MPI

- 计算π
  - 积分法
  - 蒙特卡罗法
- 计算ln2
- 连续素数
- 最大连续素数间隔【Wrong】
- 水仙花数
- 完全数
- 作业题1: 级数
- 作业题2：数列和
- 作业题3：莱布尼茨计算π
- 作业题4：Simpson计算π
- 向量中的元素求和
- 大数组排序
- MPI通信范式
- MPI数据分割范式
  - 均匀划分
  - 不均匀划分

pthread

- 范式一：创建线程
- 范式四：线程互斥锁
- 范式五：线程条件变量
- 范式六：线程信号量
- 范式八：线程取消
- 范式十：线程池
- 快速排序
- 计算ln2
- 计算π
  - 蒙特卡洛法
  - 积分法
- 辛普生法
- 输出奇数偶数
- 素数
- 水仙花数
- 完全数
- 顺序输出123
- 生产者消费者问题
- 哲学家进餐问题
- 读写者问题

OpenMP

- 计算π的值(5种方法)
  - 积分累加法
  - 蒙特卡罗法
- 计算ln2
- 矩阵相乘
- 连续素数
- 最大连续素数间隔
- 水仙花数
- 完全数

MPI

计算π

积分法

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
int n = 10000000;

int main() {
    // 初始化
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double x, local = 0.0;
    for (int i = rank; i < n; i+=size) {
        x = (i+0.5)/(double)n;
        local += 4.0/(1+x*x);
    }
    double pi = 0.0;
    MPI_Reduce(&local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("pi: %f\n", pi);
    }
    // 结束MPI环境
    MPI_Finalize();
    return 0;
}
```

蒙特卡罗法

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>

#define MAX_NUM 1000000

int main(int argc, char** argv) {
    // 初始化MPI环境
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int num = MAX_NUM/size;
    double x,y,dist;
    int local_hint_num = 0;
    for (int i = 0; i < num; i++) {
        x = (double)rand()/(double)RAND_MAX;
        y = (double)rand()/(double)RAND_MAX;
        dist = x*x + y*y;
        if (dist <= 1) {
```

```
            local_hint_num++;
        }
    }
    int total_num = 0;
    // 将局部结果累加到全局结果中
    MPI_Reduce(&local_hint_num, &total_num, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        double pi = 4 * (double)total_num/MAX_NUM;
        printf("pi: %f\n", pi);
    }

    // 结束MPI环境
    MPI_Finalize();
    return 0;
}
```

计算ln2

- `sendbuf`：指向发送缓冲区的指针，即需要进行归约操作的数据的起始地址。
- `recvbuf`：指向接收缓冲区的指针，即归约操作的结果将要存储的位置。在归约操作结束后，只有接收缓冲区中的数据是有效的。
- `count`：表示缓冲区中元素的数量。对于基本数据类型，该值通常为1；对于复合数据类型，该值通常为元素的数量。
- `datatype`：表示缓冲区中元素的数据类型。MPI提供了多种数据类型，包括基本数据类型和自定义数据类型。
- `op`：表示归约操作的类型。MPI支持多种归约操作，包括求和、求积、求最大值、求最小值、异或等。
- `root`：表示结果将要发送到的进程号。通常情况下，只有指定的进程才需要接收归约操作的结果。
- `comm`：表示通信子，即用于进行通信的进程组。通常使用 `MPI_COMM_WORLD` 表示包含所有进程的通信子。

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>

#define MAX_NUM 1000000

int main(int argc, char** argv) {
    // 初始化
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // 单个进程的数据量
    int chunk = MAX_NUM/size;
    // 划分数据集
    int start = rank*chunk;
    int end = start + chunk;
    if (rank == size-1) {
        end = MAX_NUM;
```

```
    }
    double factor;
    if (start % 2 == 0) {
        factor = 1.0;
    } else {
        factor = -1.0;
    }
    double my_ln2 = 0.0;
    for (int i = start; i < end; ++i) {
        my_ln2 += factor/(i + 1);
        factor = -factor;
    }
    double ans = 0.0;
    MPI_Reduce(&my_ln2, &ans, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("ans: %f\n", ans);
    }

    MPI_Finalize();
    return 0;
}
```

1. `MPI_CHAR`：字符类型。
2. `MPI_SHORT`：短整型。
3. `MPI_INT`：整型。
4. `MPI_LONG`：长整型。
5. `MPI_FLOAT`：单精度浮点型。
6. `MPI_DOUBLE`：双精度浮点型。
7. `MPI_LONG_DOUBLE`：长双精度浮点型。
8. `MPI_BYTE`：字节类型。
9. `MPI_PACKED`：打包数据类型。

连续素数

只考虑两个连续奇数且素数

```
bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
int n = 1000000;
int main() {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int count = 0;
```

```
    // 假设每个进程都能分到相等的数据量，则区间边界是偶数且不需要考虑最后一个进程数据不一致的情况
    int chunk_size = n / size;
    int start = rank * chunk_size;
    int end = (rank + 1) * chunk_size;
    for (int i = start+1; i <= end-1; i += 2) {
        if (is_prime(i) && is_prime(i + 2)) {
            count++;
        }
    }
    int sum = 0;
    MPI_Reduce(&count, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    printf("在小于 %d 的整数范围内，连续奇数都是素数的情况的次数为: %d\n", n, sum);
    return 0;
}
```

考虑任意长度连续奇数且素数

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>

#define MAX_NUM 1000000

bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}


int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int chunk_size = MAX_NUM / size;
    int start = rank * chunk_size;
    int end = (rank + 1) * chunk_size;

    int count = 0;
    int num_primes_in_a_row = 0;

    for (int i = start + 1; i <= end - 1; i += 2) {
        if (is_prime(i)) {
            num_primes_in_a_row++;
        } else {
             if (num_primes_in_a_row >= 2) {
                count++;
            }
            num_primes_in_a_row = 0;
        }
    }
```

```
    int total_count;
    MPI_Reduce(&count, &total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Total count: %d\n", total_count);
    }

    MPI_Finalize();
    return 0;
}
```

最大连续素数间隔【Wrong】

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>

#define MAX_NUM 1000000

bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int chunk_size = MAX_NUM / size;
    int start = rank * chunk_size;
    int end = (rank + 1) * chunk_size;

    int max_gap = 0;
    int last_prime = 3;
    for (int i = start + 1; i <= end - 1; i+=2) {
        if (is_prime(i)) {
            if (i - last_prime > max_gap) {
                max_gap = i - last_prime;
            }
            last_prime = i;
        }
    }

    int global_max_gap;
    MPI_Reduce(&max_gap, &global_max_gap, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Global max gap: %d\n", global_max_gap);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

水仙花数

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += pow(digit, digits);
        temp /= 10;
    }
    return sum == n;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int total_narcissistic = 0;
    for (int n = 3; n <= 24; n++) {
        int start = pow(10, n - 1) + rank * ((pow(10, n) - 1) / size);
        int end = pow(10, n - 1) + (rank + 1) * ((pow(10, n) - 1) / size) - 1;
        if (rank == size - 1) {
            end = pow(10, n) - 1;
        }
        int count = 0;
        for (int i = start; i <= end; i++) {
            if (is_narcissistic(i)) {
                count++;
            }
        }
        MPI_Reduce(&count, &total_narcissistic, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

        if (rank == 0) {
```

```
            cout << "在 " << n << " 位数范围内，水仙花数的总数为：" <<
total_narcissistic << endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```

不需要求连续序列，可以离散化分片：

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += pow(digit, digits);
        temp /= 10;
    }
    return sum == n;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int total_narcissistic = 0;
    for (int n = 3; n <= 5; n++) {
        int count = 0;
        for (int i = pow(10, n - 1) + rank; i < pow(10, n); i+=size) {
            if (is_narcissistic(i)) {
                count++;
            }
        }
        MPI_Reduce(&count, &total_narcissistic, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

        if (rank == 0) {
            cout << "在 " << n << " 位数范围内，水仙花数的总数为：" <<
total_narcissistic << endl;
        }
    }
```

```
    MPI_Finalize();
    return 0;
}
```

【直接把数据展开均匀分割】

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += pow(digit, digits);
        temp /= 10;
    }
    return sum == n;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int min = pow(10, 2);
    int max = pow(10, 25);

    int count = 0;
    for (int i = min + rank; i <= max; i += size) {
        if (is_narcissistic(i)) {
            count++;
        }
    }

    int total_narcissistic = 0;
    MPI_Reduce(&count, &total_narcissistic, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        cout << "在 " << max << " 位数范围内，水仙花数的总数为：" <<
total_narcissistic << endl;
    }

    MPI_Finalize();
```

```
    return 0;
}
```

完全数

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int is_perfect_number(int n) {
    int sum = 0;
    for (int i = 1; i < n; i++) {
        if (n % i == 0) {
            sum += i;
        }
    }
    return sum == n;
}

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n_min = 6;
    int n_max = 33550337;

    // 平均分配n的范围
    int n_start = n_min + rank * ((n_max - n_min) / size);
    int n_end = n_min + (rank + 1) * ((n_max - n_min) / size);
    if (rank == size - 1) {
        n_end = n_max;
    }

    // 计算每个进程分配的范围内的完全数并输出
    int count = 0;
    for (int n = n_start; n <= n_end; n++) {
        if (is_perfect_number(n)) {
            printf("%d ", n);
            count++;
            if (count == 8) {
                break;
            }
        }
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

不需要求连续序列，可以离散化分片：

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int is_perfect_number(int n) {
```

```
    int sum = 0;
    for (int i = 1; i < n; i++) {
        if (n % i == 0) {
            sum += i;
        }
    }
    return sum == n;
}

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n_max = 33550337;

    // 计算每个进程分配的范围内的完全数并输出
    int count = 0;
    for (int n = rank; n < n_max; n+=size) {
        if (is_perfect_number(n)) {
            printf("%d ", n);
            count++;
            if (count == 8) {
                break;
            }
        }
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

作业题1: 级数

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int n_max = 1000000;

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double count = 0;
    for (int n = rank; n < n_max; n+=size) {
        count += 1.0/(n*(n+1));
    }
    double total_count = 0;
    MPI_Reduce(&count, &total_count, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

作业题2：数列和

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int n_max = 625;

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double count = 0;
    for (int n = rank; n < n_max; n+=size) {
        count += 1.0/(sqrt(n)+sqrt(n+1));
    }
    double total_count = 0;
    MPI_Reduce(&count, &total_count, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

作业题3：莱布尼茨计算π

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int n_max = 625;

