

并行计算

(Parallel Computing)

共享内存编程 - OpenMP

学习内容：

- 使用 OpenMP 编写程序
- 使用 OpenMP 并行化 for 循环
- 任务并行
- 显式的线程同步
- 共享内存编程中的标准问题

3.parallel for 指令

●排序：冒泡排序（Bubble sort）

```
for (list_length = n; list_length >= 2; list_length--)  
    for (i = 0; i < list_length-1; i++)  
        if (a[i] > a[i+1]) {  
            tmp = a[i];  
            a[i] = a[i+1];  
            a[i+1] = tmp;  
        }
```



3.parallel for 指令

- 排序：奇偶转换排序（Odd-even transposition sort）

```
for (phase = 0; phase < n; phase++)  
    if (phase % 2 == 0)  
        for (i = 1; i < n; i += 2)  
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);  
    else  
        for (i = 1; i < n-1; i += 2)  
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- even phase: 每个奇数下标元素 $a[i]$ ，与其左方元素 $a[i-1]$ 比较
- odd phase: 每个奇数下标元素 $a[i]$ ，与其右方元素 $a[i+1]$ 比较
- 理论上证明， n 次循环后，将完成排序

3.parallel for 指令

- 排序：奇偶转换排序（Odd-even transposition sort）

$a = \{9, 7, 8, 6\}$

Phase	Subscript in Array			
	0	1	2	3
0	9	↔ 7	8	↔ 6
	7	9	6	8
1	7	9 ↔ 6	8	
	7	6	9	8
2	7 ↔ 6	9	↔ 8	
	6	7	8	9
3	6	7 ↔ 8	9	
	6	7	8	9

3.parallel for 指令

- 排序：奇偶转换排序（Odd-even transposition sort）

- 外层循环有循环携带的依赖（loop-carried dependences）
- 内层循环没有循环携带的依赖（loop-carried dependences）

```
for (phase = 0; phase < n; phase++)  
    if (phase % 2 == 0)  
        for (i = 1; i < n; i += 2)  
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);  
    else  
        for (i = 1; i < n-1; i += 2)  
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

3.parallel for 指令

```
1   for (phase = 0; phase < n; phase++) {
2       if (phase % 2 == 0)
3   #       pragma omp parallel for num_threads(thread_count) \
4           default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6               if (a[i-1] > a[i]) {
7                   tmp = a[i-1];
8                   a[i-1] = a[i];
9                   a[i] = tmp;
10            }
11        }
12    else
13    #       pragma omp parallel for num_threads(thread_count) \
14            default(none) shared(a, n) private(i, tmp)
15            for (i = 1; i < n-1; i += 2) {
16                if (a[i] > a[i+1]) {
17                    tmp = a[i+1];
18                    a[i+1] = a[i];
19                    a[i] = tmp;
20                }
21            }
22    }
```

3.parallel for 指令

●排序：奇偶转换排序（Odd-even transposition sort）

- 需要保证任何线程在开始 $p+1$ 阶段前，所有线程要结束 p 阶段
- 像 parallel 指令一样，parallel for 指令设置隐式的 barrier，直到所有的线程完成当前的 phase，才进入下一个 phase 处理
- fork 和 join 的开销：每次运行外层循环，都会 fork 和 join 线程
- 预先 fork 线程供循环使用

3.parallel for 指令

```
1  # pragma omp parallel num_threads(thread_count) \  
2      default(none) shared(a, n) private(i, tmp, phase)  
3      for (phase = 0; phase < n; phase++) {  
4          if (phase % 2 == 0)  
5              # pragma omp for  
6                  for (i = 1; i < n; i += 2) {  
7                      if (a[i-1] > a[i]) {  
8                          tmp = a[i-1];  
9                          a[i-1] = a[i];  
10                         a[i] = tmp;  
11                     }  
12                 }  
13             else  
14                 # pragma omp for  
15                     for (i = 1; i < n-1; i += 2) {  
16                         if (a[i] > a[i+1]) {  
17                             tmp = a[i+1];  
18                             a[i+1] = a[i];  
19                             a[i] = tmp;  
20                         }  
21                     }  
22             }
```

