清華大學
Tsinghua University

高性能计算导论
第5讲：共享内存编程模型

翟季冬

计算机系

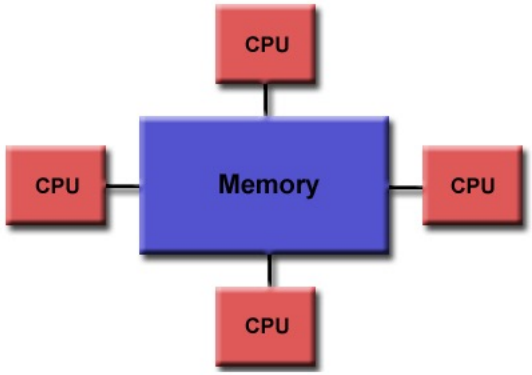
目录

- OpenMP 概述
- 并行指导语句
  - 并行区结构体
  - 任务分配结构体
- 数据共享与线程同步
  - 数据共享
  - 线程同步
  - 归约
- 编译运行命令
- MPI 对比 OpenMP
- OpenMP + MPI 混合编程

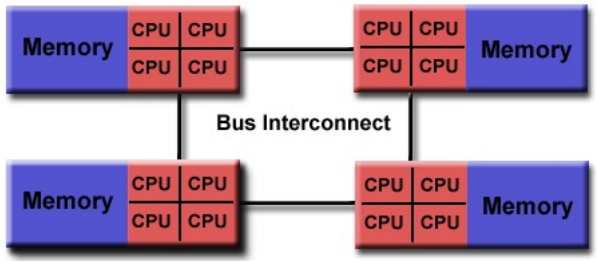
OPENMP 概述

回顾：共享内存（Shared Memory）

- 含义：所有处理单元与共享内存相连，处于同一个地址空间下，任何处理单元可通过地址直接访问任意内存位置



均匀内存访问架构-UMA



非均匀内存访问架构-NUMA

OpenMP 概述

- OpenMP
  - Open Multi-Processing 缩写
- 显式指导多线程、共享内存并行的 API
  - 轻量级、可移植的语法标准
  - 增量式并行
  - 需要编译器支持（C、C++ 或 Fortran）
- OpenMP 并非
  - 自动并行化
  - 保证加速
  - 自动避免数据竞争

OpenMP 历史

1990 年代
- 共享内存计算机厂商往往各自提供相似而不同的 Fortran 语言扩展以进行并行编程

1994 年
- ANSI X3H5 曾尝试统一这些标准，但未能成功，主要由于当时分布式内存计算机更加流行

- 随着新一代共享内存计算机的成熟，人们重新开始关注共享内存并行编程

1997 年
- OpenMP 标准被提出，取代了 ANSI X3H5 的位置

如今
- OpenMP 标准依然在不断演进
- http://www.openmp.org

OpenMP 设计理念：增量式并行

- 先编写串行程序，然后添加并行化指导语句
  - 如同写了一句注释

```
int main() {

  /** 并行化下面的代码 */
  printf( "Hello, World!\n" );

  return 0;
}
```

```
int main() {
  /** 并行化下面的代码 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }
  return 0;
}
```

OpenMP 组成

1. 编译器预处理指导语句（~80%）
2. 函数调用（~19%）
3. 环境变量（~1%）

```
int main() {
  /** 并行化下面的代码 */
  omp_set_num_threads(16); //函数调用

  #pragma omp parallel //预处理指导语句
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

环境变量：**OMP_NUM_THREADS** 环境变量

OpenMP 组成

- 编译器预处理指导语句

#pragma omp	指导语句名	[子句 ...]	换行
必须	parallel, for 等	可选，可以有若干个，无顺序	必须

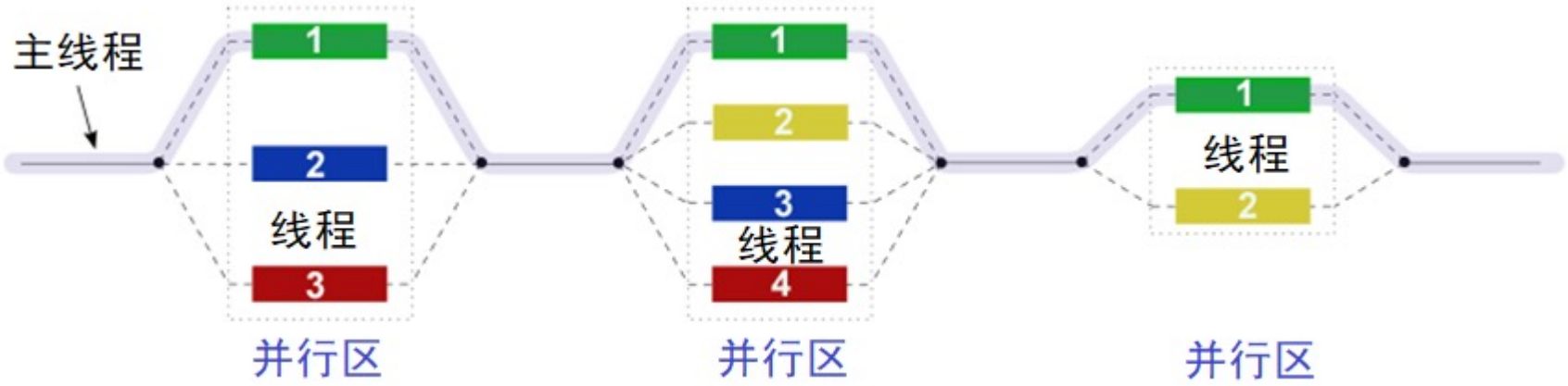
- 例

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

指导语句名　　子句　　子句

- 基本规则：
  - 区分大小写
  - 指导语句作用于指导语句之后的语句块（例如循环）

Fork-Join 模型




```
int main() {
  omp_set_num_threads(3);
  #pragma omp parallel
    printf( "Hello, World-1!\n" );

  omp_set_num_threads(4);
  #pragma omp parallel
    printf( "Hello, World-2!\n" );

  return 0;
}
```

对比：共享内存 pthread 编程

- 例子：Hello World

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

static const int NUM_THREADS = 3;

void* thread_func(void* data) {
    int tid = *(int*)data;
    printf("[%d/%d] Hello\n", tid, NUM_THREADS);

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Create
    for (int tid = 0; tid < NUM_THREADS; ++tid) {
        printf("[Main] Create thread %d\n", tid);
        thread_ids[tid] = tid;
        pthread_create(&threads[tid], NULL, thread_func, &thread_ids[tid]);
    }

    // Join
    for (int tid = 0; tid < NUM_THREADS; ++tid) {
        printf("[Main] Join thread %d\n", tid);
        pthread_join(threads[tid], NULL);
    }

    printf("[Main] Exit\n");
    return 0;
}
```

线程主函数

创建线程

等待线程结束

并行指导语句

并行指导语句

—并行区结构体

并行区结构体

- 并行区结构体：创建一组 OpenMP 线程以执行一个并行区

```
#pragma omp parallel [子句[ [,]子句] …]
    语句块
```

子句：

- private(列表)
- firstprivate(列表)
- shared(列表)
- copyin(列表)
- reduction([归约修饰符,] 归约操作符: 列表)
- proc_bind(master | close | spread)
- allocate([分配器: 列表)
- if([parallel:] 标量表达式)
- num_thread(整数表达式)
- default(shared | none)