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double count = 0;
    for (int n = rank; n < n_max; n+=size) {
        if (n % 2 == 0) {
            count += 1.0/(2*n+1);
        } else {
            count -= 1.0/(2*n+1);
        }
    }
    double total_count = 0;
    MPI_Reduce(&count, &total_count, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("π = %f\n", 4*total_count);
    }
    MPI_Finalize();
    return 0;
}
```

作业题4：Simpson计算π

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
```

```
double f(double x) {
    return 4.0 / (1.0 + x * x);
}

int main(int argc, char** argv) {
    int num_processes, process_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    double a = 0.0, b = 1.0;  // 积分区间
    int n = 1000000;  // 子区间数
    double h = (b - a) / (n * num_processes);  // 步长

    // 使用辛普生法计算积分
    double sum = 0.0;
    for (int i = process_rank * n; i < (process_rank + 1) * n; i++) {
        double x = a + (i + 0.5) * h;
        sum += f(x) + 4.0 * f(x + h / 2.0) + f(x + h);
    }
    sum *= h / 6.0;

    // 在所有进程中汇总结果
    double total_sum;
    MPI_Reduce(&sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (process_rank == 0) {
        printf("pi = %.16f\n", total_sum);
    }

    MPI_Finalize();
    return 0;
}
```

向量中的元素求和

SPMD（Single Program Multiple Data）编程例子：

下面是一个简单的SPMD程序，将一个向量中的元素求和，使用MPI_Reduce函数实现归约操作：

```
#include <mpi.h>
#include <iostream>
#include <vector>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int n = 1000000;
    std::vector<double> data(n);
    std::fill(data.begin(), data.end(), rank + 1);

    double sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += data[i];
    }
```

```
    double global_sum;
    MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        std::cout << "Sum is " << global_sum << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

MPMD（Multiple Program Multiple Data）编程例子：

下面是一个简单的MPMD程序，使用MPI_Comm_spawn函数在不同的进程中启动不同的程序：

```
#include <iostream>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        const char* command = "./worker";
        MPI_Comm intercomm;
        MPI_Comm_spawn(command, MPI_ARGV_NULL, 4, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercomm, MPI_ERRCODES_IGNORE);

        int result;
        MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, intercomm,
MPI_STATUS_IGNORE);

        std::cout << "Result from worker is " << result << std::endl;
    } else {
        int result = rank * rank;
        MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

在上面的程序中，主进程使用MPI_Comm_spawn函数启动了4个子进程，子进程运行的是另外一个程序worker，主进程通过MPI_Recv函数接收来自子进程的结果。

大数组排序

Master/Worker编程例子：

下面是一个简单的Master/Worker程序，将一个大的数组进行排序，使用MPI_Send和MPI_Recv函数实现主进程和工作进程之间的通信：

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char** argv) {
```

```
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int n = 1000000;
    std::vector<int> data(n);

    if (rank == 0) {
        // Generate data
        std::generate(data.begin(), data.end(), []() { return rand() % 10000; });

        // Send data to workers
        int chunk_size = n / (size - 1);
        for (int i = 1; i < size; i++) {
            int start = (i - 1) * chunk_size;
            int end = (i == size - 1) ? n : i * chunk_size;
            MPI_Send(&data[start], end - start, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        // Receive sorted data from workers
        for (int i = 1; i < size; i++) {
            int start = (i - 1) * chunk_size;
            int end = (i == size - 1) ? n : i * chunk_size            ;
            MPI_Recv(&data[start], end - start, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }

        // Sort the final result
        std::sort(data.begin(), data.end());

        std::cout << "Sorted data:";
        for (int i = 0; i < n; i++) {
            std::cout << " " << data[i];
        }
        std::cout << std::endl;
    } else {
        // Receive data from master
        int chunk_size = n / (size - 1);
        int start = (rank - 1) * chunk_size;
        int end = (rank == size - 1) ? n : rank * chunk_size;
        std::vector<int> chunk(end - start);
        MPI_Recv(&chunk[0], end - start, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        // Sort the chunk
        std::sort(chunk.begin(), chunk.end());

        // Send sorted chunk back to master
        MPI_Send(&chunk[0], end - start, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

在上面的程序中，主进程将一个大的数组进行划分，然后将划分后的子数组发送给工作进程，工作进程对接收到的子数组进行排序，然后将排序后的子数组发送回主进程，主进程将所有的子数组合并后进行最终的排序，输出结果。

Pipeline编程例子：

第一种：依次传递，每个模块功能相同，最后一个进程负责输出最终结果

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int n = 10;
    std::vector<int> arr(n);

    if (rank == 0) {
        // Generate data
        std::generate(arr.begin(), arr.end(), []() { return std::rand() % 100;
});
        std::cout << "Original Array: ";
        for (int i = 0; i < n; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;

        // Send data to worker 1
        MPI_Send(arr.data(), n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else {
        // Receive data from previous worker
        MPI_Recv(arr.data(), n, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        // Sort data
        std::sort(arr.begin(), arr.end());

        if (rank == size - 1) {
            // Output result
            std::cout << "Sorted Array: ";
            for (int i = 0; i < n; i++) {
                std::cout << arr[i] << " ";
            }
            std::cout << std::endl;
        } else {
            // Send data to next worker
            MPI_Send(arr.data(), n, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

第二种：依次传递，每个模块功能可以不同，最后一个进程输出结果

```
#include <mpi.h>
#include <iostream>
#include <string>
#include <algorithm>
```

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size != 3) {
        std::cerr << "This program requires exactly 3 processes" << std::endl;
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    std::string str;

    if (rank == 0) {
        // Generate data
        str = "hello, world!";

        // Send data to worker 1
        MPI_Send(str.c_str(), str.length() + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Receive data from master
        char buf[1024];
        MPI_Recv(buf, 1024, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Do something...

        // Send data to worker 2
        MPI_Send(buf, strlen(buf) + 1, MPI_CHAR, 2, 0, MPI_COMM_WORLD);
    } else if (rank == 2) {
        // Receive data from worker 1
        char buf[1024];
        MPI_Recv(buf, 1024, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Do something....

        // Output result
        std::cout << "Result: " << buf << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

MPI通信范式

MPI通信范式指的是MPI库中用于实现进程间通信的不同方式。MPI标准定义了三种基本的通信范式，分别是点对点通信、集合通信和归约通信。

1. 点对点通信：点对点通信是最基本的通信方式，它允许两个进程之间进行直接的通信。MPI库提供了一系列的点对点通信函数，包括MPI_Send、MPI_Recv、MPI_Isend、MPI_Irecv等。
2. 集合通信：集合通信是指多个进程之间进行通信的一种方式。MPI库提供了一系列的集合通信函数，包括MPI_Bcast、MPI_Scatter、MPI_Gather、MPI_Allreduce等。
3. 归约通信：归约通信是一种集合通信的特例，它用于将多个进程的数据合并成一个结果。MPI库提供了一系列的归约通信函数，包括MPI_Reduce、MPI_Allreduce、MPI_Scan等。

MPI_Send和MPI_Recv函数是MPI中最常用的点对点通信函数，用于在两个进程之间发送和接收消息。

其参数的特点：

- 指定数据+数据长度+数据类型
- 指定发送方、接收方的进程编号（rank）
- 指定消息的tag，相当于消息类型，目的是可以收到并区分同一个进程发来的不同tag的消息
- 指定通信组MPI_Comm

MPI_Send函数的语法如下：

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

参数含义如下：

- buf：指向要发送的数据的指针，类型为void*。
- count：要发送的数据的数量。
- datatype：要发送的数据的类型，如MPI_INT、MPI_DOUBLE等。
- **dest：指定接收消息的进程的rank号。**
- tag：用于标识消息的一个整型值，**接收进程可以根据tag的值来区分不同的消息。**
- comm：通信子，指定通信使用的进程组。

MPI_Recv函数的语法如下：

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

参数含义如下：

- buf：指向接收数据的缓冲区的指针，类型为void*。
- count：要接收的数据的数量。
- datatype：要接收的数据的类型，如MPI_INT、MPI_DOUBLE等。
- **source：指定发送消息的进程的rank号，如果source为MPI_ANY_SOURCE，则可以接收任意进程发送的消息。**
- **tag：与发送消息时指定的tag一致，用于标识接收到的消息。**
- comm：通信子，指定通信使用的进程组。
- **status：用于返回接收到的消息的状态，包括发送进程的rank号、消息的tag值、接收到的数据的数量等信息。**

MPI_Send和MPI_Recv函数的特点如下：

1. **阻塞式**函数：MPI_Send和MPI_Recv函数都是阻塞式函数，也就是说发送和接收操作会一直等待，直到消息传输完成才会继续执行后面的代码。如果发送和接收的进程不匹配，或者发送和接收的数据类型不匹配，就会导致程序出现错误或死锁。
2. **缓冲区**管理：MPI_Send和MPI_Recv函数需要提供缓冲区来存放发送和接收的数据。在MPI_Send函数中，发送进程将数据复制到缓冲区中，然后发送缓冲区中的数据。在MPI_Recv函数中，接收进程需要提供缓冲区来存放接收到的数据。因此，程序必须确保缓冲区的大小足够存放发送和接收的数据，否则会导致程序出现错误或崩溃。

注意事项如下：

1. 发送和接收的进程必须使用相同的通信子MPI_Comm，否则两个进程之间无法进行通信。
2. 发送和接收的**数据类型必须匹配**，否则会导致程序出现错误或崩溃。
3. 发送和接收的**数据量必须匹配**，否则会导致程序出现错误或死锁。
4. 发送和接收的进程必须按照相同的顺序调用MPI_Send和MPI_Recv函数，否则会**导致程序出现错误或死锁**。

```
if (rank == 0) {
    msg = 123;
    MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("Process %d sent message %d to process %d\n", rank, msg, 1);
}
```

MPI_Send函数的第一个参数为指向要发送的数据的指针，这里传入了msg的地址；第二个参数为要发送的数据数量，这里只发送了一个整型数据，因此传入1；第三个参数为要发送的数据类型，这里为MPI_INT；第四个参数为指定接收消息的进程的rank号，这里为1；第五个参数为用于标识消息的tag值，这里为0；第六个参数为通信子，这里使用MPI_COMM_WORLD，指定通信使用的进程组。

```
else if (rank == 1) {
    MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Process %d received message %d from process %d\n", rank, msg, 0);
}
```

MPI_Recv函数的第一个参数为接收数据的缓冲区，这里传入了msg的地址；第二个参数为要接收的数据数量，这里也只接收了一个整型数据，因此传入1；第三个参数为要接收的数据类型，这里为MPI_INT；第四个参数为指定发送消息的进程的rank号，这里为0；第五个参数为用于标识消息的tag值，这里为0；第六个参数为通信子，这里使用MPI_COMM_WORLD，指定通信使用的进程组；第七个参数为MPI_Status类型的指针，用于返回接收到的消息的状态信息。

