU核心在主存之上有L1, L2, L3多级缓存
- L1, L2缓存是核心私有的
- 硬件和操作系统保证不同核心的缓存一致性 (coherence)
- 被称为cache coherence non-uniform access architecture(ccNUMA)
- 缓存一致性会带来False Sharing的问题 (之后会讲到)

实际的内存模型更加复杂



openmp支持控制线程的绑定

- 环境变量OMP_PROC_BIND/从句 `proc_bind(master|close|spread)`：控制线程绑定与否，以及线程对于绑定单元（称为 place）分布
- 环境变量OMP_PLACES：控制每个place的对应，常用threads/cores/sockets

参考[文档](#)以及相关官方手册

OpenMP编程

安装

含在Ubuntu提供的build-essential包中

如何查看openmp版本？

```
echo |cpp -fopenmp -dM |grep -i open  
# #define _OPENMP 201511
```

编译使用

可以直接在编译语句添加 `-fopenmp`，如：

```
g++ -O2 -std=c++14 -fopenmp hello.cpp -o hello
```

如果使用cmake构建项目：

```
find_package(OpenMP)  
add_compile_options(-Wunknown-pragmas)  
  
add_executable(hello src/hello.cpp)  
target_link_libraries(hello OpenMP::OpenMP_CXX)
```

加入 `-Wunknown-pragmas` 会在编译时报告没有处理的 `#pragma` 语句

Hello

见 `src/hello.cpp`

```
#pragma omp parallel num_threads(8)  
{  
    int tid = omp_get_thread_num();  
    int num_threads = omp_get_num_threads();  
    printf("Hello from thread %d of %d\n", tid, num_threads);  
}
```

执行结果

```
Hello from thread 0 of 8
Hello from thread 4 of 8
Hello from thread 5 of 8
Hello from thread 3 of 8
Hello from thread 6 of 8
Hello from thread 1 of 8
Hello from thread 2 of 8
Hello from thread 7 of 8
```

- 同一类openmp制导语句称为一种构造 (construct)
- 形式为 `#pragma omp <directive name> <clause>`
- 使用 `{ }` 标记作用的代码块

设置运行线程数（优先级由低到高）：

- 什么也不做，系统选择运行线程数
- 设置环境变量 `export OMP_NUM_THREADS=4`
- 代码中使用库函数 `void omp_set_num_threads(int)`
- 通过制导语句从句 `num_threads(integer-expression)`
- `if` 从句判断串行还是并行执行

一些常用库函数：

```
// 设置并行区运行的线程数
void omp_set_num_threads(int)
// 获得并行区运行的线程数
int omp_get_num_threads(void)
// 获得线程编号
int omp_get_thread_num(void)
// 获得openmp wall clock时间（单位秒）
double omp_get_wtime(void)
// 获得omp_get_wtime时间精度
double omp_get_wtick(void)
```

parallel构造

支持的从句

- `if(scalar_expression)`：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

- `num_threads(integer_expression)` : 指定并行区的线程数
- `default(shared|none)` : 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(list)` : 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
 - 请保证共享访问不会冲突
 - 不特别指定并行区变量默认为**shared**
- `private(list)` : 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
 - 变量需要**重新初始化**
- `firstprivate(list)`
 - 同 `private`
 - 对变量根据主线程中的数据进行初始化

数据从句见 `src/data_clause.cpp`

```

#pragma omp parallel num_threads(4)
{
    int tid = omp_get_thread_num();
    for (int i = 0; i < 4; i++) {
        cnt += 1;
    }
    results[tid] = cnt;
}

#pragma omp parallel num_threads(4) private(cnt)
{
    int tid = omp_get_thread_num();
    for (int i = 0; i < 4; i++) {
        cnt += 1;
    }
    results[tid] = cnt;
}

#pragma omp parallel num_threads(4) firstprivate(cnt)
{
    int tid = omp_get_thread_num();
    for (int i = 0; i < 4; i++) {
        cnt += 1;
    }
    results[tid] = cnt;
}

```

执行结果

```

no clause: 5 9 17 13
private(not init): 4 -187939698 -187939698 -187939698
firstprivate: 5 5 5 5