*详细说明及用法请参见文档

```
int main() {
  omp_set_num_threads(16);

  /** 并行区 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

并行区结构体 – 语义

- 程序运行至 **parallel** 时，创建一组线程
- 所有线程均执行并行区内的代码
- 并行区结束时，所有线程会同步（有一个隐式 barrier）
- 限制：
  - 并行区须为函数内的一个语句块
  - 不能跳转（goto）出入并行区
  - 但并行区内可以调用其他函数

```
int main() {
  omp_set_num_threads(16);

  /** 并行区 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

并行区：线程数量

- 调整并行区内线程数量的方式

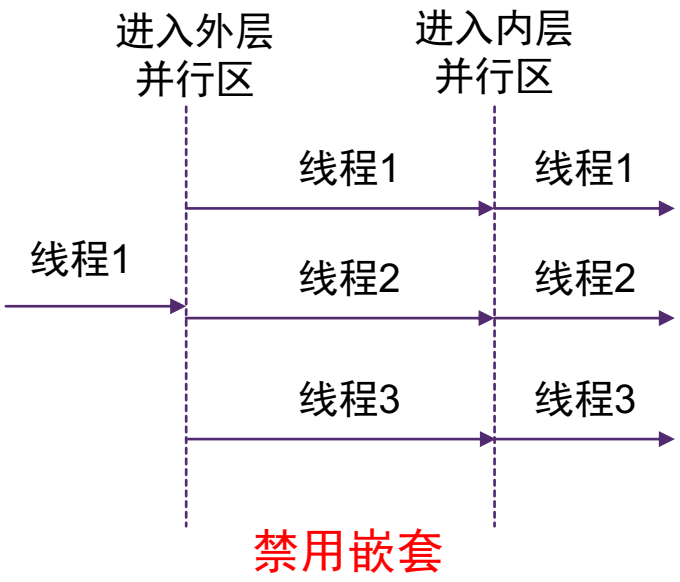
1. 添加 **num_threads** 子句
   - 例：**#pragma omp parallel num_threads(10)**
2. 调用 **omp_set_num_threads()** 函数
   - 在并行区前调用
3. 设置 **OMP_NUM_THREADS** 环境变量
   - 在并行区前调用
4. 添加 **if** 子句
   - 条件为 **false** 时，仅用主线程串行执行
   - 例：**#pragma omp parallel if (para == true)**

- 优先级：
  - pragma 里面指定 > 函数API设置 > 环境变量设置

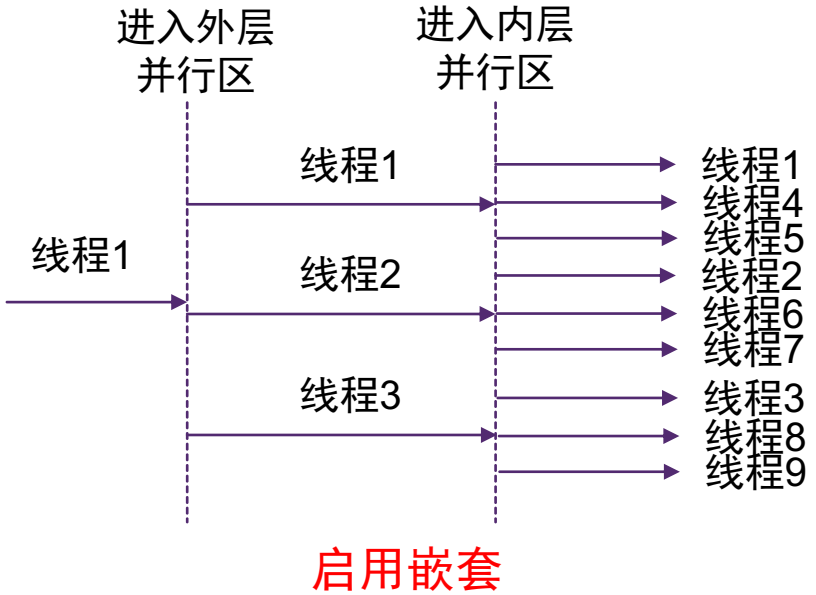
并行区结构体：嵌套并行区

- 除非启用嵌套并行区，内层并行区**只用一个线程**执行
- 启用/禁用嵌套并行区：
  - `omp_set_nested(bool)`
  - 设置 `OMP_NESTED` 环境变量
- 检查嵌套并行区是否开启：
  - `omp_get_nested()`

```
#pragma omp parallel num_threads(3)
{
#pragma omp parallel num_threads(3)
  { // ...
  }
}
```



禁用嵌套



启用嵌套

并行指导语句

—任务分配结构体

任务分配结构体

- 任务分配结构体（Work-Sharing Construct）
  - 将代码区域中的任务自动分配给不同线程执行

注意：

- 任务分配结构体不创建新线程
- 进入任务分配结构体时，各线程不同步
  - 没有隐式 barrier
- 离开任务分配结构体时，各线程同步
  - 有隐式 barrier