MPI_Bcast、MPI_Scatter和MPI_Gather是MPI中常用的集合通信函数，用于在进程之间进行数据的广播、散布和收集。

- 广播：指定一个根进程，把一份数据复制多份发送给其他所有comm中的进程。在使用广播操作时，其他进程不需要显式地接收数据就可以接收到广播的数据。
- Scatter：指定一个根进程，将一个较大的数据集分割成子集，并将每个子集发送给其他进程。在Scatter函数中，需要指定根进程中存储的原始数据集、子集大小和每个进程接收子集的缓冲区。每个进程只能接收到它自己的子集，而不是整个数据集。Scatter操作通常用于将一个较大的数据集划分为多个子集，以便并行地处理这些子集。
- Gather：在Gather操作中，每个进程都有一个缓冲区，用于存储它要发送的数据；而根进程有一个接收缓冲区，用于接收其他进程发送的数据。
- 无论是广播、Scatter还是Gather操作，都涉及到一个根进程，该进程负责发送或接收数据。在MPI中，根进程可以是任何一个进程，但通常情况下，我们会选择rank为0的进程作为根进程。根进程的选择通常取决于具体的应用场景和需求。

1. MPI_Bcast

MPI_Bcast函数用于将一个进程中的数据广播到所有其他进程中，其语法如下：

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm
comm)
```

参数含义如下：

- buf：指向要广播的数据的指针，类型为void*。
- count：要广播的数据的数量。
- datatype：要广播的数据的类型，如MPI_INT、MPI_DOUBLE等。

- **root：指定广播数据的进程的rank号，该进程的数据会被广播到所有其他进程中。**
- comm：通信子，指定通信使用的进程组。

MPI_Bcast函数的特点如下：

- 方便简洁：MPI_Bcast函数可以方便地将一个进程中的数据广播到所有其他进程中，无需使用循环或其他复杂的逻辑。
- 高效可靠：MPI_Bcast函数使用树形结构进行广播，可以有效减少数据传输的次数和数据传输的时间，同时MPI_Bcast函数保证了广播数据的可靠性，即所有进程都能正确地接收到广播的数据。

2. MPI_Scatter

MPI_Scatter函数用于将一个进程中的数据分散到所有其他进程中，其语法如下：

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

参数含义如下：

- **sendbuf：指向要发送的数据的指针，类型为const void*，只在根进程中有意义。根进程中的数据会被分散到其他进程中。**
- sendcount：要发送的数据的数量，只在根进程中有意义。
- sendtype：要发送的数据的类型，如MPI_INT、MPI_DOUBLE等，只在根进程中有意义。
- **recvbuf：指向接收数据的缓冲区的指针，类型为void*，在所有进程中都有意义。**
- **recvcount：每个进程接收的数据数量。**
- recvtype：接收的数据类型，如MPI_INT、MPI_DOUBLE等。
- **root：指定分散数据的进程的rank号，该进程的数据会被分散到所有其他进程中。**
- comm：通信子，指定通信使用的进程组。

注意事项如下：

- MPI_Scatter函数的发送和接收操作必须在所有进程中同时进行
- 在使用MPI_Scatter函数时，需要确保发送和接收的进程数量和数据类型匹配，否则可能会导致数据接收错误或崩溃。

3. MPI_Gather

MPI_Gather函数用于将所有进程中的数据收集到一个进程中，其语法如下：

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

参数含义如下：

- sendbuf：指向要发送的数据的指针，类型为const void*，在所有进程中都有意义。
- sendcount：要发送的数据的数量。
- sendtype：要发送的数据的类型，如MPI_INT、MPI_DOUBLE等。
- **recvbuf：指向接收数据的缓冲区的指针，类型为void*，只在根进程中有意义。所有进程中的数据会被收集到根进程中。**
- recvcount：每个进程接收的数据数量，只在根进程中有意义。
- recvtype：接收的数据类型，如MPI_INT、MPI_DOUBLE等，只在根进程中有意义。
- **root：指定收集数据的进程的rank号，该进程会收集所有其他进程中的数据。**
- comm：通信子，指定通信使用的进程组。

MPI_Reduce、MPI_Allreduce和MPI_Scan，这三个函数都是MPI中的集合通信函数，用于在多个进程之间共享数据。

1. MPI_Reduce函数：该函数将一个进程中的数据聚合到根进程中，并将聚合后的结果存储在根进程中的缓冲区中。MPI_Reduce函数可以用于对数据进行加、乘、最大值或最小值等操作。在MPI_Reduce函数中，需要指定要聚合的数据、聚合操作的类型、根进程的rank号和通信子。MPI_Reduce函数只能将一个进程中的数据聚合到根进程中，不能将数据广播给所有进程。
2. MPI_Allreduce函数：该函数将所有进程中的数据聚合到每个进程中，并将聚合后的结果存储在每个进程的缓冲区中。MPI_Allreduce函数可以用于对数据进行加、乘、最大值或最小值等操作。在MPI_Allreduce函数中，需要指定要聚合的数据、聚合操作的类型和通信子。MPI_Allreduce函数可以将数据广播给所有进程，并且每个进程都可以获得聚合后的结果，因此**它常用于在所有进程之间共享数据**。
3. MPI_Scan函数：该函数将一个进程中的数据按照指定的操作类型依次聚合到所有进程中，并将聚合后的结果存储在每个进程的缓冲区中。MPI_Scan函数可以用于对数据进行加、乘、最大值或最小值等操作。在MPI_Scan函数中，需要指定要聚合的数据、聚合操作的类型和通信子。MPI_Scan函数与MPI_Allreduce函数类似，但它会依次将每个进程中的数据与前面进程的聚合结果进行聚合，从而得到所有进程的聚合结果。因此，**MPI_Scan函数常用于需要依次聚合数据的应用场景**

需要注意的是，**MPI_Reduce、MPI_Allreduce和MPI_Scan函数都是阻塞操作，即在函数返回之前，所有进程必须完成发送和接收操作。**

1. MPI_Reduce函数

MPI_Reduce函数用于将一个进程中的数据聚合到根进程中，并将聚合后的结果存储在根进程中的缓冲区中。在MPI_Reduce函数中，参数的含义如下：

- sendbuf：指向要发送数据的缓冲区的指针。
- recvbuf：指向根进程中接收数据的缓冲区的指针。
- count：要发送的数据的数量。
- datatype：发送和接收数据的数据类型。
- op：聚合操作的类型，例如MPI_SUM、MPI_MAX等。
- root：根进程的rank号。
- comm：通信子。

下面是一个具体的MPI_Reduce函数的例子，假设我们有4个进程，每个进程都有一个整数数组，需要计算所有进程的数组元素之和，并将结果存储在rank为0的进程中。代码如下：

```
#include <stdio.h>
#include <mpi.h>

int main(int argc,char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int data[size];
    for (int i = 0; i < size; i++) {
        data[i] = i + rank;
    }

    int sum = 0;
    MPI_Reduce(&data[rank], &sum, size, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sum of array elements: %d\n", sum);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

在上面的例子中，我们首先创建了一个整数数组data，其中每个元素的值为进程rank号加上数组下标i。然后我们使用MPI_Reduce函数将所有进程中的data数组元素之和聚合到rank为0的进程中，并将结果存储在sum变量中。在MPI_Reduce函数中，我们指定了要聚合的数据（&data[rank]）、数据数量（size）、数据类型（MPI_INT）、聚合操作类型（MPI_SUM）、根进程的rank号（0）和通信子（MPI_COMM_WORLD）。

2. MPI_Allreduce函数

MPI_Allreduce函数用于将所有进程中的数据聚合到每个进程中，并将聚合后的结果存储在每个进程的缓冲区中。在MPI_Allreduce函数中，参数的含义与MPI_Reduce函数类似，如下：

- sendbuf：指向要发送数据的缓冲区的指针。
- recvbuf：指向接收数据的缓冲区的指针。
- count：要发送的数据的数量。
- datatype：发送和接收数据的数据类型。
- op：聚合操作的类型，例如MPI_SUM、MPI_MAX等。
- comm：通信子。

下面是一个具体的MPI_Allreduce函数的例子，假设我们有4个进程，每个进程都有一个整数数组，需要计算所有进程的数组元素的最大值，并将结果存储在每个进程的max变量中。代码如下：

```
#include <stdio.h>
#include <mpi.h>

int main(int argc,char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int data[size];
    for (int i = 0; i <size; i++) {
        data[i] = i + rank;
    }

    int max = 0;
    MPI_Allreduce(&data[rank], &max, size, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

    printf("Max of array elements for process %d: %d\n", rank, max);

    MPI_Finalize();
    return 0;
}
```

在上面的例子中，我们首先创建了一个整数数组data，其中每个元素的值为进程rank号加上数组下标i。然后我们使用MPI_Allreduce函数将所有进程中的data数组元素的最大值聚合到每个进程的max变量中，并将结果存储在max变量中。在MPI_Allreduce函数中，我们指定了要聚合的数据（&data[rank]）、数据数量（size）、数据类型（MPI_INT）、聚合操作类型（MPI_MAX）和通信子（MPI_COMM_WORLD）。

3. MPI_Scan函数

MPI_Scan函数用于将一个进程中的数据按照指定的操作类型依次聚合到所有进程中，并将聚合后的结果存储在每个进程的缓冲区中。在MPI_Scan函数中，参数的含义如下：

- sendbuf：指向要发送数据的缓冲区的指针。

- recvbuf：指向接收数据的缓冲区的指针。
- count：要发送的数据的数量。
- datatype：发送和接收数据的数据类型。
- op：聚合操作的类型，例如MPI_SUM、MPI_MAX等。
- comm：通信子。

MPI数据分割范式

均匀划分

使用Scatter

```
int rank, size;
int *data, *local_data;
int n = 100;
int block_size;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 计算每个处理单元的块大小
block_size = n / size;

// 分配数据
data = (int*)malloc(n * sizeof(int));
local_data = (int*)malloc(block_size * sizeof(int));

// 初始化数据
for (int i = 0; i < n; i++) {
    data[i] = i;
}

// 进行数据划分
MPI_Scatter(data, block_size, MPI_INT, local_data, block_size, MPI_INT, 0,
MPI_COMM_WORLD);

// 进行数据处理
for (int i = 0; i < block_size; i++) {
    local_data[i] *= 2;
}

// 将处理后的数据收集到根进程中
MPI_Gather(local_data, block_size, MPI_INT, data, block_size, MPI_INT, 0,
MPI_COMM_WORLD);

// 释放内存
free(data);
free(local_data);
```

等块划分（使用for循环）

等间隔划分(1,size,2size,3size...)（使用for循环）

不均匀划分

使用Scatterv

```
int rank, size;
int *data, *local_data;
int n = 100;
int block_sizes[4] = {20, 30, 10, 40};
int displs[4] = {0, 20, 50, 60};

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 分配数据
data = (int*)malloc(n * sizeof(int));
local_data = (int*)malloc(block_sizes[rank] * sizeof(int));

// 初始化数据
for (int i = 0; i < n; i++) {
    data[i] = i;
}

// 进行数据划分
MPI_Scatterv(data, block_sizes, displs, MPI_INT, local_data, block_sizes[rank],
MPI_INT, 0, MPI_COMM_WORLD);

// 进行数据处理
for (int i = 0; i < block_sizes[rank]; i++) {
    local_data[i] *= 2;
}

// 将处理后的数据收集到根进程中
MPI_Gatherv(local_data, block_sizes[rank], MPI_INT, data, block_sizes, displs,
MPI_INT, 0, MPI_COMM_WORLD);

// 释放内存
free(data);
free(local_data);
```

