

并行计算

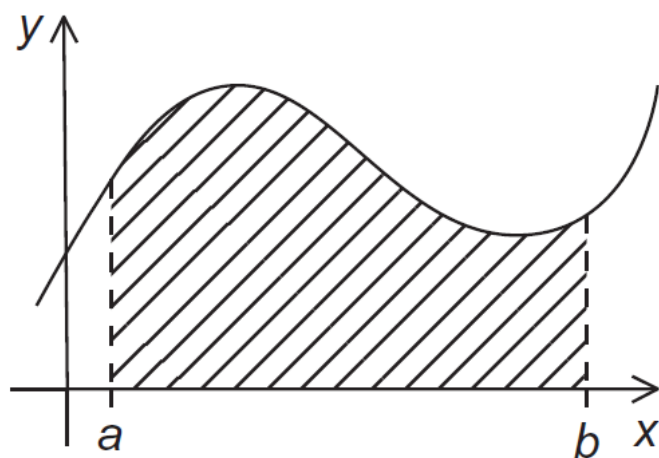
(Parallel Computing)

分布式内存编程 - MPI

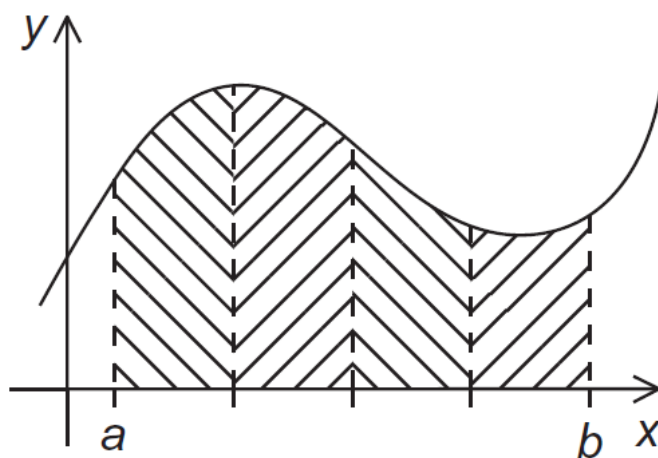
学习内容：

- 第一个 MPI 程序
- 梯形法则（Trapezoidal Rule）求面积
- 集体通信（Collective communication）

梯形法则 (Trapezoidal Rule) 数值积分



(a)



(b)

梯形法则 (Trapezoidal Rule) 数值积分

● Version 1

- 1. 每个进程计算各自的积分区域
- 2. 每个进程利用梯形法则计算该区域内的面积
- 3a. rank 不为 0 的进程将结果发送给进程 0
- 3b. 进程 0 累加接收到到的结果并打印
- $f(x)$, a , b , and n 在程序中赋值

梯形法则 (Trapezoidal Rule) 数值积分

- Version 2 – Get_input

- a, b, n 由进程 0 从输入设备获得 (scanf)
- 进程 0 将 a, b, n 发送给其他进程
- 其他进程接收 a, b, n

梯形法则（Trapezoidal Rule）数值积分

- Version 3 – MPI_Reduce

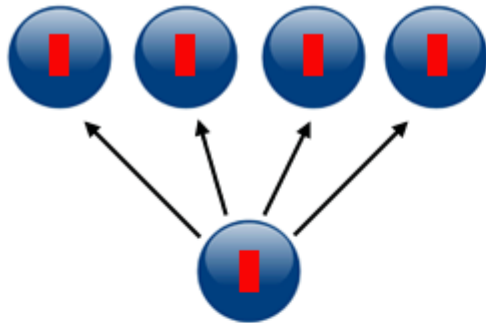
- 将 version 1 中的 3a 和 3b 中的发送、接收、累加操作替换为 MPI_Reduce