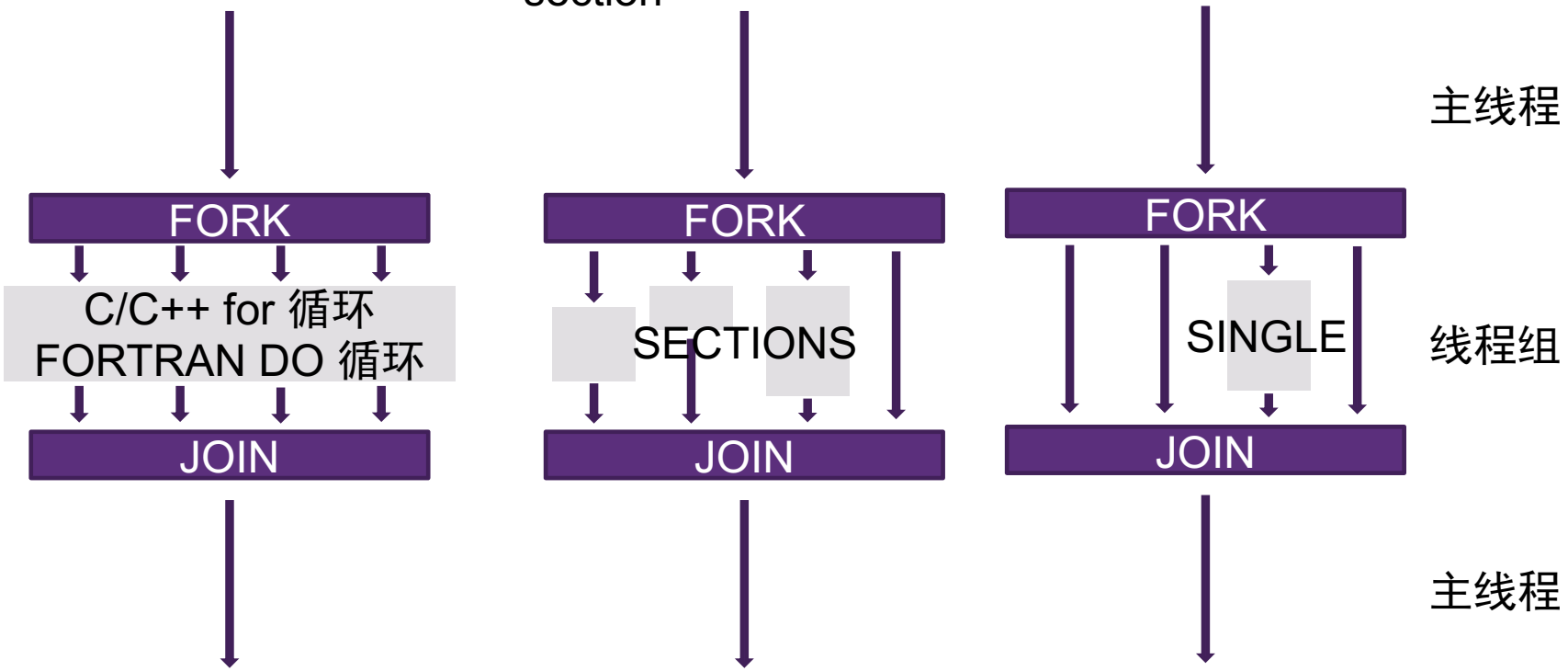
任务分配结构体

- 三种典型的任务分配结构体
  - 注意：任务分配结构体需要包含在并行区内部

DO / for: 将循环的各次迭代分配给不同线程。是一种数据并行

sections: 将任务划分成若干个 section，每个线程执行各自的 section

single: 仅使用并行区中的一个线程执行



任务分配结构体

- 将循环的各次迭代自动分配给并行区线程组中的各个线程执行

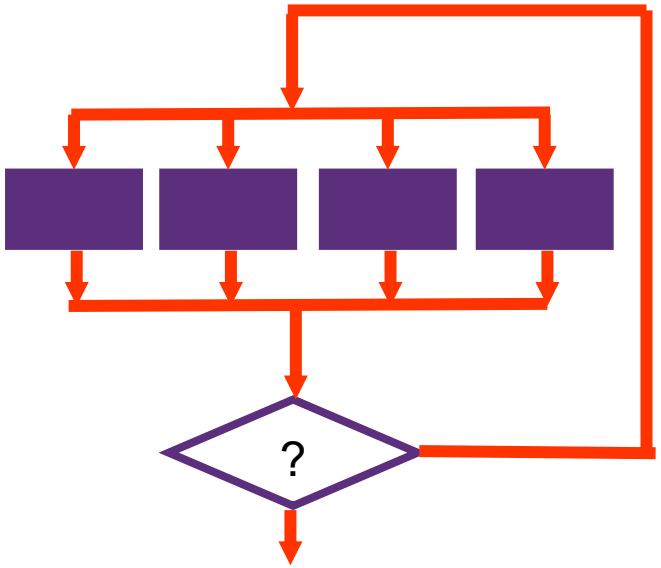
```
#pragma omp for [子句[ [,]子句] …]
    for 循环
```

子句：

- private(列表)
- firstprivate(列表)
- lastprivate([lastprivate修饰符：]列表)
- linear(列表[： 步长])
- schedule([调度修饰符 [,调度修饰符]:]调度策略 [,块大小])
- collapse(n)
- ordered[(n)]
- allocate([分配器： 列表)
- order(concurrent)
- reduction([归约修饰符,] 归约操作符： 列表)
- nowait

*详细说明及用法请参见文档

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < 25; i++) {
  printf("foo");
}
```



任务分配结构体

- 简便写法：以下两种写法是等价的

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < 25; i++) {
    printf("Foo");
  }
}
```

```
#pragma omp parallel for
for (i = 0; i < 25; i++) {
  printf("Foo");
}
```

任务如何分配？

任务分配结构体：调度方式

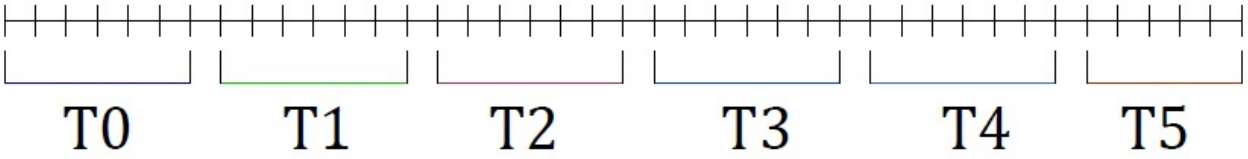
- 调度策略（ schedule 子句）：任务分配方式

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

static 调度

- 设有N_i次迭代和N_t个线程，为每个线程分配包含N_i/N_t次迭代的一块



- 线程 T_0：迭代 $0 \sim \frac{N_i}{N_t} - 1$
- 线程 T_1：迭代 $\frac{N_i}{N_t} \sim 2\frac{N_i}{N_t} - 1$
- 线程 T_2：迭代 $2\frac{N_i}{N_t} \sim 3\frac{N_i}{N_t} - 1$
- ……
- 线程 $T_{N_t - 1}$：迭代 $(\mathrm{N_t} - 1)\frac{N_i}{N_t} \sim N_i - 1$

任务分配结构体：调度方式

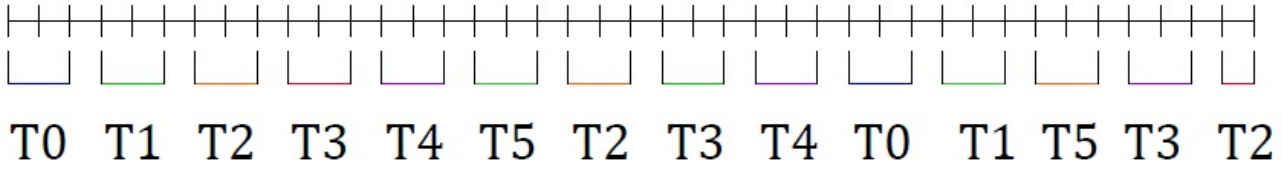
- 调度策略（schedule 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

dynamic 调度

- 设有N_i次迭代和N_t个线程，每个线程被分配若干个包含 k 次迭代的块



- 当某个线程完成了一块时，该线程会被立即分配新的一块
- 因此，迭代与线程的对应关系是不确定的
- 优点：灵活
- 缺点：高开销—分配过程耗时很长

任务分配结构体：调度方式

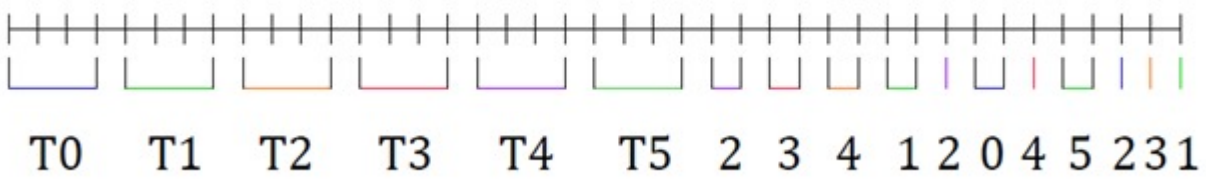
- 调度策略（**schedule** 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. **guided**
   - 与 **dynamic** 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

guided 调度

- 设有N_i次迭代和N_t个线程，一开始每个线程被分配一个包含 k 次迭代的块；
- 某个线程完成后，该线程再被分配包含 $k/2$ 个迭代的块；
- 再完成后，会被分配包含 $k/4$ 个迭代的块；以此类推