使用for循环生成索引

pthread

以下是10个常见的 `pthread` 范式，每个范式都有注释说明：

范式一：创建线程

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h> // 添加头文件

void *thread_func(void *arg) {
    // 线程函数的代码
    int rank = *(int *)arg; // 修改类型
    printf("rank: %ld\n", rank); // 修改格式化字符串
```

```
    return NULL;
}

int main() {
    int num = 4;
    pthread_t* thread;
    thread = (pthread_t* )malloc (num * sizeof(pthread_t));
    for (int i = 0; i < num; i++) {
        pthread_create(thread+i,NULL,thread_func,(void*)i); // 修改类型转换
    }
    for (int i = 0; i < num; i++) {
        pthread_join(thread[i],NULL);
    }
    free(thread);
    // 主线程的代码
    return 0;
}
```

【创建数组】

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h> // 添加头文件
#define NUM 4

void *thread_func(void *arg) {
    // 线程函数的代码
    int rank = *(int *)arg; // 修改类型
    printf("rank: %ld\n", rank); // 修改格式化字符串
    return NULL;
}

int main() {
    pthread_t thread[NUM];
    for (int i = 0; i < NUM; i++) {
        pthread_create(&thread[i],NULL,thread_func,(void*)i); // 修改类型转换
    }
    for (int i = 0; i < NUM; i++) {
        pthread_join(thread[i],NULL);
    }

    return 0;
}
```

范式四：线程互斥锁

```
#include <pthread.h>

pthread_mutex_t lock;

void *thread_func(void *arg) {
    // 加锁
    pthread_mutex_lock(&lock);
    // 临界区
    // 解锁
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

```c
int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

范式五：线程条件变量

```c
#include <pthread.h>

pthread_mutex_t lock;
pthread_cond_t cond;

void *thread_func(void *arg) {
    // 加锁
    pthread_mutex_lock(&lock);
    // 等待条件
    pthread_cond_wait(&cond, &lock);
    // 条件成立，执行代码
    // 解锁
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_create(&thread, NULL, thread_func, NULL);
    // 主线程的代码
    // 发送条件信号
    pthread_cond_signal(&cond);
    pthread_join(thread, NULL);
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
    return 0;
}
```

范式六：线程信号量

```c
#include <pthread.h>
#include <semaphore.h>

sem_t sem;

void *thread_func(void *arg) {
    // 等待信号量
    sem_wait(&sem);
    // 信号量可用，执行代码
    return NULL;
}

int main() {
    pthread_t thread;
    sem_init(&sem, 0, 0);
```

```
    pthread_create(&thread, NULL, thread_func, NULL);
    // 主线程的代码
    // 发送信号量
    sem_post(&sem);
    pthread_join(thread, NULL);
    sem_destroy(&sem);
    return 0;
}
```

范式八：线程取消

```
#include <pthread.h>

void *thread_func(void *arg) {
    // 设置线程取消状态为启用
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    // 设置线程取消类型为异步取消
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    // 执行代码
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    // 主线程的代码
    // 取消线程
    pthread_cancel(thread);
    pthread_join(thread, NULL);
    return 0;
}
```

范式十：线程池

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

#define MAX_QUEUE_SIZE 100
#define MAX_THREADS 10

typedef struct Task {
    void (*func)(void *);
    void *arg;
} Task;

typedef struct ThreadPool {
    Task *queue[MAX_QUEUE_SIZE];
    int front;
    int rear;
    int size;
    pthread_mutex_t queue_mutex;
    pthread_cond_t queue_not_empty;
    pthread_cond_t queue_not_full;
    pthread_t threads[MAX_THREADS];
    int thread_count;
    int is_shutdown;
} ThreadPool;

void *thread_pool_worker(void *arg) {
```

```
    ThreadPool *pool = (ThreadPool *)arg;
    Task *task;
    while (1) {
        pthread_mutex_lock(&pool->queue_mutex);
        while (pool->size == 0 && !pool->is_shutdown) {
            pthread_cond_wait(&pool->queue_not_empty, &pool->queue_mutex);
        }
        if (pool->size == 0 && pool->is_shutdown) {
            pthread_mutex_unlock(&pool->queue_mutex);
            pthread_exit(NULL);
        }
        task = pool->queue[pool->front];
        pool->front = (pool->front + 1) % MAX_QUEUE_SIZE;
        pool->size--;
        pthread_cond_signal(&pool->queue_not_full);
        pthread_mutex_unlock(&pool->queue_mutex);
        (*(task->func))(task->arg);
        free(task);
    }
    pthread_exit(NULL);
}

ThreadPool *thread_pool_create(int thread_count) {
    ThreadPool *pool = (ThreadPool *)malloc(sizeof(ThreadPool));
    // 初始化锁和信号量
    pthread_mutex_init(&pool->queue_mutex, NULL);
    pthread_cond_init(&pool->queue_not_empty, NULL);
    pthread_cond_init(&pool->queue_not_full, NULL);
    pool->front = 0;
    pool->rear = 0;
    pool->size = 0;
    pool->thread_count = thread_count;
    pool->is_shutdown = 0;

    // 初始化给n个线程分配好worker，worker持续执行
    for (int i = 0; i < thread_count; i++) {
        pthread_create(&pool->threads[i], NULL, thread_pool_worker, (void
*)pool);
    }

    return pool;
}

void thread_pool_submit(ThreadPool *pool, void (*func)(void *), void *arg) {
    pthread_mutex_lock(&pool->queue_mutex);
    while (pool->size == MAX_QUEUE_SIZE && !pool->is_shutdown) {
        pthread_cond_wait(&pool->queue_not_full, &pool->queue_mutex);
    }
    if (pool->is_shutdown) {
        pthread_mutex_unlock(&pool->queue_mutex);
        return;
    }
    Task *task = (Task *)malloc(sizeof(Task));
    task->func = func;
    task->arg = arg;
    pool->queue[pool->rear] = task;
    pool->rear = (pool->rear + 1) % MAX_QUEUE_SIZE;
    pool->size++;
    pthread_cond_signal(&pool->queue_not_empty);
    pthread_mutex_unlock(&pool->queue_mutex);
}
```

```c
void thread_pool_shutdown(ThreadPool *pool) {
    pthread_mutex_lock(&pool->queue_mutex);
    pool->is_shutdown = 1;
    pthread_cond_broadcast(&pool->queue_not_empty);
    pthread_mutex_unlock(&pool->queue_mutex);

    for (int i = 0; i < pool->thread_count; i++) {
        pthread_join(pool->threads[i], NULL);
    }

    pthread_mutex_destroy(&pool->queue_mutex);
    pthread_cond_destroy(&pool->queue_not_empty);
    pthread_cond_destroy(&pool->queue_not_full);
    free(pool);
}

// 以下是一个使用线程池的示例函数
void print_hello(void *arg) {
    int tid = *(int *)arg;
    printf("Hello World from thread %d\n", tid);
}

int main() {
    ThreadPool *pool = thread_pool_create(4);
    int *args[10];
    for (int i = 0; i < 10; i++) {
        args[i] = (int *)malloc(sizeof(int));
        *args[i] = i;
        thread_pool_submit(pool, print_hello, args[i]);
    }
    thread_pool_shutdown(pool);
    for (int i = 0; i < 10; i++) {
        free(args[i]);
    }
    return 0;
}
```

快速排序

```c
// The Data sorted
struct Data {
    Data (int* num, int l, int r) : num(num), l(l), r(r) {}
    int* num;
    int l, r;
};

struct Queue {
    Data* data;
    int head, tail;
    int size;
    // 使用信号量：Thread 同步
    sem_t* full, *empty, *PushMutex, *PopMutex;
};

Queue* initQueue(int n) {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->data = (Data*)malloc(sizeof(Data) * n);
    // Init semorephore
    q->full = (sem_t*)malloc(sizeof(sem_t));
```

```
    q->PushMutex = (sem_t*)malloc(sizeof(sem_t));
    q->PopMutex = (sem_t*)malloc(sizeof(sem_t));
    q->empty = (sem_t*)malloc(sizeof(sem_t));
    sem_init(q->empty, 0, n);
    sem_init(q->full, 0, 0);
    sem_init(q->PushMutex, 0, 1);
    sem_init(q->PopMutex, 0, 1);
    q->head = q->tail = 0;
    q->size = n;
    for (int i = 0; i < q->size; i++) {
        q->data[i].num = NULL;
    }
    return q;
}

void push(Queue* q, Data&& data) {
    sem_wait(q->PushMutex);
    sem_wait(q->empty);
    q->data[q->head] = data;
    ++q->head %= q->size;
    sem_post(q->full);
    sem_post(q->PushMutex);
}

Data pop(Queue* q) {
    sem_wait(q->PopMutex);
    sem_wait(q->full);
    Data ret = q->data[q->tail];
    q->data[q->tail].num = NULL;
    ++q->tail %= q->size;
    sem_post(q->empty);
    sem_post(q->PopMutex);
    return ret;
}

void clear(Queue* q) {
    free(q->PushMutex);
    free(q->PopMutex);
    free(q->full);
    free(q->empty);
    free(q->data);
    free(q);
}

/*
* 不需要两次尾递归，只需要右半部分递归，左半部分留在下一轮循环执行即可
*/
void quick_sort(int* num, int l, int r) {
    if (num == NULL) return;
    if (l >= r) return;
    while (l < r) {
        int x = l, y = r;
        int z = num[(l + r) >> 1];
        do {
            while (num[x] < z) x++;
            while (num[y] > z) y--;
            if (x <= y) {
                int temp = num[x];
                num[x] = num[y];
                num[y] = temp;
                ++x, --y;
            }
```

```
        } while (x <= y);
        quick_sort(num, x, r);
        r = y;
    }
}

// 划分数据集
void* thread_work(void* arg) {
    Queue* q = (Queue*)arg;
    while (1) {
        Data data = pop(q);
        if (thread_ok) break;
        if (data.r - data.l <= INTERVAL) {
            quick_sort(data.num, data.l, data.r);
        } else {
            int x = data.l, y = data.r;
            int ans = data.num[(data.l + data.r) >> 1];
            do {
                while (data.num[x] < ans) x++;
                while (data.num[y] > ans) y--;
                if (x <= y) {
                    int temp = data.num[x];
                    data.num[x] = data.num[y];
                    data.num[y] = temp;
                    x++, y--;
                }
            } while (x <= y);
            push(q, {data.num, data.l, y});
            push(q, {data.num, x, data.r});
        }
    }
    return NULL;
}