3.parallel for 指令

- 排序：奇偶转换排序（Odd-even transposition sort）

thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239

输入为20000元素的数组，时间单位为秒



3.parallel for 指令

●循环调度 (Scheduling loops)

- 假设我们想并行化下面的循环

```
sum = 0.0;  
for (i = 0; i <= n; i++)  
    sum += f(i);
```

- 假设 f 的调用时间与 i 的大小成正比
- 则块划分循环将使得线程 `thread_count - 1` 比 `thread 0` 做更多工作
- 更好的策略是采用周期划分 (cyclic partitioning)

3.parallel for 指令

- 循环调度 (Scheduling loops)

Thread	Iterations
0	$0, n/t, 2n/t, \dots$
1	$1, n/t + 1, 2n/t + 1, \dots$
\vdots	\vdots
$t - 1$	$t - 1, n/t + t - 1, 2n/t + t - 1, \dots$

Assignment of work using cyclic partitioning.

3.parallel for 指令

- 循环调度 (Scheduling loops)

```
double f(int i) {  
    int j, start = i*(i+1)/2, finish = start + i;  
    double return_val = 0.0;  
  
    for (j = start; j <= finish; j++) {  
        return_val += sin(j);  
    }  
    return return_val;  
} /* f */
```

$f(i)$ 调用 \sin 函数 i 次

3.parallel for 指令

●循环调度 (Scheduling loops)

➤ $n = 10,000$

- one thread
- run-time = 3.67 seconds

➤ $n = 10,000$

- two threads
- default assignment
- run-time = 2.76 seconds
- speedup = 1.33

3.parallel for 指令

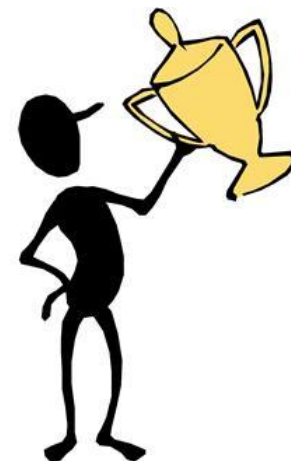
● 循环调度 (Scheduling loops)

➤ $n = 10,000$

- one thread
- run-time = 3.67 seconds

➤ $n = 10,000$

- two threads
- cyclic assignment
- run-time = 1.84 seconds
- speedup = 1.99



3.parallel for 指令

●循环调度（Scheduling loops）：schedule 从句

➤默认调度（Default schedule）

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
    for (i = 0; i <= n; i++)
        sum += f(i);
```

➤周期调度（Cyclic schedule）

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) schedule(static,1)
    for (i = 0; i <= n; i++)
        sum += f(i);
```


3.parallel for 指令

● 循环调度（Scheduling loops）： `schedule (type , chunksize)`

➤ type

- static：迭代可以在循环执行之前分配给线程
- dynamic or guided：循环执行时，迭代被分配给线程，在线程完成当前的一组迭代之后，它可以从运行时系统请求更多工作
- auto：编译器或运行时系统决定调度
- runtime：调度在运行时确定

➤ chunksize：正整数

3.parallel for 指令

- 循环调度 (Scheduling loops) : `schedule (type , chunksize)`

➤ 12次迭代, 3个线程

```
schedule(static, 1)
```

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

3.parallel for 指令

- 循环调度 (Scheduling loops) : `schedule (type , chunksize)`

➤ 12次迭代, 3个线程

```
schedule (static, 2)
```

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11

3.parallel for 指令

- 循环调度 (Scheduling loops) : `schedule (type , chunksize)`
 - 12次迭代, 3个线程

```
schedule (static, 4)
```