- 特点：
- 相比 **static**：更好的负载均衡、更多开销
- 相比 **dynamic**：更少开销、没那么好的负载均衡

任务分配结构体：调度方式

- 调度策略（schedule 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：嵌套循环

- **collapse** 子句：将嵌套循环的迭代统一调度，而不是串行执行内层循环

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      printf("i=%d, j=%d, thread = %d\n",
             i, j, omp_get_thread_num());
```

```
i=1, j=0, thread = 1
i=2, j=0, thread = 2
i=0, j=0, thread = 0
i=1, j=1, thread = 1
i=2, j=1, thread = 2
i=0, j=1, thread = 0
i=1, j=2, thread = 1
i=2, j=2, thread = 2
i=0, j=2, thread = 0
```

仅仅外层循环被并行、只有3个线程

任务分配结构体：嵌套循环

- **collapse** 子句：将嵌套循环的迭代统一调度，而不是串行执行内层循环

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      printf("i=%d, j=%d, thread = %d\n",
             i, j, omp_get_thread_num());
```

```
i=1, j=0, thread = 1
i=2, j=0, thread = 2
i=0, j=0, thread = 0
i=1, j=1, thread = 1
i=2, j=1, thread = 2
i=0, j=1, thread = 0
i=1, j=2, thread = 1
i=2, j=2, thread = 2
i=0, j=2, thread = 0
```

仅仅外层循环被并行、只有3个线程

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
                    collapse(2)
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      printf("i=%d, j=%d, thread = %d\n",
             i, j, omp_get_thread_num());
```

```
i=0, j=0, thread = 0
i=0, j=2, thread = 1
i=1, j=0, thread = 2
i=2, j=0, thread = 4
i=0, j=1, thread = 0
i=1, j=2, thread = 3
i=2, j=2, thread = 5
i=1, j=1, thread = 2
i=2, j=1, thread = 4
```

内层和外层循环统一调度、6个线程

任务分配结构体：sections

- 任务分配方式
- 将不同片段的代码组成不同 section，再分配给不同线程
- 各 **section** 命令需要包含在一个 **sections** 命令中
- 每个 section 只被一个线程执行一次
- 线程和 section 的对应关系是不确定的

```
#pragma omp sections [子句[ [,]子句] …]
    {
    [#pragma omp section]
        语句块
    [#pragma omp section]
        语句块
    …
    }
```

子句：

private(列表)
firstprivate(列表)
lastprivate([lastprivate修饰符：]列表)
allocate([分配器： 列表)
reduction([归约修饰符,] 归约操作符：列表)
nowait

*详细说明及FORTRAN用法请参见文档

任务分配结构体：sections

```
#pragma omp parallel num_threads(2)
{
#pragma omp sections
  {
#pragma omp section // 第一个 section
    {
      for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
    }
#pragma omp section // 第二个 section
    {
      for (int i = 0; i < n; i++)
        d[i] = a[i] + b[i];
    }
  }
}
```

任务分配结构体：single

- 令某一段代码仅被并行区中的一个线程执行
- 可用于执行非线程安全的操作（例如 I/O）
- 不执行此代码段的线程将等待此代码段执行结束（除非用了 **nowait** 子句）

```
#pragma omp parallel num_threads(10)
{
#pragma omp single
  {
    scanf("%d", &input);
  }
  printf("input is %d", input);
}
```

默认存在隐式的同步

数据共享与线程同步

数据共享与线程同步

—数据共享

数据共享

```
#pragma omp parallel num_threads(10) {
#pragma omp single
  {
    scanf("%d", &input);
  }
  printf("input is %d", input); }
```

- 理解数据的作用域十分重要
  - OpenMP 属于共享内存编程模型，大多数变量默认是共享的
- 默认线程间共享变量：
  - 全局变量、static 变量
  - 除了并行循环的循环下标外，所有并行区内、任务分配结构体外局部变量
- 默认线程内私有变量：
  - 并行循环的循环下标
  - 任务分配结构体内的局部变量
  - 并行区中调用的函数中的局部变量
- 作用域可以通过子句显式控制
  - **private**, **shared**, **firstprivate**, **lastprivate**
  - **default**, **reduction**, **copyin**

数据共享

- **private(var_list)**
  - 将列表中的变量声明为线程私有（创建私有变量）
  - 变量在进入并行区时不会初始化，离开并行区后不会保留
- **shared(var_list)**
  - 将列表中的变量声明为线程间共享
- **firstprivate(var_list)**
  - 与 **private** 类似，但变量在进入并行区时会被初始化成进入并行区前的值
- **lastprivate(var_list)**
  - 与 **private** 类似，但变量在离开并行区后，其在最后一次迭代时的值会被保留

数据共享

- **private** 子句举例

```
void work(flaot *c, int N) {
  float x, y;
#pragma omp parallel for private(x, y)
  for (int i = 0; i < N; i++) {
    x = a[i];
    y = b[i];
    c[i] = x + y;
  }
}
```

数据共享

- 点乘
- 有什么错误？

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
    sum = sum + a[i] * b[i];
  }
  return sum;
}
```

数据竞争 Data race

数据共享与线程同步

—线程同步

线程同步

- 对存在数据竞争（data race）的变量访问需要保护
  - 同时只有1个线程对共享变量进行修改

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
#pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

线程同步

线程同步指导语句：

- **#pragma omp critical**
  - 只有一个线程执行临界区

```
#pragma omp critical
{ /** 临界区代码*/ }
```

- **#pragma omp barrier**
  - 等待所有线程到达此处后继续运行

```
#pragma omp barrier
```

- **#pragma omp single**
  - 任务分配结构体的一种：只用一个线程执行代码
  - 也可以用 **#pragma omp master**：只用主线程执行一段代码
    - 执行完毕后没有隐式 barrier
    - 比 **single** 命令更高效

```
#pragma omp single // 或 master
{ /** 仅被一个线程执行 */ }
```

- **#pragma omp atomic**
  - 使用原子指令访问共享变量
  - 单条语句、性能更高

```
#pragma omp atomic
a[x] += b[y]; // 支持原子操作的运算
```

线程同步

- OpenMP 的锁函数：

```
omp_set_lock(l);
/* 临界区代码 */
omp_unset_lock(l);
```

- **void omp_init_lock(omp_lock_t *lock)**
  - 初始化锁
- **void omp_destroy_lock(omp_lock_t *lock)**
  - 销毁锁
- **void omp_set_lock(omp_lock_t *lock)**
  - 获取锁。当锁已被其他线程获取时，则等待其他线程释放锁
- **void omp_unset_lock(omp_lock_t *lock)**
  - 释放锁
- **int omp_test_lock(omp_lock_t *lock)**
  - 测试某个锁是否已被获取。此函数不会阻塞

线程同步

- OpenMP 对共享变量维持“放松的一致性”
  - 各线程中，某一共享变量的值不总是最新的
- 当需要获取一个共享变量的最新值时，程序员须保证变量被 flush
- 下列命令会自动进行 flush：
  - **parallel**（进出）、**critical**（进出)、**ordered**（进出）、**for**（出）、**sections**（出）、**single**（出）
- 手动 flush：

```
#pragma omp flush [内存序] [(变量列表)]
```