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

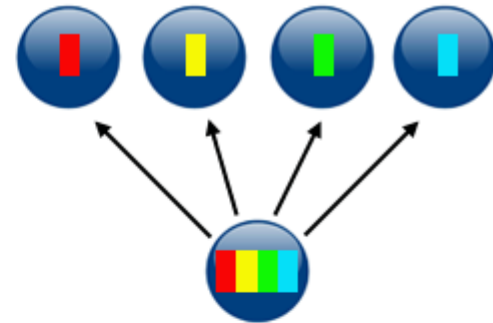
- 将 version 2 Get_input 中的发送、接收替换为广播操作

```
MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

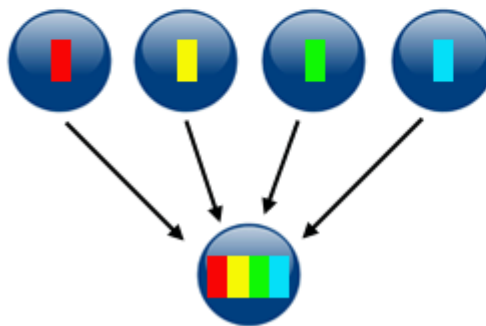
集体通信 (Collective communication)



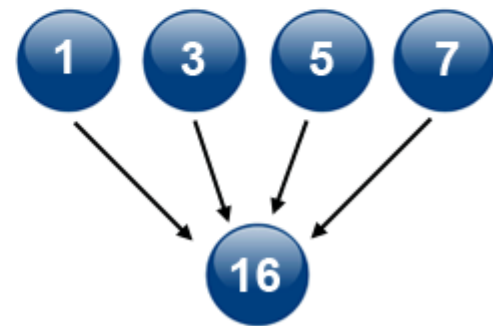
broadcast



scatter



gather



reduction

分布式内存编程 - MPI

学习内容：

- MPI 中的派生数据类型（derived datatypes）
- MPI 程序的性能评价
- 并行排序
- MPI 程序的安全性

5.MPI派生数据类型（Derived Datatype）

- 在分布式内存系统中，通信的代价要比局部计算的代价高
- 发送固定长度多条消息的成本要高于同样长度的单条消息

```
double x[1000];  
...  
if (my rank == 0)  
    for (i = 0; i < 1000; i++)  
        MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);  
else / my rank == 1 /  
    for (i = 0; i < 1000; i++)  
        MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

5.MPI派生数据类型（Derived Datatype）

- 在分布式内存系统中，通信的代价要比局部计算的代价高
- 发送固定长度多条消息的成本要高于同样长度的单条消息

```
double x[1000];  
...  
  
if (my rank == 0)  
    MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);  
else / my rank == 1 /  
    MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

5.MPI派生数据类型（Derived Datatype）

- 通过在内存中存储数据的类型和它们的相对位置来表示内存中的任何数据项集合
- 该思路是：如果发送数据的函数知道有关数据项集合的信息，则可以在发送之前从内存中收集这些项
- 类似地，接收数据的函数可以在接收到的内存中将这些项分配到正确的目的地

5.MPI派生数据类型（Derived Datatype）

- 由一系列基本MPI数据类型和每个数据类型的位移组成
- 梯形法则求面积的例子中：

Variable	Address
a	24
b	40
n	48

$\{(\text{MPI_DOUBLE}, 0), (\text{MPI_DOUBLE}, 16), (\text{MPI_INT}, 24)\}$

5.MPI派生数据类型（Derived Datatype）

- MPI_Type_create_struct：生成派生的数据类型，该数据类型由具有不同基本类型的单个元素组成

```
int MPI_Type_create_struct(  
    int          count          /* in    */,  
    int          array_of_blocklengths[] /* in    */,  
    MPI_Aint     array_of_displacements[] /* in    */,  
    MPI_Datatype array_of_types[]  /* in    */,  
    MPI_Datatype* new_type_p      /* out   */);
```

5.MPI派生数据类型（Derived Datatype）

- MPI_Type_create_struct：生成派生的数据类型，该数据类型由具有不同基本类型的单个元素组成

```
array of blocklengths[3] = {1, 1, 1};
```

```
array of displacements[] = {0, 16, 24};
```

```
array of types[3] = {MPI DOUBLE, MPI DOUBLE, MPI INT};
```

5.MPI派生数据类型（Derived Datatype）

- MPI_Get_address: 返回 location_p 引用的内存地址