```

for构造

在并行区内对for循环进行线程划分，且for循环满足格式要求：

- init-expr: 需要是 `var=lb` 形式，类型也有限制
- test-expr: 限制为 `var relational-opb` 或者 `b relational-op var`
- incr-expr: 仅限加减法

详细参考[OpenMP API 5.0 Specification](#), p53

for构造见 `src/hello_for.cpp`

```
#pragma omp parallel num_threads(8)
{
    int tid = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    #pragma omp for ordered
    for (int i = 0; i < num_threads; i++) {
        // do something
        // #pragma omp ordered
        // #pragma omp critical
        std::cout << "Hello from thread " << tid << std::endl;
    }
}
```

执行结果

```

#no synchronization
Hello from thread 0Hello from thread
Hello from thread 4
Hello from thread Hello from thread Hello from thread 7
Hello from thread 1
2
Hello from thread 5
6
3

# ordered
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 4
Hello from thread 5
Hello from thread 6
Hello from thread 7

# critical
Hello from thread 5
Hello from thread 4
Hello from thread 1
Hello from thread 7
Hello from thread 6
Hello from thread 3
Hello from thread 2
Hello from thread 0

```

常常将parallel和for合并为parallel for指导语句

parallel for支持的从句：

	parallel	for	parallel for
if	√		√
num_threads	√		√
default	√		√
copyin	√		√
private	√	√	√

	parallel	for	parallel for
firstprivate	√	√	√
shared	√	√	√
reduction	√	√	√
lastprivate		√	√
schedule		√	√
ordered		√	√
collapse		√	√
nowait		√	

- `lastprivate(list)`
 - 同 `private`
 - 将执行最后一个循环的线程的私有数据取出赋值给主线程的变量
- `ordered`
 - 声明有潜在的顺序执行部分
 - 使用 `#pragma omp ordered` 标记顺序执行代码（搭配使用）
 - `ordered`区内的语句任意时刻仅由最多一个线程执行
- `collapse(n)`：应用于n重循环
 - 合并循环
 - 注意循环之间是否有数据依赖
- `nowait` 取消代码块结束时的栅栏同步（barrier）
- `schedule(type [,chunk])`：控制调度方式
 - `static`：chunk大小固定（默认n/p）
 - `dynamic`：动态调度，chunk大小固定（默认为1）
 - `guided`：chunk大小动态缩减
 - `runtime`：由系统环境变量OMP_SCHEDULE决定

`schedule` 见 `src/inner_product.cpp`

Reduction

见 `src/vector_norm.cpp`

```
for (int i = 0; i < N; i++) {
    ans_seq += b[i] * b[i];
}
ans_seq = sqrt(ans_seq);

#pragma omp parallel for reduction(+ : ans_omp)
for (int i = 0; i < N; i++) {
    ans_omp += b[i] * b[i];
}
ans_omp = sqrt(ans_omp);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    #pragma omp atomic
    // #pragma omp critical
    ans_omp_sync += b[i] * b[i];
}
ans_omp_sync = sqrt(ans_omp_sync);
```

执行结果

```
# atomic
size: 33554432
sequence result: 3344.05
omp result: 3344.05
omp sync result: 3344.05
sequence time: 0.0928805
omp time: 0.0180116
omp sync time: 5.17156

# critical
size: 33554432
sequence result: 3344.2
omp result: 3344.2
omp sync result: 3344.2
sequence time: 0.0929021
omp time: 0.0179938
omp sync time: 7.74378
```

Identifier	Initializer	Combiner
+	<code>omp_priv = 0</code>	<code>omp_out += omp_in</code>
*	<code>omp_priv = 1</code>	<code>omp_out *= omp_in</code>
-	<code>omp_priv = 0</code>	<code>omp_out -= omp_in</code>
&	<code>omp_priv = 0</code>	<code>omp_out &= omp_in</code>
	<code>omp_priv = 0</code>	<code>omp_out = omp_in</code>
^	<code>omp_priv = 0</code>	<code>omp_out ^= omp_in</code>
&&	<code>omp_priv = 1</code>	<code>omp_out = omp_in && omp_out</code>
	<code>omp_priv = 0</code>	<code>omp_out = omp_in omp_out</code>