// 不断地检查相邻元素的大小关系，来判断是否已经有序
// 像是一个自旋锁，只有整个序列有序时才会返回
// 完全可以不需要这个逻辑，替换为其他互斥方法
void* thread_check(void* arg) {
    Data* data = (Data*)arg;
    bool flag = 1;
    while (flag) {
        flag = 0;
        for (int i = data->l; i < data->r; i++) {
            if (data->num[i] <= data->num[i + 1]) continue;
            flag = 1;
        }
    }
    return NULL;
}

void with_thread_quick_sort(int* num, int len, int threadNum) {
    Queue* q = initQueue(len / 2);
    push(q, {num, 0, len - 1});
    pthread_t thread[threadNum];
    for (int i = 0; i < threadNum; i++) {
        pthread_create(thread + i, NULL, thread_work, q);
    }
    Data data = {num, 0, len - 1};
    pthread_t check_thread;
    pthread_create(&check_thread, NULL, thread_check, &data);
    pthread_join(check_thread, NULL);
    thread_ok = 1;
```

```
    for (int i = 0; i < threadNum; i++) {
        sem_post(q->full);
    }
    for (int i = 0; i < threadNum; i++) {
        pthread_join(thread[i], NULL);
    }
    clear(q);
}

int main() {
    srand(time(NULL));
    int num[MAX];
    for (int i = 0; i < MAX; i++) {
        num[i] = rand() % MAX + 1;
    }

    // Benchmark test for sequential quick sort
    double seqMin = 1e10, seqMax = -1e10, seqAvg = 0.0;
    for (int i =0; i < 3; i++) {
        int* seqNum = new int[MAX];
        copy(num, num + MAX, seqNum);
        chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
        quick_sort_multiThreads(seqNum, 0, MAX - 1);
        chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
        double duration = chrono::duration_cast<chrono::duration<double>>(t2 -
t1).count();
        seqMin = min(seqMin, duration);
        seqMax = max(seqMax, duration);
        seqAvg += duration;
        delete[] seqNum;
    }
    seqAvg /= 3.0;

    // Benchmark test for concurrent quick sort
    double conMin = 1e10, conMax = -1e10, conAvg = 0.0;
    for (int i = 0; i < 3; i++) {
        int* conNum = new int[MAX];
        copy(num, num + MAX, conNum);
        chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
        // sort(conNum, conNum + MAX);
        with_thread_quick_sort(conNum, MAX, 9);
        chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
        double duration = chrono::duration_cast<chrono::duration<double>>(t2 -
t1).count();
        conMin = min(conMin, duration);
        conMax= max(conMax, duration);
        conAvg += duration;
        delete[] conNum;
    }
    conAvg /= 3.0;

    // Output benchmark results
    printf("List Size       Sequential      Sort      Time (s)
Concurrent      Sort      Time (s)\n");
    printf("              min            max            average         min
     max            average\n");
    printf("-----------   ------------   ------------   ------------    ---------
---   ------------   ------------\n");
    printf("%-9d     %-12.6f   %-12.6f   %-12.6f    %-12.6f   %-12.6f
%-12.6f\n", MAX, seqMin, seqMax, seqAvg, conMin, conMax, conAvg);

    return 0;
```

```
}
```

计算ln2

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

// 全局结果
double ans = 0.0;
// 数据量
int n = 1000000;
// 线程数量
int tnum = 8;
// 全局互斥锁
pthread_mutex_t lock;

void* calculn2(void* arg);

int main() {
    // 初始化
    pthread_t* threads;
    threads = (pthread_t*) malloc(sizeof(pthread_t)*tnum);
    pthread_mutex_init(&lock, NULL);

    // 分配任务
    for (int i = 0; i < tnum; i++) {
        pthread_create(&threads[i], NULL, calculn2, (void*)i);
    }
    // 结束和资源回收
    for (int i = 0; i < tnum; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("ans: %f\n",ans);
    pthread_mutex_destroy(&lock);
    free(threads);
    pthread_exit(NULL);
    return 0;
}


// 任务内串行逻辑
void* calculn2(void* arg) {
    int rank = (int) arg;
    // 单个进程的数据量
    int chunk = n/tnum;
    // 划分数据集
    int start = rank*chunk;
    int end = start + chunk;

    if (rank == tnum-1) {
        end = n;
    }
    double factor;
    if (start % 2 == 0) {
        factor = 1.0;
    } else {
        factor = -1.0;
    }
```

```
    double my_ln2 = 0.0;
    for (int i = start; i < end; ++i) {
        // start 从0开始，所以应该是start+1
        my_ln2 += factor/(i + 1);
        factor = -factor;
    }
    // 互斥修改全局值
    pthread_mutex_lock(&lock);
    ans += my_ln2;
    pthread_mutex_unlock(&lock);

    return NULL;
}
```

计算π

蒙特卡洛法

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define NUM 8
int n = 10000000;
double pi = 0.0;
int hint_num = 0;
pthread_mutex_t lock;

void* calcupi(void* arg) {
    int rank = (int) arg;
    // 数据量
    int num = n/NUM;
    double x,y,dist;
    int local_hint_num = 0;
    for (int i = 0; i < num; i++) {
        x = (double)rand()/(double)RAND_MAX;
        y = (double)rand()/(double)RAND_MAX;
        dist = x*x + y*y;
        if (dist <= 1) {
            local_hint_num++;
        }
    }
    pthread_mutex_lock(&lock);
    hint_num += local_hint_num;
    pthread_mutex_unlock(&lock);
}

int main() {
    // init
    pthread_mutex_init(&lock, NULL);
    pthread_t threads[NUM];
    for (int i = 0; i < NUM; i++) {
        pthread_create(&threads[i], NULL, calcupi, (void*)i);
    }

    for (int i = 0; i < NUM; i++) {
        pthread_join(threads[i],NULL);
    }
    pi = 4*(double)hint_num/n;
```

```
    printf("pi: %f\n",pi);
    pthread_mutex_destroy(&lock);
    pthread_exit(NULL);
    return 0;
}
```

积分法

【不考虑顺序，可以不分片】

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define NUM 8
int n = 10000000;
double pi = 0.0;

pthread_mutex_t lock;

void* calcupi(void* arg) {
    int rank = (int) arg;

    double x, local = 0.0;
    for (int i = rank; i < n; i+=NUM) {
        x = (i+0.5)/(double)n;
        local += 4.0/(1+x*x);
    }
    printf("local: %f\n",local);
    pthread_mutex_lock(&lock);
    pi += local / (double)n;
    pthread_mutex_unlock(&lock);
}

int main() {
    // init
    pthread_mutex_init(&lock, NULL);
    pthread_t threads[NUM];
    for (int i = 0; i < NUM; i++) {
        pthread_create(&threads[i], NULL, calcupi, (void*)i);
    }

    for (int i = 0; i < NUM; i++) {
        pthread_join(threads[i],NULL);
    }
    printf("pi: %f\n",pi);
    pthread_mutex_destroy(&lock);
    pthread_exit(NULL);
    return 0;
}
```

辛普生法

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

#define NUM_THREADS 4 // 线程数量
#define N 100000000 // 积分区间上限
```

```c
#define a 0.0 // 积分区间下限
#define b 1.0 // 积分区间上限

double h = (b - a) / N; // 步长
double sum = 0.0; // 总和
pthread_mutex_t mutex; // 互斥锁

// 辛普森法
double f(double x) {
    return 4.0 / (1.0 + x * x);
}

// 计算积分
void *integrate(void *threadid) {
    long tid = (long)threadid;
    double x;
    double localsum = 0.0;
    int i;
    int start = tid * N / NUM_THREADS;
    int end = (tid + 1) * N / NUM_THREADS;

    for (i = start + 1; i < end; i += 2) {
        x = a + i * h;
        localsum += 4.0 * f(x);
    }

    for (i = start + 2; i < end - 1; i += 2) {
        x = a + i * h;
        localsum += 2.0 * f(x);
    }

    pthread_mutex_lock(&mutex);
    sum += localsum;
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    pthread_mutex_init(&mutex, NULL);

    for (t = 0; t < NUM_THREADS; t++) {
        rc = pthread_create(&threads[t], NULL, integrate, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    pthread_mutex_destroy(&mutex);

    double pi = h * (f(a) + f(b) + 2.0 * sum) / 3.0;
    printf("Pi = %f\n", pi);

    pthread_exit(NULL);
```

```
}
```

输出奇数偶数

【锁+条件变量】

```
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int buffer;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t odd_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t even_available = PTHREAD_COND_INITIALIZER;
bool is_empty = true;
bool is_odd = false;
bool is_even = false;

void* Producer(void* arg) {
    while (true) {
        pthread_mutex_lock(&lock);
        while (!is_empty) {
            pthread_cond_wait(&data_available, &lock);
        }
        int k = rand() % 100;
        printf("producing k = %d\n", k);
        buffer = k;
        if (k % 2 == 0) {
            is_even = true;
            pthread_cond_signal(&even_available);
        } else {
            is_odd = true;
            pthread_cond_signal(&odd_available);
        }
        is_empty = false;
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

void* Consumers_odd(void* arg) {
    while (true) {
        pthread_mutex_lock(&lock);
        while (!is_odd) {
            pthread_cond_wait(&odd_available, &lock);
        }
        printf("get odd number: %d\n", buffer);
        is_odd = false;
        is_empty = true;
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

void* Consumers_even(void* arg) {
    while (true) {
```

```
        pthread_mutex_lock(&lock);
        while (!is_even) {
            pthread_cond_wait(&even_available, &lock);
        }
        printf("get even number: %d\n", buffer);
        is_even = false;
        is_empty = true;
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

int main() {
    pthread_t producer_thread, consumer_odd_thread, consumer_even_thread;

    // create threads
    pthread_create(&producer_thread, NULL, Producer, NULL);
    pthread_create(&consumer_odd_thread, NULL, Consumers_odd, NULL);
    pthread_create(&consumer_even_thread, NULL, Consumers_even, NULL);

    // join all threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_odd_thread, NULL);
    pthread_join(consumer_even_thread, NULL);

    return 0;
}
```

```
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 8

int buffer[BUFFER_SIZE];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t odd_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t even_available = PTHREAD_COND_INITIALIZER;
int odd = 0;
int even = 0;

void* Producer(void* arg) {
    while (true) {
        pthread_mutex_lock(&lock);
        while ((odd + even) >= BUFFER_SIZE) {
            pthread_cond_wait(&data_available, &lock);
        }
        int k = rand() % 100;
        printf("producing k = %d\n", k);
        buffer[odd + even] = k;
        if (k % 2 == 0) {
            even++;
            pthread_cond_signal(&even_available);
        } else {
            odd++;
            pthread_cond_signal(&odd_available);
```