```
int MPI_Get_address(  
    void*      location_p  /* in */,  
    MPI_Aint*  address_p   /* out */);
```

5.MPI派生数据类型（Derived Datatype）

- MPI_Get_address: 返回 location_p 引用的内存地址

```
MPI_Aint a_addr, b_addr, n_addr;  
  
MPI_Get_address(&a, &a_addr);  
array_of_displacements[0] = 0;  
MPI_Get_address(&b, &b_addr);  
array_of_displacements[1] = b_addr - a_addr;  
MPI_Get_address(&n, &n_addr);  
array_of_displacements[2] = n_addr - a_addr;
```


5.MPI派生数据类型（Derived Datatype）

- MPI_Type_commit：在使用派生数据类型之前调用。它允许 MPI 在通信函数中对所使用的数据类型的内部表示进行优化

```
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

- 原先的 MPI_Bcast 变为：

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

- MPI_Type_free：当使用完新的数据类型，用来释放为其分配的额外存储

```
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```

5.MPI派生数据类型（Derived Datatype）

```
void Build_mpi_type(  
    double*      a_p          /* in */,  
    double*      b_p          /* in */,  
    int*         n_p          /* in */,  
    MPI_Datatype* input_mpi_t_p /* out */) {  
  
    int array_of_blocklengths[3] = {1, 1, 1};  
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};  
    MPI_Aint a_addr, b_addr, n_addr;  
    MPI_Aint array_of_displacements[3] = {0};  
  
    MPI_Get_address(a_p, &a_addr);  
    MPI_Get_address(b_p, &b_addr);  
    MPI_Get_address(n_p, &n_addr);  
    array_of_displacements[1] = b_addr - a_addr;  
    array_of_displacements[2] = n_addr - a_addr;  
    MPI_Type_create_struct(3, array_of_blocklengths,  
                          array_of_displacements, array_of_types,  
                          input_mpi_t_p);  
    MPI_Type_commit(input_mpi_t_p);  
} /* Build_mpi_type */
```

5.MPI派生数据类型（Derived Datatype）

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {
    MPI_Datatype input_mpi_t;

    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```

梯形法则（Trapezoidal Rule）数值积分

- Version 4 – Build_mpi_type

- 将 Get_input 中的多次广播改为广播派生数据类型

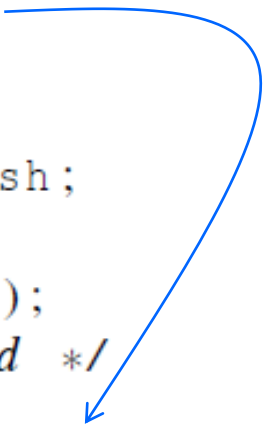
```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

6.性能评估 (Performance evaluation)

●计时：并行计时

- MPI_Wtime：返回逝去的秒数

```
double MPI_Wtime(void);  
  
double start, finish;  
...  
start = MPI_Wtime();  
/* Code to be timed */  
...  
finish = MPI_Wtime();  
printf("Proc %d > Elapsed time = %e seconds\n"  
       my_rank, finish-start);
```



6.性能评估 (Performance evaluation)

●计时：串行计时

➤gettimeofday：返回逝去的毫秒数

➤time.h 中的宏：GET_TIME，返回逝去的秒数

```
#include "timer.h"  
.  
.  
.  
double now;  
.  
.  
.  
GET_TIME(now);
```



6.性能评估 (Performance evaluation)

●计时：串行计时

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

6.性能评估 (Performance evaluation)

- 计时：并行计时（返回一个时间，同步开始）

- MPI_Barrier：确保调用过程中没有进程返回，直到通信器中的每个进程都开始调用它

```
int MPI_Barrier(MPI_Comm comm /* in */);
```



6.性能评估 (Performance evaluation)