Identifier	Initializer	Combiner
max	<code>omp_priv = Least representable number in the reduction list item type</code>	<code>omp_out = omp_in > omp_out ? omp_in : omp_out</code>
min	<code>omp_priv = Largest representable number in the reduction list item type</code>	<code>omp_out = omp_in < omp_out ? omp_in : omp_out</code>

- fork线程并分配任务
- 每一个线程定义一个私有变量 `omp_priv`
- 各个线程执行计算
- 所有 `omp_priv` 和 `omp_in` 一起顺序进行reduction，写回原变量

False Sharing

见 `src/matrix_vector.cpp`

```

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        x_seq[i] += A[i * N + j] * b[j];
    }
}

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    double tmp = 0;
    for (int j = 0; j < N; j++) {
        tmp += A[i * N + j] * b[j];
    }
    x_omp[i] = tmp;
}

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        x_omp_fs[i] += A[i * N + j] * b[j];
    }
}

```

执行结果

```

size: 16384
sequence time: 1.07896
simple omp time: 0.0938018
false sharing time: 0.110479

size: 16384
sequence time: 1.38333
simple omp time: 0.0958252
false sharing time: 0.115473

size: 16384
sequence time: 1.0359
simple omp time: 0.0973124
false sharing time: 0.129693

```

- 耗时增加24%
- 不同核心对同一cache line的同时读写会造成严重的冲突，导致该级缓存失效。

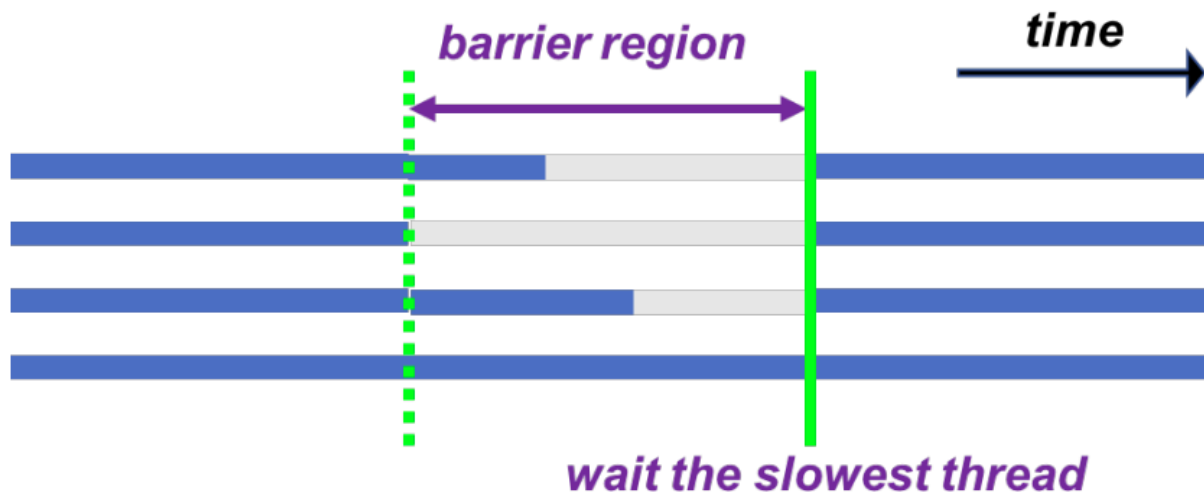
sections构造

- 将并行区内的代码块划分为多个section分配执行
- 可以搭配parallel合成为parallel sections构造
- 每个section由一个线程执行
 - 线程数大于section数目：部分线程空闲
 - 线程数小于section数目：部分线程分配多个section