内存序：

acq_rel, release, acquire

*详细说明及用法请参见文档

线程同步

- 若干 OpenMP 结构体具有隐式 barrier，例如
  - **parallel**, **for**, **single**
- 若已知不需要 barrier，可通过 **nowait** 子句取消 barrier

```
#pragma omp for nowait
  for (int i = 0; i < n; i++)
    a[i] = bigFunc1(i);
#pragma omp for nowait
  for (int j = 0; j < m; j++)
    b[j] = bigFunc2(j);
```

线程同步

OpenMP 同步方法：

- Barrier
  - 等待所有线程到达
  - 适用于等待一个计算步骤结束以开始下一步骤
- 临界区（底层采用锁实现）
  - 保护共享资源的简便方法
  - 可以多行代码
- 显式的锁操作
  - 保护共享资源
  - 比临界区灵活但繁琐，实现复杂同步操作时可能需要
- 原子操作 Atomic
  - 保护共享资源
  - 轻量级，适用于单个操作：单个语句
- 单线程任务分配结构体
  - 保护共享资源
  - 用于 `omp parallel` 语句块中（`omp for` 之外）

数据共享与线程同步

—归约

归约

- 问题是什么？是否有更高效的方法？

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
#pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

归约

- 使用 **reduction** 子句

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

是否想到 MPI_Reduce 函数

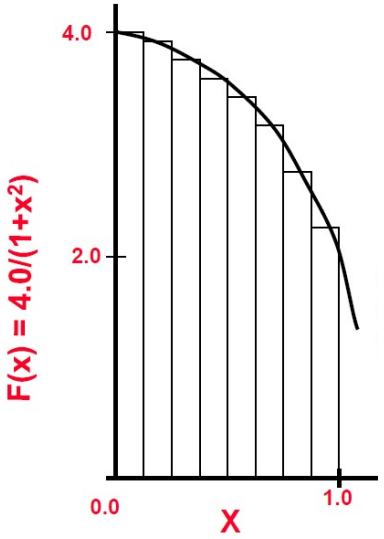
归约

- **`reduction(op: list)`**
- **`list`** 中的变量需为共享变量
- 实现原理：
  1. 首先，各线程各自在私有变量上进行局部归约
  2. 任务结束时，局部结果再被跨线程地归约成全局结果

归约

- 性能对比

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$




```
static long num_steps = 100000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step; //中点
  sum += 4.0 / (1.0 + x * x); //计算函数值
}
double pi = step * sum; //乘以高度
```

串行伪代码

归约

- 方案1：**reduction**

```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

归约

- 方案2： critical

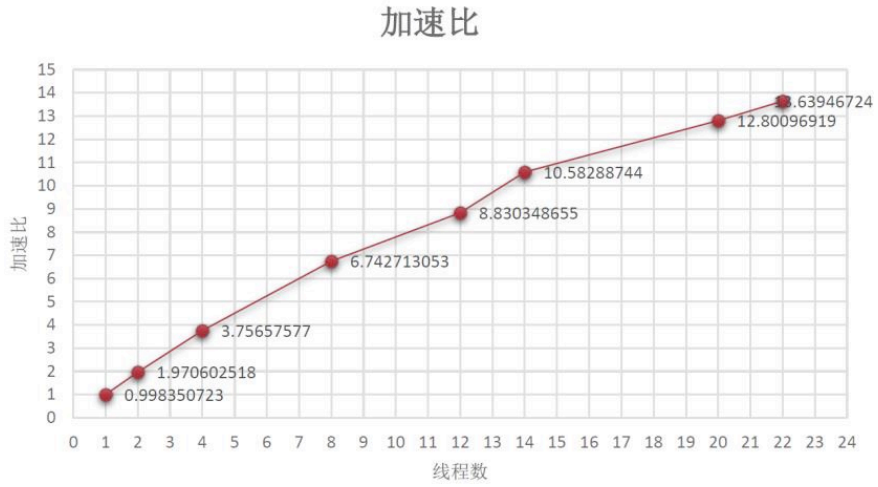
```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
#pragma omp critical
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

归约

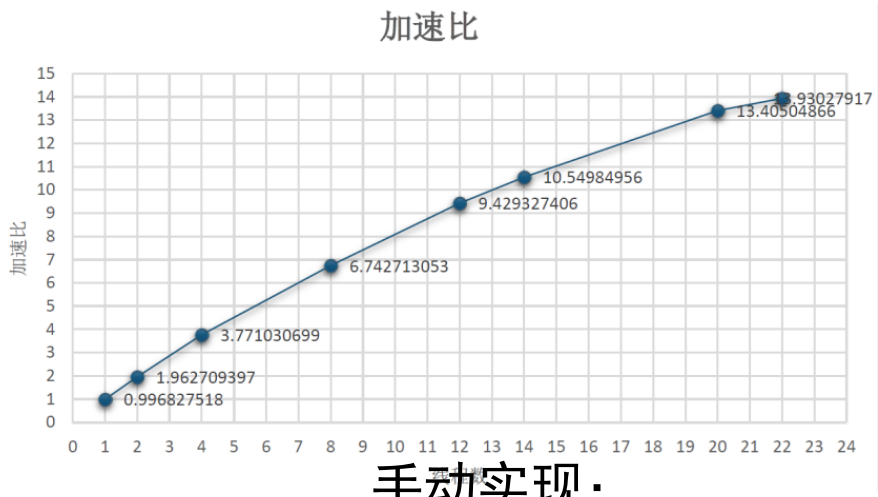
- 方案3： atomic

```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel for
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
#pragma omp atomic
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

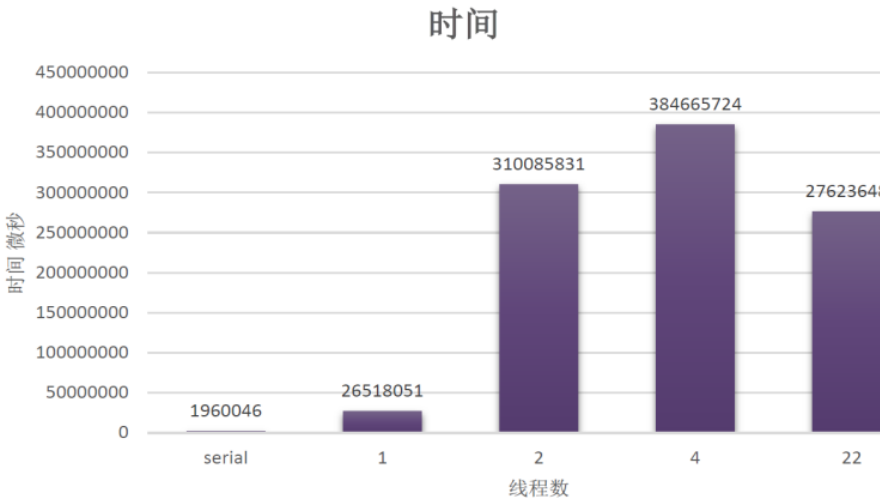
不同归约性能对比



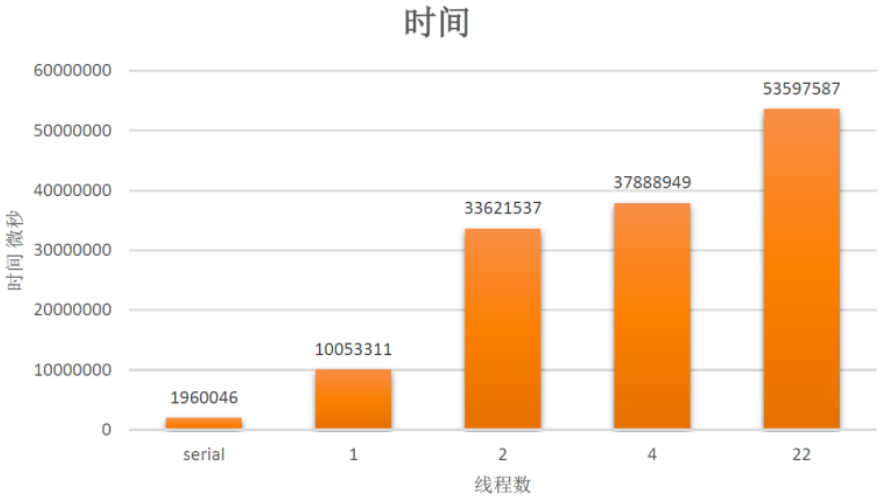
方案1： **reduction**



手动实现：
先求局部和，再串行求全局和



方案2： **critical**



方案3： **atomic**

归约

- 使用 **reduction** 子句性能最优
  - 仅在循环结束后进行同步
  - 与先求局部和，再串行求全局和的手动实现性能相当
- 使用 **critical** 或 **atomic** 使并行性能反而远不如串行
  - 每次迭代均进行同步
  - 同步开销过大
  - Critial比Atomic的性能还差
  - 注意：加速比

编译运行命令

OpenMP 示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /** 线程函数 */

int main(int argc, char* argv[]) {
    /** 从命令行得到线程数 */
    int thread_count = strtol(argv[1], NULL, 10);
    #pragma omp parallel num_threads(thread_count)
    Hello();
    return 0;
} /** main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /** Hello */
```