- 计时：并行计时（返回一个时间，同步开始）

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
           MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
```

6.性能评估（Performance evaluation）

●Matrix-Vector 相乘的并行和串行结果比较

(milliseconds)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}$$

6.性能评估 (Performance evaluation)

- 加速和效率 (Speedup and efficiency)

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

6.性能评估（Performance evaluation）

●加速和效率（Speedup and efficiency）

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Speedup

6.性能评估（Performance evaluation）

- 加速和效率（Speedup and efficiency）

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}$$

6.性能评估（Performance evaluation）

●加速和效率（Speedup and efficiency）

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

efficiency

6.性能评估 (Performance evaluation)

●可扩展性 (Scalability)

- 如果问题的大小可以以一定的速度增加，使得效率不会随着进程数量的增加而降低，那么程序是可伸缩的
- 强可扩展：保持恒定效率，与问题大小无关
 - Program A 的效率为 0.75 ($p \geq 2$)，与问题大小无关
- 弱可扩展：问题的大小以与进程的数量相同的速率增加，则可以保持恒定的效率
 - Program B 的效率为 $n / (625p)$ ($p \geq 2, 1000 \leq n \leq 625p$)



7. 一个并行的排序算法

- n 个关键字 和 $p = \text{comm_sz}$ 个进程
- n/p 个关键字分配给每个进程
- 对哪个关键字分配给哪个进程没有限制，但是
- 当算法结束：
 - 每个进程中的关键字已经排序（如：升序排列）
 - 如果 $0 \leq q < r < p$, 则 q 进程中的每个关键字 \leq r 进程中的每个关键字

7. 一个并行的排序算法

● 串行的冒泡排序 (Bubble Sort)



```
void Bubble_sort(  
    int  a[]  /* in/out */,  
    int  n    /* in      */) {  
    int  list_length, i, temp;  
  
    for (list_length = n; list_length >= 2; list_length--)  
        for (i = 0; i < list_length-1; i++)  
            if (a[i] > a[i+1]) {  
                temp = a[i];  
                a[i] = a[i+1];  
                a[i+1] = temp;  
            }  
}  
/* Bubble_sort */
```

7. 一个并行的排序算法

● 串行的冒泡排序（Bubble Sort）

- 比较交换（compare - swap）发生的顺序对算法的正确性至关重要
- $a[i - 1] = 9, a[i] = 5, a[i + 1] = 7$

7. 一个并行的排序算法

● 奇偶换位排序 (Odd-even transposition sort)

➤ 偶数阶段，比较交换对：

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$$

➤ 奇数阶段，比较交换对：

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$$

7. 一个并行的排序算法

● 奇偶换位排序 (Odd-even transposition sort)

- 5, 9, 4, 3
- 偶数阶段, 比较交换对: (5,9)和(4,3)
得到 (5,9,3,4)
- 奇数阶段, 比较交换对: (9,3)
得到 (5,3,9,4)
- 偶数阶段, 比较交换对: (5,3)和(9,4)
得到 (3,5,4,9)
- 奇数阶段, 比较交换对: (5,4)
得到 (3,4,5,9)

7. 一个并行的排序算法

● 串行的奇偶换位排序

- 定理：假设 A 为 n 个 key 的列表，将 A 作为奇偶换位排序算法的输入，则： n 阶段后， A 将被排序