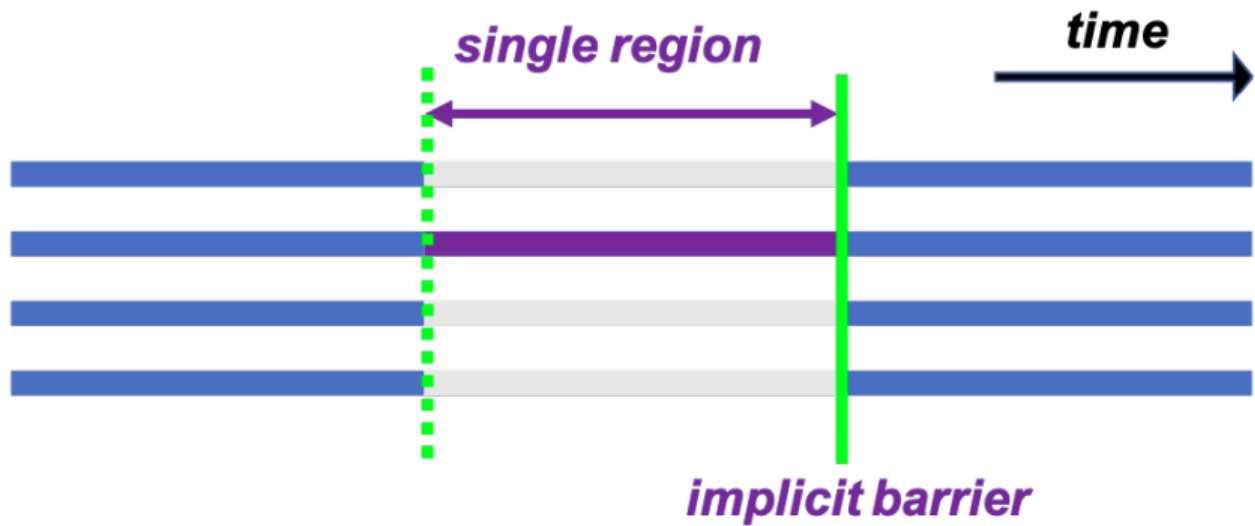
```
#pragma omp sections
{
  #pragma omp section
  code1();
  #pragma omp section
  code2();
}
```