Thread 0 : 0, 1, 2, 3

Thread 1 : 4, 5, 6, 7

Thread 2 : 8, 9, 10, 11

3.parallel for 指令

●循环调度（Scheduling loops）： `schedule (type , chunksize)`

➤ dynamic

- 迭代被分解成 chunksize 大小的连续迭代的块
- 每个线程执行一个块，当一个线程完成一个块时，它从运行时系统请求另一个块，一直持续到所有迭代完成
- chunksize 可以省略。如果省略，则 `chunksize = 1`

3.parallel for 指令

●循环调度（Scheduling loops）： `schedule (type , chunksize)`

➤ guide

- 每个线程也执行一个块，当一个线程完成一个块时，它会请求另一个块
- 然而，在 guide schedule 中，随着块的完成，新块的大小会减小
- 如果未指定 chunksize，则块的大小将减小到1
- 如果指定了 chunksize，它将减小到 chunksize，但最后一个块可以小于 chunksize

3.parallel for 指令

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

3.parallel for 指令

- 循环调度（Scheduling loops）： `schedule (type , chunksize)`

- runtime

- 系统使用环境变量 `OMP_SCHEDULE` 来决定如何调度
- `OMP_SCHEDULE` 的值可以为 `static`, `dynamic` 或者 `guide`

3.parallel for 指令

●Which schedule?

- schedule 从句会有一定的开销：static < dynamic < guided
- 如果循环的每一次迭代都需要大致相同的计算量，那么默认分布可能会提供最佳性能
- 如果迭代的成本随着循环的执行而线性地减少（或增加），那么具有较小块大小的静态调度可能会提供最佳性能
- 如果每次迭代的成本不能预先确定，那么探索各种调度选项可能是有意义的。可以使用 schedule (runtime) 子句，通过运行不同环境变量 OMP_schedule 下的程序进行探索

4.生产者和消费者



4.生产者和消费者

●队列（Queues）

- 可看做对“顾客在超市排队买单”的一种抽象
- 多线程应用中常用的一种数据结构
- 例如：假设我们有几个“生产者”线程和几个“消费者”线程
 - 生产者线程可能“产生”对数据的请求（如：股票价格）
 - 消费者线程可能通过查找被请求的数据（股票价格）来“消费”请求
 - 生产者线程“入列”（enqueue）请求，消费者线程“出列”（dequeue）请求

4.生产者和消费者

●消息传递（Message-Passing）

- 队列的另一个应用是共享内存系统中的消息传递
- 每个线程都可以有一个共享消息队列，当一个线程想要向另一个线程“发送消息”时，它可以将消息入列（enqueue）到目标线程的队列中
- 线程可以通过将其消息队列头部的消息出列（dequeue）来接收消息

4.生产者和消费者

●消息传递（Message-Passing）

- 一个消息传递程序：每个线程随机生成整数条消息和随机的消息目的地。创建消息后，线程将消息入列（enqueue）到对应的线程消息队列中。发送一条消息后，线程检查自己的队列是否收到消息

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
  
while (!Done())  
    Try_receive();
```

4.生产者和消费者

●发送消息

- 访问消息队列，将消息入列，可能为临界区域（critical section）

```
mesg = random();  
dest = random() % thread_count;  
# pragma omp critical  
  Enqueue(queue, dest, my_rank, mesg);
```

4.生产者和消费者

●接收消息

- 只有队列的所有者会从消息队列中出列（dequeue）消息
- 如果队列中至少有两条消息，且一次从队列中出列一条消息，则 Dequeue 操作不会与 Enqueue 操作冲突
- 如果跟踪队列大小，可以避免同步操作

4.生产者和消费者

●接收消息