```

void Odd_even_sort(
    int  a[]  /* in/out */,
    int  n    /* in    */) {
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        } else { /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
    } /* Odd_even_sort */

```

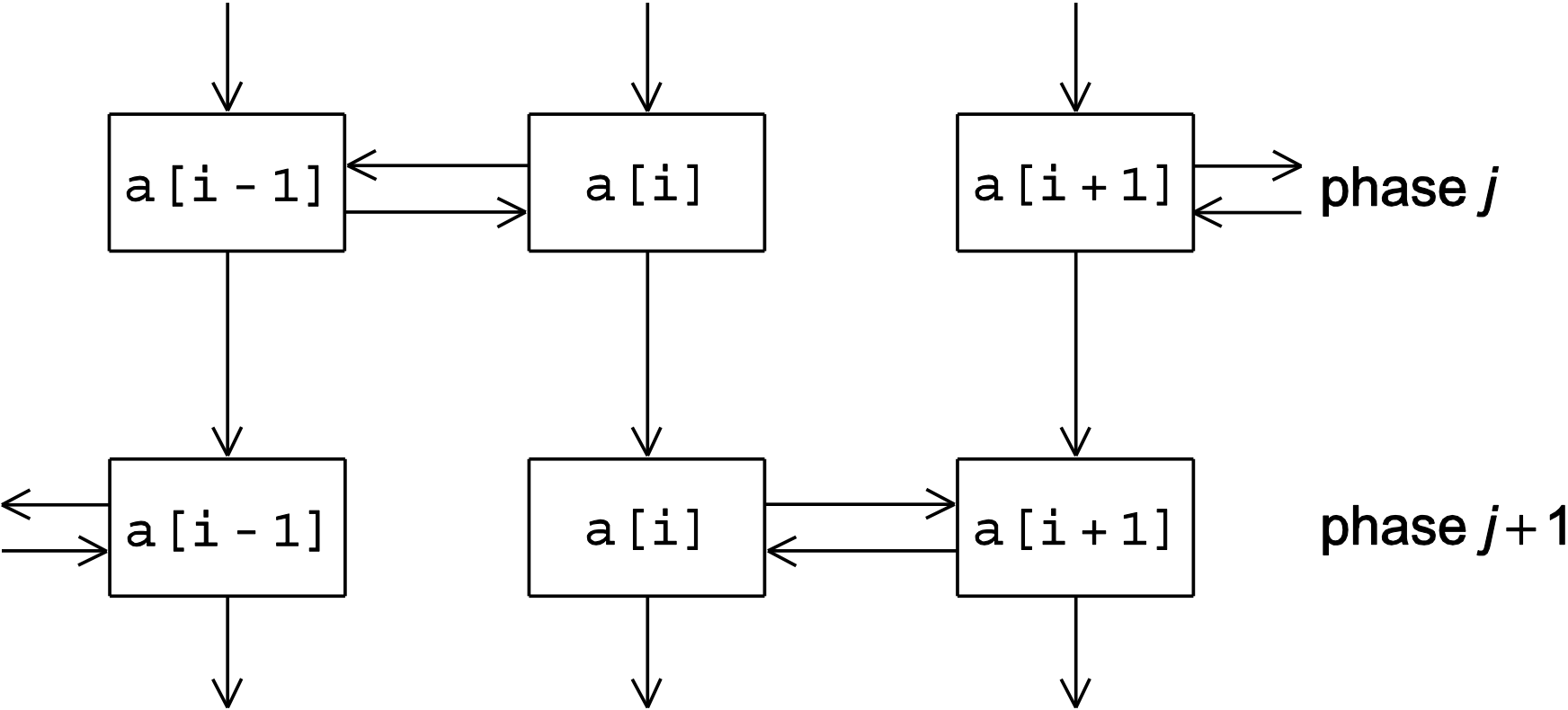
7. 一个并行的排序算法

● 并行的奇偶换位排序

- 任务：确定阶段 j 结束后 $a[i]$ 的值
- 通信：确定 $a[i]$ 值的任务需要与确定 $a[i-1]$ 或者 $a[i+1]$ 的任务通信，并且在阶段 j 结束后确定的 $a[i]$ 需要用来确定 $j+1$ 阶段的 $a[i]$ 值

7.一个并行的排序算法

- 并行的奇偶换位排序



Tasks determining $a[i]$ are labeled with $a[i]$.

7.一个并行的排序算法

●并行的奇偶换位排序

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

7. 一个并行的排序算法

● 并行的奇偶换位排序

- 定理：如果并行的奇偶换位排序在 p 个进程上运行，则 p 阶段后，将完成排序

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

7. 一个并行的排序算法

● 并行的奇偶换位排序

➤ 如何计算 partner?

```
if (phase % 2 == 0)           /* Even phase */
    if (my_rank % 2 != 0)      /* Odd rank */
        partner = my_rank - 1;
    else                       /* Even rank */
        partner = my_rank + 1;
else                           /* Odd phase */
    if (my_rank % 2 != 0)      /* Odd rank */
        partner = my_rank + 1;
    else                       /* Even rank */
        partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL;
```

7. 一个并行的排序算法

● MPI 程序中的安全性

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
          MPI_STATUS_IGNORE);
```

- MPI 标准允许 MPI_Send 以两种不同的方式工作
 - 拷贝消息到 MPI 管理的缓冲区并返回
 - 阻塞，直到匹配的 MPI_Recv 函数开始工作