同步构造

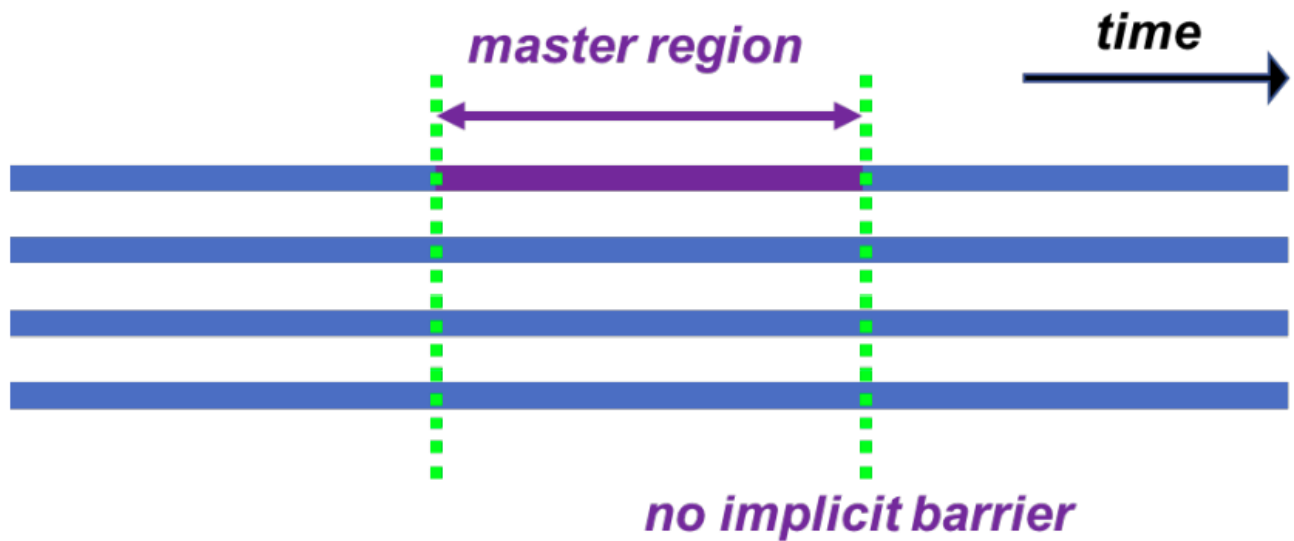
`#pragma omp barrier`：在特定位置进行栅栏同步



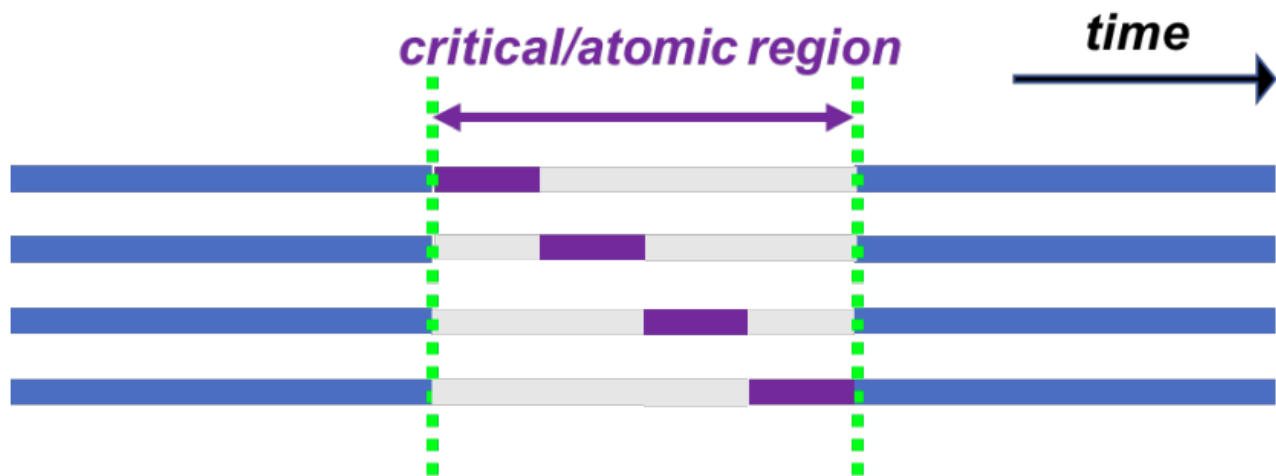
`#pragma omp single`：某段代码单线程执行，带隐式同步（使用nowait去掉）



`#pragma omp master` : 采用主线程执行，无隐式同步



`#pragma omp critical` : 某段代码线程互斥执行



`#pragma omp atomic` : 单个特定格式的语句或语句组中某个变量进行原子操作

更多特性

任务构造

task [2.9.1] [2.11.1]

Defines an explicit task. The data environment of the task is created according to data-sharing attribute clauses on **task** construct and any defaults that apply.

```
#pragma omp task [clause [ , ] clause ] ...]  
    structured-block
```

clause:

```
    if([ task : ] scalar-expression)  
    final(scalar-expression)  
    untied  
    default(shared | none)  
    mergeable  
    private(list)  
    firstprivate(list)  
    shared(list)  
    depend(dependence-type : list)  
    priority(priority-value)
```

- 前述的构造都遵循Fork-Join模式，对任务类型有限制
- 任务（task）构造允许定义任务以及依赖关系，动态调度执行
- 即动态管理线程池（thread pool）和任务池（task pool）

向量化

simd [2.8.1] [2.8.1]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

```
#pragma omp simd [clause [ , ]clause ...]  
    for-loops
```

clause:

safelen(*length*)

simdlen(*length*)

linear(*list* [: *linear-step*])

aligned(*list* [: *alignment*])

private(*list*)

lastprivate(*list*)

reduction(*reduction-identifier* : *list*)

collapse(*n*)

- 将循环转换为SIMD循环
- `aligned` 用于列出内存对齐的指针
- `safelen` 用于标记循环展开时的数据依赖

编译器也自带向量化功能，例如gcc：

- `-O3`
- `-ffast-math`
- `-fivopts`
- `-march=native`
- `-fopt-info-vec`
- `-fopt-info-vec-missed`

GPU支持

- 参考[网页](#)
- 从OpenMP API 4.0开始支持

参考资料

- 杨超，北京大学课程《并行与分布式计算基础》课件
- Schmidt B, Gonzalez-Dominguez J, Hundt C, et al. Parallel programming: concepts and practice[M]. Morgan Kaufmann, 2017.
- [API reference](#)
- [C/C++ Reference Guide](#)