```
    if (queue_size == 0) return;  
    else if (queue_size == 1)  
#        pragma omp critical  
        Dequeue(queue, &src, &mesg);  
    else  
        Dequeue(queue, &src, &mesg);  
    Print_message(src, mesg);
```


4.生产者和消费者

- 终止检测：Done()

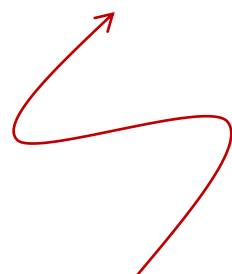
```
queue_size = enqueued - dequeued;  
if (queue_size == 0)  
    return TRUE;  
else  
    return FALSE;
```

?

4.生产者和消费者

●终止检测：Done()

```
queue_size = enqueued - dequeued;  
if (queue_size == 0 && done_sending == thread_count)  
    return TRUE;  
else  
    return FALSE;
```



each thread increments this after completing its for loop

4.生产者和消费者

●启动（Startup）

- 主线程从命令行获取参数并分配消息队列数组，每个线程一个
- 该数组被所有线程所共享
- 消息队列存储以下信息：
 - 消息列表
 - 队列尾部指针
 - 队列头部指针
 - 入列消息计数
 - 出列消息计数

4.生产者和消费者

●启动（Startup）

- 为减少传递参数的拷贝，数组中可以存放结构指针
- 由每个线程为各自的队列分配内存
- 有的线程可能会先于其他线程结束分配，并开始发送消息

```
# pragma omp barrier
```

4.生产者和消费者

●atomic 指令

- 与 critical 指令不同，它只能保护由单个 C 赋值语句组成的临界区域

```
# pragma omp atomic
```

- 语句的形式：

```
x <op>= <expression>;  
x++;  
++x;  
x--;  
--x;
```

4.生产者和消费者

●atomic 指令

- <op> 可以为以下二元运算符之一

$+$, $*$, $-$, $/$, $\&$, $^$, $|$, \ll , or \gg

- <expression> 不能引用 x
- 许多处理器提供特殊的加载修改存储（load-modify-store）指令
- 使用 atomic 指令可以更有效地保护仅执行加载修改存储的临界区域

4.生产者和消费者

●临界区域（Critical Sections）

- OpenMP 为 critical 指令提供了 name 选项

```
# pragma omp critical(name)
```

- 此时，两个用不同 name 的临界区域指令保护的代码块可以同时执行
- 但 name 是在编译时设置的，我们希望为每个线程的队列设置不同的临界区，需要在运行时设置 name，因此 critical(name) 指令无法满足要求



4.生产者和消费者

●锁（Lock）

- 锁包含数据结构和函数，允许程序在临界区域显式强制互斥

```
/* Executed by one thread */
```

```
Initialize the lock data structure;
```

```
. . .
```

```
/* Executed by multiple threads */
```

```
Attempt to lock or set the lock data structure;
```

```
Critical section;
```

```
Unlock or unset the lock data structure;
```

```
. . .
```

```
/* Executed by one thread */
```

```
Destroy the lock data structure;
```


4.生产者和消费者

●锁 (Lock)

```
# pragma omp critical
    /* q_p = msg_queues[dest] */
    Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

4.生产者和消费者

●锁 (Lock)

```
# pragma omp critical
/* q_p = msg_queues[my_rank] */
Dequeue(q_p, &src, &msg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &msg);
omp_unset_lock(&q_p->lock);
```

4.生产者和消费者

●锁（Lock）

- 当线程尝试发送或接收消息时，它只能被试图访问同一消息队列的线程阻塞，因为不同的消息队列具有不同的锁
- 而最初的实现中，一次只能有一个线程发送消息，而不管目的地是哪

4.生产者和消费者

●critical 指令，atomic 指令 or lock？

- atomic 指令有可能是获得互斥的最快方法
- 但是，OpenMP规范允许 atomic 指令在程序中的所有 atomic 指令之间强制互斥（这也是未命名的 critical 指令的行为方式）

```
# pragma omp atomic  
x++;
```

```
# pragma omp atomic  
y++;
```

- 数据结构需要互斥，而不是代码块需要互斥时，可采用 lock

4.生产者和消费者

●注意事项

- 不应对一个临界区域采用不同类型的互斥

```
# pragma omp atomic
x += f(y);
```

```
# pragma omp critical
x = g(x);
```