```
        }
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

void* Consumers_odd(void* arg) {
    while (true) {
        pthread_mutex_lock(&lock);
        while (!odd) {
            pthread_cond_wait(&odd_available, &lock);
        }
        printf("get odd number: %d\n", buffer[odd - 1]);
        odd--;
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

void* Consumers_even(void* arg) {
    while (true) {
        pthread_mutex_lock(&lock);
        while (!even) {
            pthread_cond_wait(&even_available, &lock);
        }
        printf("get even number: %d\n", buffer[even - 1]);
        even--;
        pthread_cond_signal(&data_available);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
}

int main() {
    pthread_t producer_thread, consumer_odd_thread, consumer_even_thread;

    // create threads
    pthread_create(&producer_thread, NULL, Producer, NULL);
    pthread_create(&consumer_odd_thread, NULL, Consumers_odd, NULL);
    pthread_create(&consumer_even_thread, NULL, Consumers_even, NULL);

    // join all threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_odd_thread, NULL);
    pthread_join(consumer_even_thread, NULL);

    // destroy mutex and conditions
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&data_available);
    pthread_cond_destroy(&odd_available);
    pthread_cond_destroy(&even_available);

    pthread_exit(NULL);
    return 0;
}
```

【使用信号量实现（PV操作）】

```
#include <pthread.h>
```

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>

int buffer;
sem_t empty, odd, even;

void* Producer(void* arg) {
    while (true) {
        sem_wait(&empty);
        int k = rand() % 100;
        printf("producing k = %d\n", k);
        buffer = k;
        if (k % 2 == 0) {
            sem_post(&even);
        } else {
            sem_post(&odd);
        }
        sleep(1);
    }
}

void* Consumers_odd(void* arg) {
    while (true) {
        sem_wait(&odd);
        printf("get odd number: %d\n", buffer);
        sem_post(&empty);
        sleep(1);
    }
}

void* Consumers_even(void* arg) {
    while (true) {
        sem_wait(&even);
        printf("get even number: %d\n", buffer);
        sem_post(&empty);
        sleep(1);
    }
}

int main() {
    sem_init(&empty, 0, 1);
    sem_init(&even, 0, 0);
    sem_init(&odd, 0, 0);
    pthread_t producer_thread, consumer_odd_thread, consumer_even_thread;

    // create threads
    pthread_create(&producer_thread, NULL, Producer, NULL);
    pthread_create(&consumer_odd_thread, NULL, Consumers_odd, NULL);
    pthread_create(&consumer_even_thread, NULL, Consumers_even, NULL);

    // join all threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_odd_thread, NULL);
    pthread_join(consumer_even_thread, NULL);

    // destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&odd);
    sem_destroy(&even);
```

```
    pthread_exit(NULL);
    return 0;
}
```

素数

```
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>

#define NUM 8
int n = 1000000;
int count = 0;
pthread_mutex_t lock;

bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void* task(void* arg) {
    int rank = (int) arg;
    int chunk_size = n / NUM;
    int start = rank * chunk_size;
    int end = (rank + 1) * chunk_size;
    if (rank == NUM - 1) {
        end = n;
    }
    int local_count = 0;
    for (int i = start + 1; i <= end - 1; i += 2) {
        if (is_prime(i) && is_prime(i + 2)) {
            local_count++;
        }
    }
    pthread_mutex_lock(&lock);
    count += local_count;
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_mutex_init(&lock, NULL);
    pthread_t threads[NUM];
    for (int i = 0; i < NUM; i++) {
        pthread_create(&threads[i], NULL, task, (void*)i);
    }

    for (int i = 0; i < NUM; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    printf("在小于 %d 的整数范围内，连续奇数都是素数的情况的次数为: %d\n", n, count);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

水仙花数

完全数

顺序输出123

条件变量:

```
#include <bits/pthreadtypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM 3

pthread_mutex_t mutex;
int p2_done = 0;
int p3_done = 0;
pthread_cond_t cond1;
pthread_cond_t cond2;

void* task(void *arg) {
    int rank = *(int *)arg;
    printf("rank: %d\n", rank);
    if (rank == 1) {
        pthread_mutex_lock(&mutex);
        while (!p2_done) {
            pthread_cond_wait(&cond1, &mutex);
        }
        printf("1\n");
        pthread_mutex_unlock(&mutex);

    } else if (rank == 2) {
        pthread_mutex_lock(&mutex);
        while (!p3_done) {
            pthread_cond_wait(&cond2, &mutex);
        }
        printf("2\n");
        p2_done = 1;
        pthread_cond_signal(&cond1);
        pthread_mutex_unlock(&mutex);
    } else {
        pthread_mutex_lock(&mutex);
        printf("3\n");
        p3_done = 1;
        pthread_cond_signal(&cond2);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
int main() {
    // initialize 3 threads
    pthread_t* threads;
    threads = (pthread_t* ) malloc(NUM);
    // allocate tasks
    for (int i = 0; i < NUM; ++i) {
        pthread_create(&threads[i], NULL, task, &i);
    }
    // join all tasks
    for (int i = 0; i < NUM; ++i) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

信号量:

```
#include <bits/pthreadtypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM 3

pthread_mutex_t mutex;
sem_t sem1;
sem_t sem2;

void* task(void *arg) {
    int rank = *(int *)arg;
    if (rank == 1) {
        sem_wait(&sem1);
        printf("1\n");

    } else if (rank == 2) {
        sem_wait(&sem2);
        printf("2\n");
        sem_post(&sem1);
    } else {
        printf("3\n");
        sem_post(&sem2);
    }
    return NULL;
}

int main() {
    // initialize 3 threads
    pthread_t* threads;
    threads = (pthread_t* ) malloc(NUM);
    sem_init(&sem1, 0, 0);  // 初始化信号量为0
    sem_init(&sem2, 0, 0);  // 初始化信号量为0
    // allocate tasks
    for (int i = 0; i < NUM; ++i) {
        pthread_create(&threads[i], NULL, task, &i);
    }
    // join all tasks
    for (int i = 0; i < NUM; ++i) {
        pthread_join(threads[i], NULL);
    }
```

```
    return 0;
}
```

生产者消费者问题

1. 条件变量实现

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE]; // 缓冲区
int in = 0; // 生产者写入位置
int out = 0; // 消费者读取位置
int count = 0; // 缓冲区中的数据数量

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER; // 缓冲区不满的条件变量
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER; // 缓冲区不空的条件变量

void *producer(void *arg) {
    int id = (int)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (count == BUFFER_SIZE) { // 缓冲区已满，等待缓冲区不满的条件变量
            printf("Producer %d: buffer is full, waiting...\n", id);
            pthread_cond_wait(&not_full, &mutex);
        }
        int item = rand() % 100; // 生产一个随机数
        buffer[in] = item;
        printf("Producer %d: produced item %d at buffer[%d]\n", id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        count++;
        pthread_cond_signal(&not_empty); // 发送缓冲区不空的条件变量
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (count == 0) { // 缓冲区已空，等待缓冲区不空的条件变量
            printf("Consumer %d: buffer is empty, waiting...\n", id);
            pthread_cond_wait(&not_empty, &mutex);
        }
        int item = buffer[out];
        printf("Consumer %d: consumed item %d from buffer[%d]\n", id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        count--;
        pthread_cond_signal(&not_full); // 发送缓冲区不满的条件变量
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
int main() {
    pthread_t producer_threads[3], consumer_threads[3];

    // 创建三个生产者线程和三个消费者线程
    for (int i = 0; i < 3; i++) {
        pthread_create(&producer_threads[i], NULL, producer, (void *)i);
        pthread_create(&consumer_threads[i], NULL, consumer, (void *)i);
    }

    // 等待所有线程完成
    for (int i = 0; i < 3; i++) {
        pthread_join(producer_threads[i], NULL);
        pthread_join(consumer_threads[i], NULL);
    }

    return 0;
}
```

如果加上单个进程的发送和接受限制，需要使用for循环：

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE]; // 缓冲区
int in = 0; // 生产者写入位置
int out = 0; // 消费者读取位置
int count = 0; // 缓冲区中的数据数量

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER; // 缓冲区不满的条件变量
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER; // 缓冲区不空的条件变量

void *producer(void *arg) {
    int id = (int)arg;
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == BUFFER_SIZE) { // 缓冲区已满，等待缓冲区不满的条件变量
            printf("Producer %d: buffer is full, waiting...\n", id);
            pthread_cond_wait(&not_full, &mutex);
        }
        int item = rand() % 100; // 生产一个随机数
        buffer[in] = item;
        printf("Producer %d: produced item %d at buffer[%d]\n", id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        count++;
        pthread_cond_signal(&not_empty); // 发送缓冲区不空的条件变量
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *consumer(void *arg) {
    int id = (int)arg;
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == 0) { // 缓冲区已空，等待缓冲区不空的条件变量
            printf("Consumer %d: buffer is empty, waiting...\n", id);
            pthread_cond_wait(&not_empty, &mutex);
```

```
        }
        int item = buffer[out];
        printf("Consumer %d: consumed item %d from buffer[%d]\n", id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        count--;
        pthread_cond_signal(&not_full); // 发送缓冲区不满的条件变量
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t producer_threads[3], consumer_threads[3];

    // 创建三个生产者线程和三个消费者线程
    for (int i = 0; i < 3; i++) {
        pthread_create(&producer_threads[i], NULL, producer, (void *)i);
        pthread_create(&consumer_threads[i], NULL, consumer, (void *)i);
    }

    // 等待所有线程完成
    for (int i = 0; i < 3; i++) {
        pthread_join(producer_threads[i], NULL);
        pthread_join(consumer_threads[i], NULL);
    }

    return 0;
}
```

2. 信号量实现

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

#define BUFFER_SIZE 5
#define MAX_ITEMS 20

int buffer[BUFFER_SIZE]; // 缓冲区
int in = 0; // 生产者写入位置
int out = 0; // 消费者读取位置

sem_t empty; // 空缓冲区的信号量
sem_t full; // 满缓冲区的信号量
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁

void *producer(void *arg) {
    int id = (int)arg;
    int produced = 0;
    while (produced < MAX_ITEMS) {
        int item = rand() % 100; // 生产一个随机数
        sem_wait(&empty); // 等待空缓冲区
        pthread_mutex_lock(&mutex); // 上锁
        buffer[in] = item;
        printf("Producer %d: produced item %d at buffer[%d]\n", id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        produced++;
        pthread_mutex_unlock(&mutex); // 解锁
        sem_post(&full); // 发送满缓冲区信号
    }
    return NULL;
```

```
}

void *consumer(void *arg) {
    int id = (int)arg;
    int consumed = 0;
    while (consumed < MAX_ITEMS) {
        sem_wait(&full); // 等待满缓冲区
        pthread_mutex_lock(&mutex); // 上锁
        int item = buffer[out];
        printf("Consumer %d: consumed item %d from buffer[%d]\n", id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        consumed++;
        pthread_mutex_unlock(&mutex); // 解锁
        sem_post(&empty); // 发送空缓冲区信号
    }
    return NULL;
}

int main() {
    // 初始化信号量
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_t producer_threads[3], consumer_threads[3];