头文件

并行指导语句

函数调用-线程 ID

线程数

编译运行

```
gcc  -g  -Wall  -fopenmp  -o  main  main.c
```

因编译器不同而不同

```
OMP_NUM_THREADS=4  ./main
```

环境变量设置

可能的输出

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

编译运行

编译器	编译命令	OpenMP 选项
Intel	`icc`	`-qopenmp`

GNU	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

PGI	`pgcc`
	`pgCC`
	`pgf77`
	`pgf90`
		`-mp`

Clang	`clang` `clang++`	`-fopenmp`

常用的库函数

- `void omp_set_num_threads(int num_threads)`
  - 设置后续并行区的线程数
- `int omp_get_num_threads()`
  - 获取当前设置的线程数
- `int omp_get_thread_num()`
  - 获取当前线程的编号
  - 主线程编号为 0
- `int omp_get_thread_limit()`
  - 获取程序可用的最大线程数
- `int omp_get_num_procs()`
  - 获取程序可用的处理器数量
- `int omp_in_parallel()`
  - 判断当前代码是否在并行区中
- 更多函数请查阅 OpenMP 文档

常用的环境变量

- **OMP_NUM_THREADS**
  - 设置并行区线程数
- **OMP_PROC_BIND**
  - 设置线程与处理器的绑定关系
  - 可设为 **true**（绑定）或 **false**（不绑定）
  - 也可通过 **master**, **close** 和 **spread** 为 NUMA 架构设定更具体的绑定策略（详见文档）
- **OMP_WAIT_POLICY 线程同步等待策略**
  - 设为 **ACTIVE** 表示利用自旋锁进行线程间等待
    - **低延迟，等待需要消耗CPU时间**
  - 设为 **PASSIVE** 表示利用操作系统调度进行线程间等待
    - **高延迟，等待不消耗 CPU 时间**
- 更多环境变量请查阅 OpenMP 文档

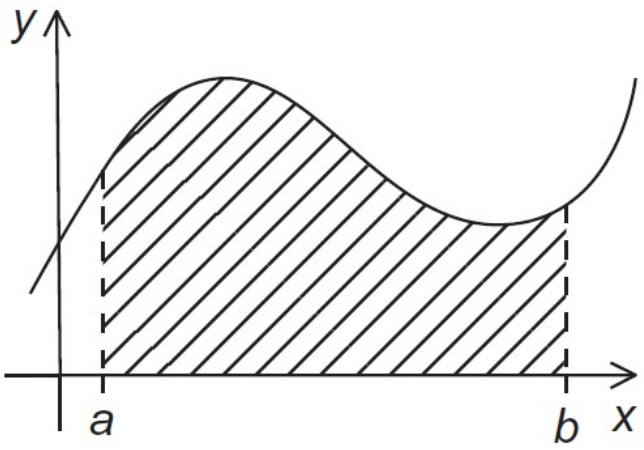
常用的环境变量

• OMP_CANCELLATION:		Set whether cancellation is activated
• OMP_DISPLAY_ENV:		Show OpenMP version and environment variables
• OMP_DEFAULT_DEVICE:		Set the device used in target regions
• OMP_DYNAMIC:		Dynamic adjustment of threads
• OMP_MAX_ACTIVE_LEVELS:		Set the maximum number of nested parallel regions
• OMP_MAX_TASK_PRIORITY:		Set the maximum task priority value
• OMP_NESTED:		Nested parallel regions
• OMP_NUM_THREADS:		Specifies the number of threads to use
• OMP_PROC_BIND:		Whether theads may be moved between CPUs
• OMP_PLACES:		Specifies on which CPUs the theads should be placed
• OMP_STACKSIZE:		Set default thread stack size
• OMP_SCHEDULE:		How threads are scheduled
• OMP_THREAD_LIMIT:		Set the maximum number of threads
• OMP_WAIT_POLICY:		How waiting threads are handled
• GOMP_CPU_AFFINITY:		Bind threads to specific CPUs
• GOMP_DEBUG:		Enable debugging output
• GOMP_STACKSIZE:		Set default thread stack size
• GOMP_SPINCOUNT:		Set the busy-wait spin count
• GOMP_RTEMS_THREAD_POOLS:		Set the RTEMS specific thread pools