- 互斥结构没有公平性的保证。这意味着线程在等待访问临界区域时可能会被永远阻塞

```
while(1) {
    . . .
    # pragma omp critical
    x = g(my_rank);
    . . .
}
```

4.生产者和消费者

●注意事项

- 嵌套互斥结构可能是危险的

```
# pragma omp critical
y = f(x);
. . .
double f(double x) {
#   pragma omp critical
    z = g(x);  /* z is shared */
    . . .
}
```

4.生产者和消费者

●注意事项

- 死锁（deadlock）。当一个线程试图进入第二个临界区域时，它将永远阻塞。如果线程 u 正在第一个关键区域中执行代码，则没有线程可以在第二个块中执行代码。特别是，线程 u 无法执行此代码

```
# pragma omp critical(one)
y = f(x);
. . .
double f(double x) {
#     pragma omp critical(two)
    z = g(x);  /* z is global */
    . . .
}
```

4.生产者和消费者

●注意事项

Time	Thread u	Thread v
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

- 仅仅为关键区域使用不同的名称是不够的，程序员必须确保不同的关键区域总是以相同的顺序进入

小结

- OpenMP 是共享内存系统编程的标准
- OpenMP 使用特殊的函数和预处理器指令（pragma）
- 因此与 Pthreads 和 MPI 不同，OpenMP 需要编译器的支持
- OpenMP 最重要的特点是允许开发者在已有串程序基础上，增量的并行化，而不是从头开始写并程序
- OpenMP 启动多个线程，而不是多个进程
- OpenMP 指令可以通过从句（clause）来修改

小结

- 共享内存程序开发中的一个主要问题是 race condition
- OpenMP 提供了几种互斥机制
 - Critical 指令保证同一时间只有一个线程执行临界区域代码
 - Named Critical 指令允许不同的临界区域可以被同时执行
 - Atomic 指令设计用来利用特殊的硬件指令
 - Simple lock

```
omp_set_lock(&lock);  
critical section  
omp_unset_lock(&lock);
```

小结

- 默认情况下，大多数系统采用块划分（block partitioning）的机制来并行化 for 循环
- OpenMP 提供了可变的调度机制
 - static, dynamic, guided, auto, runtime
- OpenMP 中的变量作用域为哪些线程可以访问该变量，OpenMP 指令前声明的变量为共享变量，for 和 parallel for 循环中的循环变量除外；OpenMP 指令内部的变量为私有变量

小结

- reduction 变量同时具有私有变量和共享变量的特性

```
int sum = 0;  
for (i = 0; i < n; i++)  
    sum += A[i];
```

作业4

●Count sort:

- 基本思想是对于列表 a 中的每个元素 $a[i]$ ，计算小于 $a[i]$ 的元素个数，将 $a[i]$ 插入到由 $count$ 决定的列表下标位置中，算法结束后，用临时列表覆盖原始列表

```
void Count_sort(int a[], int n) {  
    int i, j, count;  
    int* temp = malloc(n*sizeof(int));  
  
    for (i = 0; i < n; i++) {  
        count = 0;  
        for (j = 0; j < n; j++)  
            if (a[j] < a[i])  
                count++;  
            else if (a[j] == a[i] && j < i)  
                count++;  
        temp[count] = a[i];  
    }  
  
    memcpy(a, temp, n*sizeof(int));  
    free(temp);  
} /* Count_sort */
```

作业4

●Count sort:

➤ 问题:

- 如果我们试图并行化外层循环，哪些变量为 private，哪些变量为 shared？
- 是否存在循环携带的数据依赖性？为什么？
- 编写并行化的 Count_sort
- 并行化的 Count_sort 与串行化的 Count_sort 相比，性能如何？
- 并行化的 Count_sort 与串行化的 qsort 库函数相比，性能如何？