    // 创建三个生产者线程和三个消费者线程
    for (int i = 0; i < 3; i++) {
        pthread_create(&producer_threads[i], NULL, producer, (void *)i);
        pthread_create(&consumer_threads[i], NULL, consumer, (void *)i);
    }

    // 等待所有线程完成
    for (int i = 0; i < 3; i++) {
        pthread_join(producer_threads[i], NULL);
        pthread_join(consumer_threads[i], NULL);
    }

    // 销毁信号量
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

哲学家进餐问题

【判断奇数偶数，不同的拿筷子顺序】

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

#define N 5  // 哲学家数量
#define LEFT (i + N - 1) % N  // 左邻居编号
#define RIGHT (i + 1) % N     // 右邻居编号

pthread_mutex_t chopsticks[N];   // 筷子数组，每个元素对应一个筷子

void *philosopher(void *arg) {
```

```
    int i = *(int *)arg;
    int left, right;

    if (i % 2 == 0) {   // 偶数号哲学家先拿右边的筷子
        left = RIGHT;
        right = LEFT;
    } else {            // 奇数号哲学家先拿左边的筷子
        left = LEFT;
        right = RIGHT;
    }

    while (1) {
        printf("Philosopher %d is thinking\n", i);
        sleep(rand() % 3);  // 思考一段时间

        printf("Philosopher %d is hungry\n", i);
        pthread_mutex_lock(&chopsticks[left]);   // 拿起左边的筷子
        printf("Philosopher %d picked up left chopstick\n", i);
        pthread_mutex_lock(&chopsticks[right]);  // 拿起右边的筷子
        printf("Philosopher %d picked up right chopstick\n", i);

        printf("Philosopher %d is eating\n", i);
        sleep(rand() % 3);  // 进餐一段时间

        pthread_mutex_unlock(&chopsticks[right]);  // 放下右边的筷子
        printf("Philosopher %d put down right chopstick\n", i);
        pthread_mutex_unlock(&chopsticks[left]);   // 放下左边的筷子
        printf("Philosopher %d put down left chopstick\n", i);
    }
}
int main() {
    pthread_t thread_ids[N];
    int ids[N];
    int i;

    // 初始化筷子
    for (i = 0; i < N; i++) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    // 创建哲学家线程
    for (i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&thread_ids[i], NULL, philosopher, &ids[i]);
    }

    // 等待所有线程结束
    for (i = 0; i < N; i++) {
        pthread_join(thread_ids[i], NULL);
    }

    // 销毁筷子
    for (i = 0; i < N; i++) {
        pthread_mutex_destroy(&chopsticks[i]);
    }

    return 0;
}
```

【简单粗暴，直接使用一把互斥锁】

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

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

#define N 5  // 哲学家数量

pthread_mutex_t chopsticks[N];   // 筷子数组，每个元素对应一个筷子
pthread_mutex_t extra_lock;       // 额外的锁

void *philosopher(void *arg) {
    int i = *(int *)arg;
    while (1) {
        printf("Philosopher %d is thinking\n", i);
        // sleep(rand() % 3);  // 思考一段时间

        printf("Philosopher %d is hungry\n", i);
        pthread_mutex_lock(&extra_lock);         // 获取额外的锁
        pthread_mutex_lock(&chopsticks[i]);      // 获取左边的筷子
        printf("Philosopher %d picked up left chopstick\n", i);
        pthread_mutex_lock(&chopsticks[(i + 1) % N]);  // 获取右边的筷子
        printf("Philosopher %d picked up right chopstick\n", i);

        printf("Philosopher %d is eating\n", i);
        // sleep(rand() % 3);  // 进餐一段时间

        pthread_mutex_unlock(&chopsticks[(i + 1) % N]);  // 放下右边的筷子
        printf("Philosopher %d put down right chopstick\n", i);
        pthread_mutex_unlock(&chopsticks[i]);            // 放下左边的筷子
        printf("Philosopher %d put down left chopstick\n", i);
        pthread_mutex_unlock(&extra_lock);               // 释放额外的锁
    }
}

int main() {
    pthread_t thread_ids[N];
    int ids[N];
    int i;

    // 初始化筷子
    for (i = 0; i < N; i++) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    // 初始化额外的锁
    pthread_mutex_init(&extra_lock, NULL);

    // 创建哲学家线程
    for (i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&thread_ids[i], NULL, philosopher, &ids[i]);
    }

    // 等待所有线程结束
    for (i = 0; i < N; i++) {
        pthread_join(thread_ids[i], NULL);
    }

    // 销毁筷子和额外的锁
    for (i = 0; i < N; i++) {
        pthread_mutex_destroy(&chopsticks[i]);
    }
    pthread_mutex_destroy(&extra_lock);
```

```
    return 0;
}
```

读写者问题

为了保证并发的正确性，我们需要遵循以下原则：

1. 写者优先：当有等待的写者时，读者必须等待，直到所有等待的写者都完成写操作。
2. 避免饥饿：所有线程必须有机会访问共享资源，不能因为等待时间过长而一直等待。
3. 避免死锁：当多个线程同时等待对方释放锁时，可能会出现死锁的情况。因此，我们需要避免这种情况的发生。

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

#define NUM_READERS 5  // 读者数量
#define NUM_WRITERS 3  // 写者数量
#define NUM_READS 5    // 每个读者读取的次数
#define NUM_WRITES 3   // 每个写者写入的次数

pthread_mutex_t mutex;       // 互斥锁，用于保护共享资源
pthread_cond_t cond_read;    // 条件变量，用于控制读者的访问
pthread_cond_t cond_write;   // 条件变量，用于控制写者的访问

int shared_data = 0;         // 共享资源

int num_readers_active = 0;  // 当前活跃的读者数量
int num_writers_waiting = 0; // 当前等待的写者数量

void *reader(void *arg) {
    int id = *(int *)arg;
    int i;
    for (i = 0; i < NUM_READS; i++) {
        pthread_mutex_lock(&mutex);  // 获取互斥锁
        while (num_writers_waiting > 0) {  // 如果有等待的写者，则等待
            pthread_cond_wait(&cond_read, &mutex);
        }
        num_readers_active++;  // 当前活跃的读者数量加1
        pthread_mutex_unlock(&mutex);  // 释放互斥锁

        // 读取共享资源
        printf("Reader %d read shared_data = %d\n", id, shared_data);
        sleep(rand() % 3);

        pthread_mutex_lock(&mutex);  // 获取互斥锁
        num_readers_active--;  // 当前活跃的读者数量减1
        if (num_readers_active == 0) {  // 如果没有读者了，则唤醒等待的写者
            pthread_cond_signal(&cond_write);
        }
        pthread_mutex_unlock(&mutex);  // 释放互斥锁
    }
    pthread_exit(NULL);
}
```

```c
void *writer(void *arg) {
    int id = *(int *)arg;
    int i;
    for (i = 0; i < NUM_WRITES; i++) {
        pthread_mutex_lock(&mutex);  // 获取互斥锁
        num_writers_waiting++;  // 当前等待的写者数量加1
        while (num_readers_active > 0) {  // 如果有活跃的读者，则等待
            pthread_cond_wait(&cond_write, &mutex);
        }
        num_writers_waiting--;  // 当前等待的写者数量减1
        // 写入共享资源
        printf("Writer %d writes shared_data = %d\n", id, ++shared_data);
        sleep(rand() % 3);
        pthread_cond_signal(&cond_read);  // 唤醒等待的读者
        pthread_cond_signal(&cond_write);  // 唤醒等待的写者
        pthread_mutex_unlock(&mutex);  // 释放互斥锁
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t reader_ids[NUM_READERS];
    pthread_t writer_ids[NUM_WRITERS];
    int reader_args[NUM_READERS];
    int writer_args[NUM_WRITERS];
    int i;

    pthread_mutex_init(&mutex, NULL);  // 初始化互斥锁
    pthread_cond_init(&cond_read, NULL);  // 初始化条件变量
    pthread_cond_init(&cond_write, NULL);

    // 创建读者线程
    for (i = 0; i < NUM_READERS; i++) {
        reader_args[i] = i;
       pthread_create(&reader_ids[i], NULL, reader, &reader_args[i]);
    }

    // 创建写者线程
    for (i = 0; i < NUM_WRITERS; i++) {
        writer_args[i] = i;
        pthread_create(&writer_ids[i], NULL, writer, &writer_args[i]);
    }

    // 等待所有线程结束
    for (i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_ids[i], NULL);
    }
    for (i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writer_ids[i], NULL);
    }

    pthread_mutex_destroy(&mutex);  // 销毁互斥锁
    pthread_cond_destroy(&cond_read);  // 销毁条件变量
    pthread_cond_destroy(&cond_write);

    return 0;
}
```

【信号量实现】

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

```c
#include <pthread.h>
#include <semaphore.h>

#define MAX_READERS 5
#define MAX_WRITERS 3

int shared_variable = 0;  // 共享变量
int reader_count = 0;  // 当前正在访问共享变量的读者数量

// 信号量
sem_t mutex;  // 互斥信号量，用于控制对共享变量的互斥访问
sem_t rw_mutex;  // 读写信号量，用于控制读者和写者的访问顺序

void *reader(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // 等待对共享变量的互斥访问
        sem_wait(&mutex);
        reader_count++;
        if (reader_count == 1) {
            // 第一个读者需要等待对读写者的访问顺序
            sem_wait(&rw_mutex);
        }
        // 释放对共享变量的互斥访问
        sem_post(&mutex);

        // 读取共享变量
        printf("Reader %d reads shared variable: %d\n", id, shared_variable);

        // 等待对共享变量的互斥访问
        sem_wait(&mutex);
        reader_count--;
        if (reader_count == 0) {
            // 最后一个读者需要释放对读写者的访问顺序
            sem_post(&rw_mutex);
        }
        // 释放对共享变量的互斥访问
        sem_post(&mutex);

        // 等待一段时间再进行下一次访问
        sleep(rand() % 5);
    }
}

void *writer(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // 等待对读写者的访问顺序
        sem_wait(&rw_mutex);

        // 修改共享变量
        shared_variable++;
        printf("Writer %d writes shared variable: %d\n", id, shared_variable);

        // 释放对读写者的访问顺序
        sem_post(&rw_mutex);

        // 等待一段时间再进行下一次访问
        sleep(rand() % 5);
    }
}
```

```
int main() {
    // 初始化信号量
    sem_init(&mutex, 0, 1);
    sem_init(&rw_mutex, 0, 1);

    // 创建多个读者和写者线程
    pthread_t readers[MAX_READERS], writers[MAX_WRITERS];
    int i, id[MAX_READERS + MAX_WRITERS];
    for (i = 0; i < MAX_READERS; i++) {
        id[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &id[i]);
    }
    for (i = 0; i < MAX_WRITERS; i++) {
        id[MAX_READERS + i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &id[MAX_READERS + i]);
    }