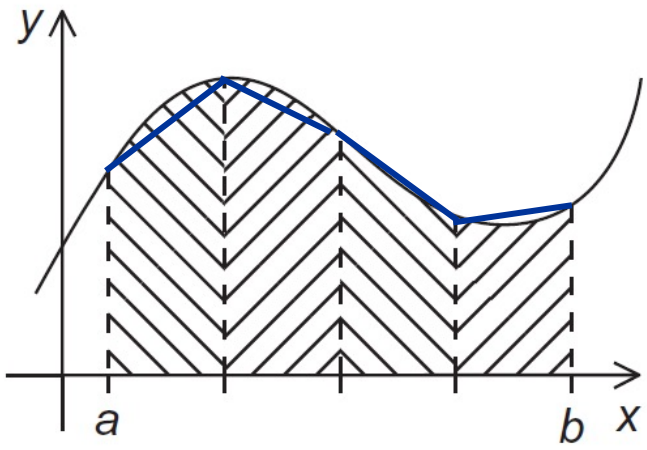
MPI 对比 OPENMP

回顾：The Trapezoidal Rule（梯形法则）

- 用复合梯形法则来计算积分



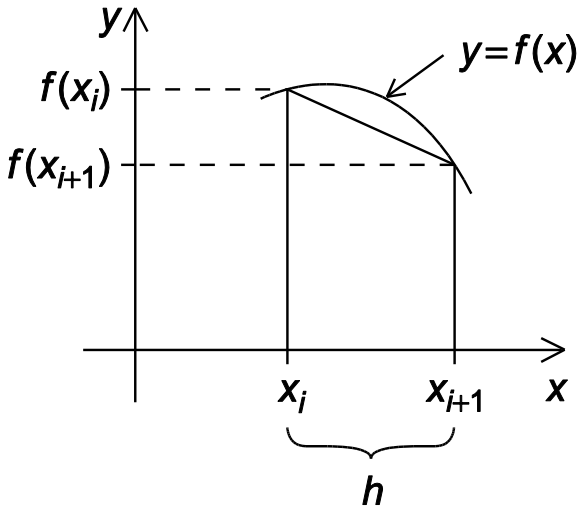
(a)



(b)

回顾：The Trapezoidal Rule（梯形法则）

区间[a，b]分为n个子区间



一个梯形的面积 = $\frac{h}{2}[f(x_i) + f(x_{i+1})]$

$$h = \frac{b-a}{n}$$

$x_0 = a, x_1 = a + h, x_2 = a + 2h, \ldots, x_{n-1} = a + (n-1)h, x_n = b$

所有梯形面积的总和 =

$$h\left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

回顾：The Trapezoidal Rule（梯形法则）

- 串行伪代码

```
/** 输入 a, b, n */
Get a, b, n;
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i * h;
   approx += f(x_i);
}
approx = h * approx;
```

MPI代码：The Trapezoidal Rule（梯形法则）

```
int main(){
    int my_rank, comm_size, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_integral, total_integral;
    int source;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
```

```
    /** 将计算任务划分至各个进程中 */
    h = (b - a) / n;
    local_n = n / comm_size; /** local_n 是每个进程计算的梯形数 */
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
```

```
    /** 每个进程分别计算对应的计算任务 */
    local_integral = Trap(local_a, local_b, local_n, h);
```

MPI代码：The Trapezoidal Rule（梯形法则）

```
    /** 将各个进程的 local_integral 汇总至 0号进程 */
    if (my_rank != 0) {
        MPI_Send(&local_integral, 1, MPI_DOUBLE, 0, 0,
                                                MPI_COMM_WORLD);
    } else {
        total_integral = local_integral;
        for (source = 1; source < comm_size; source ++) {
            MPI_Recv(&local_integral, 1, MPI_DOUBLE, source, 0,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_integral += local_integral;
        }
    }
```

从每个进程接收消息

```
    /** 0号进程输出汇总结果 */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n", a, b,
                                                total_integral);
    }
    MPI_Finalize();
    return 0;
} /** main */
```

0号进程输出结果

OpenMP 程序：The Trapezoidal Rule（梯形法则）

- 对比 MPI 代码
- 串行代码

```
/** 输入 a, b, n */
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i * h;
    approx += f(x_i);
}
approx = h * approx;
```

- OpenMP 代码

```
/** 输入 a, b, n */
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
#pragma omp parallel for reduction(+:approx) private(x_i)
for (i = 1; i <= n-1; i++) {
    x_i = a + i * h;
    approx += f(x_i);
}
approx = h * approx;
```

增量式并行、并行代码撰写非常简单（半自动并行）

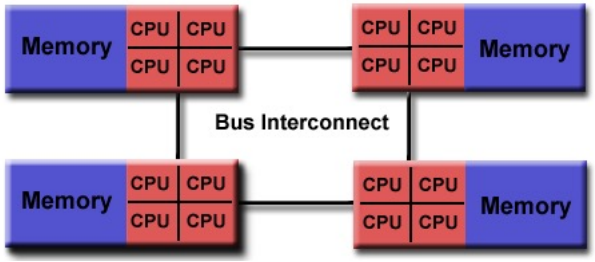
OPENMP + MPI 混合编程

OpenMP + MPI 混合编程

- 标准的编程模型：
  - 共享内存：共享内存模型—OpenMP
  - 分布式内存：消息传递模型—MPI
- 典型集群系统：
  - 结点内共享内存
  - 结点间分布式内存
- 问题1：如何应对计算机集群？
  - OpenMP + MPI 混合编程

OpenMP + MPI 混合编程

- 回顾：非均匀内存访问架构（NUMA）
  - 一般包括多个对称多处理器（SMP）服务器
  - 每个SMP拥有自己局部内存
  - 跨SMP的远程内存访问代价较高
- 问题2：如何应对 NUMA？
  - OpenMP + MPI 混合编程



NUMA

OpenMP + MPI 混合编程

- 程序编程示例：

```
#include <mpi.h>
int main(int argc, char **argv) {
  MPI_Init_thread(...);

#pragma omp parallel for // Compute
  for(i=0; i<n; i++){
  ...
  }
  MPI_Sendrecv(...) // Communicate

#pragma omp parallel for // Compute
  for(i=0; i<n; i++){
  ...
  }
  MPI_Finalize();
}
```

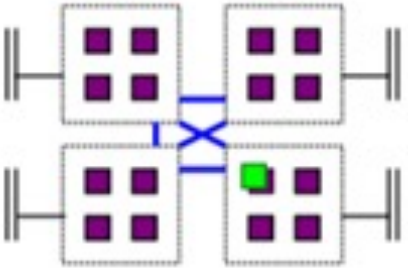
OpenMP + MPI 混合编程

- 调用 MPI 时应考虑线程安全性
- 使用 **`MPI_Init_thread`** 以初始化 MPI，并设置线程安全级别
  - 自 MPI-2 开始支持，替代 **`MPI_Init`**
- 线程安全级别（MPI_Init_thread 候选参数）：
  - **`MPI_THREAD_SINGLE`**：单线程
  - **`MPI_THREAD_FUNNELED`**：只有主线程允许调用 MPI
  - **`MPI_THREAD_SERIALIZED`**：所有线程均可调用 MPI，但用户须保证同一时刻只进行一次调用
  - **`MPI_THREAD_MULTIPLE`**：提供线程安全的 MPI 接口

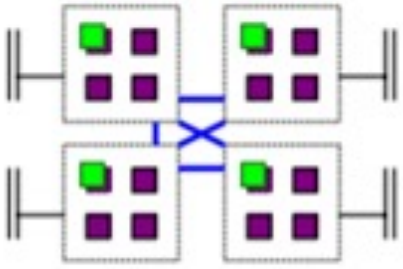
OpenMP + MPI 混合编程

- NUMA 结点中的不同的混合编程配置（三种方式）：

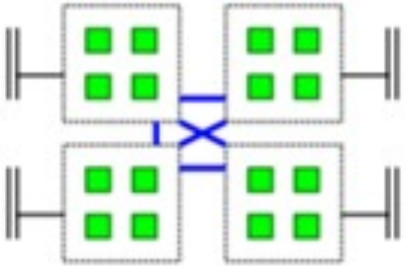
1 MPI 进程
16 OpenMP 线程



4 MPI 进程
4 OpenMP 线程（每进程）



16 MPI 进程



- 为了实现不同配置，需要：
  - 正确设定进程/线程与处理器的绑定关系
  - 将内存分配到正确的 NUMA 结点上

OpenMP + MPI 混合编程

- 运行命令举例（Open MPI）

```
mpirun --bind-to socket -np 4 \
    -x OMP_NUM_THREADS=4 \
    -x OMP_PROC_BIND=true \
    numactl -l ./main
```