7.一个并行的排序算法

●MPI 程序中的安全性

- 许多 MPI 的实现会设置一个阈值，用来在缓冲和阻塞之间切换
 - 小于阈值的消息将被 MPI_Send 缓冲
 - 大于阈值的消息将被阻塞

7.一个并行的排序算法

●MPI 程序中的安全性

- 如果每个执行 MPI_Send 的进程被阻塞，则没有进程可以执行 MPI_Recv，程序将被挂起（hang）或死锁（deadlock），也就是每个阻塞的进程在等待一个无法发生的事件
- 一个依赖于 MPI 提供的缓冲机制的程序是不安全的

7. 一个并行的排序算法

● MPI 程序中的安全性

- 如何判断一个程序是否安全？
- 如何修改并行的奇偶换位排序程序，使之安全？

7. 一个并行的排序算法

● MPI 程序中的安全性

➤ 如何判断一个程序是否安全？

- 用 MPI_Ssend 替换 MPI_Send
- “s” 代表同步（synchronous），MPI_Ssend 保证在匹配的接收调用开始前被阻塞

```
int MPI_Ssend(  
    void*      msg_buf_p      /* in */,  
    int        msg_size       /* in */,  
    MPI_Datatype msg_type      /* in */,  
    int        dest           /* in */,  
    int        tag            /* in */,  
    MPI_Comm   communicator    /* in */);
```

7. 一个并行的排序算法

● MPI 程序中的安全性

- 如何修改并行的奇偶换位排序程序，使之安全？
 - MPI_Sendrecv：在一个调用中执行阻塞发送和接收
 - dest 和 source 可以相同，也可以不同
 - 由 MPI 来调度通信，防止程序挂起

7.一个并行的排序算法

```
int MPI_Sendrecv(  
    void*          send_buf_p      /* in */,  
    int           send_buf_size   /* in */,  
    MPI_Datatype   send_buf_type  /* in */,  
    int           dest            /* in */,  
    int           send_tag        /* in */,  
    void*          recv_buf_p      /* out */,  
    int           recv_buf_size   /* in */,  
    MPI_Datatype   recv_buf_type  /* in */,  
    int           source          /* in */,  
    int           recv_tag        /* in */,  
    MPI_Comm       communicator   /* in */,  
    MPI_Status*    status_p       /* in */);
```

7.一个并行的排序算法

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
             recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
             MPI_Status_ignore);
```

7. 一个并行的排序算法

```

void Merge_low(
    int    my_keys[],      /* in/out    */
    int    recv_keys[],   /* in        */
    int    temp_keys[],   /* scratch   */
    int    local_n        /* = n/p, in */) {
    int m_i, r_i, t_i;

    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if (my_keys[m_i] <= recv_keys[r_i]) {
            temp_keys[t_i] = my_keys[m_i];
            t_i++; m_i++;
        } else {
            temp_keys[t_i] = recv_keys[r_i];
            t_i++; r_i++;
        }
    }