- 请查阅 MPI、OpenMP 和 numactl 手册，按实际需求绑定进程和线程
- 不同 MPI 实现的选项有所差异

其它共享内存编程框架/库

- Pthreads：POSIX 标准的线程
  - 相对底层（Linux 中的 OpenMP 基于 Pthreads 实现）
  - 未经优化前可能效率低：反复创建线程
- TBB：Thread Building Block
  - Intel 开发的多线程编程框架
- CILK
  - 轻量级的 C 语言多线程编程框架

其他共享内存编程框架/库

- Pthreads 代码示例

```
typedef struct {
  int id;
} parm;

void *hello(void *arg)
{
  parm *p=(parm *)arg;
  printf("Hello from node %d\n", p->id);
  return NULL;
}

void main() {
  pthread_t *threads=(pthread_t *)malloc(n*sizeof(*threads));
  pthread_attr_t pthread_custom_attr;
  pthread_attr_init(&pthread_custom_attr);
  parm *p=(parm *)malloc(sizeof(parm)*n);
  for (int i = 0; i < 8; i++) {
    p[i].id=i;
    pthread_create(&threads[i], &pthread_custom_attr, hello, (void *)(p + i));
  }
  for (int i = 0; i < 8; i++) {
    pthread_join(threads[i],NULL);
  }
  free(p);
}
```

总结

- OpenMP
  - 是一种在共享内存计算机中实现并行的编程方法，尤其适用于并行化循环
  - 采用了 Fork-Join 模型
  - 是基于编译器的编程框架
- 使用 OpenMP 编程注意：
  - 定义并行区（**`omp parallel`**）
  - 设置并行度
  - 采用合适的并行结构（**`for`**; **`sections`**; **`single`**）
  - 设置调度策略（**`schedule`**）
  - 数据管理 / 变量分类（ **`private`** / **`shared`**; **`flush`** ）
  - 同步控制（**`critical`**等）

课后阅读和练习

- **OpenMP Tutorial**
  - https://hpc-tutorials.llnl.gov/openmp/
  - 各种子句的详细语法说明、参考例子
  - 认真阅读
- **OpenMP API Cheat Sheet**
  - https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf
- 课后练习
  - 完成 72 页的 OpenMP 代码并编译运行

小作业

- 测量 OpenMP 并行 for 循环不同调度策略的性能差别
  - 使用 OpenMP 并行 for 循环并行以下程序，并测量不同调度选择带来的性能差异
  - 给定长度为 n 的数组 input，它被分为用数组 parts 表示的 nParts 个分段，其中第 i 段的范围为 [parts[i], parts[i + 1])。程序会将其复制到数组 output，并为每一段分别排序
  - 在**较多的均匀分段**和**较少的随机分段**两个测例上应用 OpenMP 的 **static**、**dynamic** 和 **guided** 三种调度模式，测量其运行时间，并分析原因
  - 详细见网络学堂发布

大作业：奇偶排序

- 核心思想：由 **偶数阶段** 和 **奇数阶段** 组成，交替进行两个阶段直至每个阶段都没有元素交换

1. 【偶数阶段】 满足 (偶数，奇数) 索引的相邻元素组成元素对。

```
Index   0   1   2   3   4   5   6   7
Value  (6   1)(4   8)(2   5)(9   3)
```

2. 【偶数阶段】 如果元素对中存在顺序错误，则交换两个元素。

```
Index   0   1   2   3   4   5   6   7
Value  (1   6)(4   8)(2   5)(3   9)
```

3. 【奇数阶段】 满足 (奇数，偶数) 索引相邻元素被组成元素对。

```
Index   0   1   2   3   4   5   6   7
Value   1  (6   4)(8   2)(5   3)  9
```

4. 【奇数阶段】 如果元素对中存在顺序错误，则交换两个元素。

```
Index   0   1   2   3   4   5   6   7
Value   1  (4   6)(2   8)(3   5)  9
```

5. 交替运行【偶数阶段】和【奇数阶段】，直到两个阶段都 **没有元素交换**

大作业：奇偶排序

- 并行思路：**进程号** 作为索引号，**进程内** 处理一组元素，**进程间** 进行元素组的对比和交换

1. 进程内排序。

```
Process     0          1          2          3
Value    |6 8 3 5|  |9 4 1 7|  |8 2 1 7|  |8 7 3 9|  （排序前）
Value    |3 5 6 8|  |1 4 7 9|  |1 2 7 8|  |3 7 8 9|  （排序后）
```

2. 【偶数阶段】 满足 (偶数，奇数) 索引的相邻进程组成进程对。

```
Process     0          1          2          3
Value   (|3 5 6 8|  |1 4 7 9|)(|1 2 7 8|  |3 7 8 9|)
```

3. 【偶数阶段】 如果偶数进程的最大值大于奇数进程的最小值，则合并、排序两个进程的序列并重新分配至进程中（元素交换）。

```
Process     0          1          2          3
Value   (|1 3 4 5    6 7 8 9|)(|1 2 3 7    7 8 8 9|)  （合并排序）
Value   (|1 3 4 5|  |6 7 8 9|)(|1 2 3 7|  |7 8 8 9|)  （重新分配）
```

4. 【奇数阶段】 满足 (奇数，偶数) 索引的相邻进程组成进程对。

```
Process     0          1          2          3
Value    |1 3 4 5| (|6 7 8 9|  |1 2 2 7|) |7 8 8 9|
```

5. 【奇数阶段】 如果奇数进程的最大值大于偶数进程的最小值，则合并、排序两个进程的序列并重新分配至进程中（元素交换）。

```
Process     0          1          2          3
Value    |1 3 4 5| (|1 2 2 6    7 7 8 9|) |7 8 8 9|  （合并排序）
Value    |1 3 4 5| (|1 2 2 6|  |7 7 8 9|) |7 8 8 9|  （重新分配）
```

6. 交替运行【**偶数阶段**】和【**奇数阶段**】，直到两个阶段都 **没有元素交换**

大作业：奇偶排序

- 优化策略
  - 计算通信重叠：例如合并相邻进程序列的计算和下一轮迭代的通信之间存在重叠机会。可以通过点对点非阻塞通信实现
  - 进程绑定：将进程与核绑定，运行性能会更稳定（具体可阅读集群使用手册的 SLURM 使用部分）
  - **如果不确定自己的优化方法或实现是否符合规则，请与助教进行讨论**

- 评分标准
  - 正确分（80%）+ 性能分（10%，设置上限）+ 实验报告（10%）
  - 10组测试用例，每组单独给正确分和性能分
  - 资源限定：至多使用 2 机 56 核，性能测试所用进程规模由同学自己决定
  - 截止日期：三周时间、迟交有惩罚
  - 作业内容详见网页文档