    for (m_i = 0; m_i < local_n; m_i++)
        my_keys[m_i] = temp_keys[m_i];
} /* Merge_low */

```

7.一个并行的排序算法

●运行时间

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

(times are in milliseconds)

小结

- MPI（Message-Passing Interface）是能够被 C, C++ 或者 Fortran 程序调用的库函数
- 通信器（communicator）是可以互相发送消息的进程的集合。MPI 程序启动后，MPI通常会创建一个包含所有进程的通信器：
MPI_COMM_WORLD
- 许多并行程序使用单程序多数据（SPMD）的方法

小结

- 用 `MPI_Send` 向进程发送消息，`MPI_Recv` 接收消息。调用 `MPI_Recv` 将被阻塞，而 `MPI_Send` 的行为由 MPI 的实现定义
- 编写 MPI 程序时要注意区分局部变量和全局变量
- 大多数串行程序是确定性的：如果我们用相同的输入运行相同的程序，我们将得到相同的输出。而并行程序通常不具有这个特性

小结

- 集体通信（Collective communications）包含通信器中的所有进程
 - MPI_Reduce 和 MPI_Allreduce
 - MPI_Bcast
 - MPI_Scatter
 - MPI_Gather 和 MPI_Allgather
 - MPI_Barrier

小结

- 给并行程序计时时，通常对运行时间或挂钟时间（wall clock time）感兴趣，挂钟时间是一个代码块所占用的总时间。它包括用户代码的时间、库函数中时间、用户代码启动操作系统函数的时间以及空闲时间
 - MPI_Wtime、GET_TIME
- 对同一程序进行计时通常会得到不同的结果，通常取最小值

小结

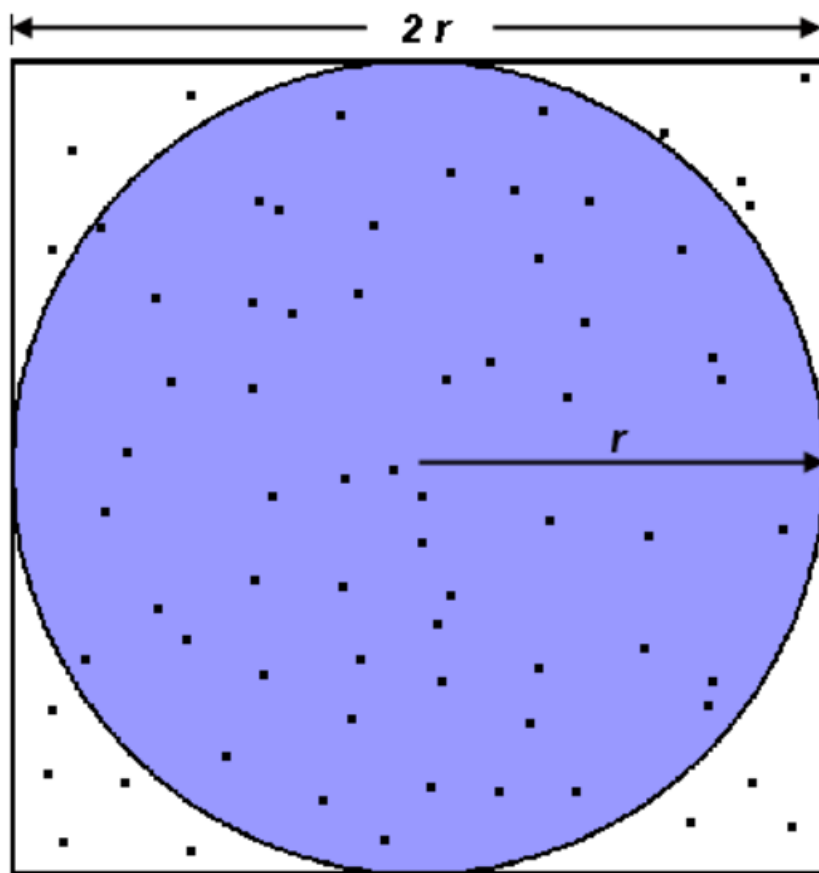
- 加速（Speedup）和效率（Efficiency）
- 如果增加问题的大小，可以使得效率就不随 p 的增加而降低，那么并行程序就是可扩展的（scalable）
- 如果一个并行程序依赖于 `MPI_Send` 的实现方式，则它是不安全的

第二次作业

- 假设我们把飞镖随机扔在一个正方形的飞镖板上，它的靶心在 origin，正方形的边长为两英尺。假设有一个圆刻在正方形的飞镖板上。圆的半径是1英尺，面积是 π 平方英尺。如果被飞镖击中的点是均匀分布的（我们总是击中正方形），那么飞镖击中的点在圆内的数量应该近似满足等式

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4}$$

第二次作业



第二次作业

- 这种方法称为蒙特卡洛法 (Monte Carlo)
- 编写一个 MPI 程序，使用蒙特卡洛法估计 π 值
 - 进程 0 读入总的投掷数，并广播到其他进程
 - 使用 MPI_Reduce 获得局部变量 number_in_circle 的总和，由进程 0 打印结果（可能需要使用 long long int）

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
    x = random double between -1 and 1;
    y = random double between -1 and 1;
    distance_squared = x*x + y*y;
    if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```