    // 等待所有线程结束
    for (i = 0; i < MAX_READERS; i++) {
        pthread_join(readers[i], NULL);
    }
    for (i = 0; i < MAX_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    // 销毁信号量
    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);

    return 0;
}
```

OpenMP

计算π的值(5种方法)

积分累加法

1. 串行

```
long n = 100000;
int main() {
    double x, sum = 0;
    double h = 1.0/n;
    // 把0-1的区间分为n分，每小份的坐标取 (i + 0.5) * h;
    for (int i = 0; i < n; ++i) {
        x = (i + 0.5) * h;
        sum += 4.0/(1.0+x*x);
    }
    double pi = sum * h;
    return 0;
}
```

2. 并行1：手动切片

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

```
#define NUM 8

long n = 1000000;

int main() {
    // 0. 初始化结果数组，避免互斥
    double pi, sum[NUM] = {0.0};
    double h = 1.0 / n;
    // 1. 设置进程数量
    omp_set_num_threads(NUM);
    // 2. 并行化
    #pragma omp parallel
    {
        double x;
        // 3. 获得进程id
        int tid = omp_get_thread_num();
        // 4. 切片数据
        for (int i = tid; i < n; i += NUM) {
            x = (i + 0.5) * h;
            sum[tid] += 4.0 / (1.0 + x * x);
        }
    }
    // 5. 累加各个线程的结果
    for (int i = 0; i < NUM; i++) {
        pi += sum[i] * h;
    }
    printf("π：%f\n", pi);
    return 0;
}
```

2. 并行化2：使用for自动切片

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

#define NUM 8

long n = 1000000;

int main() {
    // 0. 初始化结果数组，避免互斥
    double pi, sum[NUM];
    double h = 1.0/n;
    // 1. 设置进程数量
    omp_set_num_threads(NUM);
    // 2. 并行化
    #pragma omp parallel
    {
        double x;
        // 3. 获得进程id
        int tid = omp_get_thread_num();
        // 4. 自动切片
        #pragma omp for
        for (int i = 0; i < n; i += 1) {
            x = (i + 0.5) * h;
            sum[tid] += 4.0/(1.0+x*x);
        }
    }
    for (int i = 0; i < NUM; i++) {
        pi += sum[i] * h;
    }
```

```
    return 0;
}
```

3. 并行3：使用private复用外部变量

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

#define NUM 8

long n = 1000000;

int main() {
    double pi, sum[NUM] = {0.0};
    double h = 1.0/n;
    double x;

    omp_set_num_threads(NUM);

    #pragma omp parallel private(x)
    {
        int tid = omp_get_thread_num();
        for (int i = tid; i < n; i += NUM) {
            x = (i + 0.5) * h;
            sum[tid] += 4.0/(1.0+x*x);
        }
    }

    for (int i = 0; i < NUM; i++) {
        pi += sum[i] * h;
    }

    printf("Pi: %f\n", pi);
    return 0;
}
```

4. 并行化4：使用for+reduction实现互斥

使用reduction后遍量自动变为私有的，不能再手动声明相同的遍量为private类型【注意使用for时不能用大括号】

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

#define NUM 8

long n = 1000000;

int main() {
    double pi, sum = 0.0;
    double h = 1.0/n;
    double x;

    omp_set_num_threads(NUM);

    #pragma omp parallel for reduction(+:sum) private(x)
    for (int i = 0; i < n; i += 1) {
        x = (i + 0.5) * h;
        sum += 4.0/(1.0+x*x);
    }
```

```
    pi = sum * h;

    printf("Pi: %f\n", pi);

    return 0;
}
```

5. 使用同步critical【任意代码块】，不能使用for

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

#define NUM 8

long n = 1000000;

int main() {
    double pi = 0.0;
    double h = 1.0/n;
    double x;

    omp_set_num_threads(NUM);

    #pragma omp parallel private(x)
    {
        double sum = 0.0;
        int tid = omp_get_thread_num();
        for (int i = tid; i < n; i += NUM) {
            x = (i + 0.5) * h;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum * h;
    }

    printf("Pi: %f\n", pi);
    return 0;
}
```

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

#define NUM 8

long n = 1000000;

int main() {
    double pi = 0.0;
    double h = 1.0/n;
    double x;

    omp_set_num_threads(NUM);

    #pragma omp parallel for private(x)
    {
        double sum = 0.0;
        for (int i = 0; i < n; i += 1) {
            x = (i + 0.5) * h;
```

```
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum * h;
    }

    printf("Pi: %f\n", pi);
    return 0;
}
```

蒙特卡罗法

【pthread和MPI都需要先计算每个线程局部的num，然后再归约累加。而openMP可以直接用reduction计算全局num，其自动私有化并归约】

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

#define MAX_NUM 10000000
#define TNUM 8

int main() {
    // 设置线程数
    omp_set_num_threads(TNUM);

    double x,y,dist;
    int total_num = 0;
    #pragma omp parallel for reduction(+:total_num) private(x,y,dist)
    for (int i = 0; i < MAX_NUM; i++) {
        x = (double)rand()/(double)RAND_MAX;
        y = (double)rand()/(double)RAND_MAX;
        dist = x*x + y*y;
        if (dist <= 1) {
            total_num++;
        }
    }

    double pi = 4 * (double)total_num/MAX_NUM;
    printf("pi: %f\n", pi);
    return 0;
}
```

计算ln2

【不能使用自动划分，因为需要划分后判断第一个元素的奇偶性】

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

#define NUM 8

// 数据量
int n = 1000000;

int main() {
    // 初始化
```

```
    omp_set_num_threads(NUM);
    // 结果
    double ans = 0.0;
    #pragma omp parallel reduction(+:ans)
    {
        int rank = omp_get_thread_num();
        // 单个进程的数据量
        int chunk = n/NUM;
        // 划分数据集
        int start = rank*chunk;
        int end = start + chunk;

        if (rank == NUM-1) {
            end = n;
        }
        double factor;
        if (start % 2 == 0) {
            factor = 1.0;
        } else {
            factor = -1.0;
        }
        for (int i = start; i < end; ++i) {
            ans += factor/(i + 1);
            factor = -factor;
        }
    }
    printf("ans:  %f\n",ans);
    return 0;
}
```

矩阵相乘

```
vector<vector<long long>> matrix_multiply_OMP(const vector<vector<long long>>& A,
const vector<vector<long long>>& B, int tnum) {
    omp_set_num_threads(tnum);
    uint64_t n = A.size();
    vector<vector<long long>> c(A_x, vector<long long>(B_y, 0));
    #pragma omp parallel for collapse(2)
    for (long long i = 0; i < n; i++) { // For each raw of A
        for (long long j = 0; j < n; j++) { // For each Colum of B
            long long sum = 0;
            #pragma omp simd reduction(+:sum)
            for (long long k = 0; k < n; k++) { // multiple n times
                sum += A[i][k] * B[k][j];
            }
            c[i][j] = sum;
        }
    }
    return move(c);
}
```

连续素数

只考虑两个连续奇数且素数【reduction】

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

```c
#include <stdbool.h>
#include <omp.h>

#define NUM 8

bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

int main() {
    int n = 1000000;
    int chunk_size = n / NUM;
    int count = 0;
    omp_set_num_threads(NUM);
    #pragma omp parallel reduction(+:count)
    {
        int tid = omp_get_thread_num();
        int start = tid * chunk_size;
        int end = (tid + 1) * chunk_size;
        if (tid == NUM - 1) {
            end = n;
        }
        for (int i = start + 1; i <= end - 1; i += 2) {
            if (is_prime(i) && is_prime(i + 2)) {
                count++;
            }
        }
    }
    printf("在小于 %d 的整数范围内，连续奇数都是素数的情况的次数为: %d\n", n, count);
    return 0;
}
```

【critical同步】

```c
#include <stdio.h>
#include <stdbool.h>
#include <omp.h>

#define NUM 8

bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

int main() {
    int n = 1000000;
```

```
    int chunk_size = n / NUM;
    int count = 0;
    omp_set_num_threads(NUM);
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int start = tid * chunk_size;
        int end = (tid + 1) * chunk_size;
        for (int i = start + 1; i <= end - 1; i += 2) {
            if (is_prime(i) && is_prime(i + 2)) {
                #pragma omp critical
                count++;
            }
        }
    }
    printf("在小于 %d 的整数范围内，连续奇数都是素数的情况的次数为: %d\n", n, count);
    return 0;
}
```

考虑任意长度连续奇数且素数

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

#define MAX_NUM 1000000
#define TNUM 8
bool is_prime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}


int main() {
    int count = 0;a
    int num_primes_in_a_row = 0;
    int chunk_size = MAX_NUM / TNUM;
    omp_set_num_threads(TNUM);
    #pragma omp parallel reduction(+:count) private(num_primes_in_a_row)
    {
        int tid = omp_get_thread_num();
        int start = tid * chunk_size;
        int end = (tid + 1) * chunk_size;
        for (int i = start + 1; i <= end - 1; i += 2) {
            if (is_prime(i)) {
                num_primes_in_a_row++;
            } else {
                 if (num_primes_in_a_row >= 2) {
                    count++;
                }
                num_primes_in_a_row = 0;
            }
```

```
        }
    }
    printf("连续奇数都是素数的情况的次数为: %d\n", count);
    return 0;
}
```

最大连续素数间隔

```

```

水仙花数

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

#define NUM 8

double mpow(double x, int n) {
    if (n == 0) {
        return 1.0; // 任何数的 0 次方等于 1
    }
    if (n < 0) {
        n = -n;
        x = 1 / x; // 如果 n 是负数，将 x 变为其倒数，并将 n 变为其相反数
    }
    double ans = 1.0;
    while (n) {
        if (n % 2 == 1) {
            ans *= x; // 如果 n 是奇数，将 x 乘到答案中
        }
        x *= x; // 将 x 的平方赋值给 x
        n /= 2; // 将 n 除以 2
    }
    return ans;
}

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += mpow(digit, digits);
        temp /= 10;
    }
    return sum == n;
```

```
}
int main() {
    int count = 0;
    int ss = 0;
    omp_set_num_threads(NUM);
    for (int n = 3; n <= 24; n++) {
        int min = mpow(10, n-1);
        int max = mpow(10, n);
        #pragma omp parallel for reduction(+:count)
        for (int i = min; i < max; i++) {
            if (is_narcissistic(i)) {
                {
                    printf(" %d ", i);
                    count++;
                }
            }
        }
    }

    printf("Count: %d\n", count);
    return 0;
}
```

【原子】

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

#define NUM 8

double mpow(double x, int n) {
    if (n == 0) {
        return 1.0; // 任何数的 0 次方等于 1
    }
    if (n < 0) {
        n = -n;
        x = 1 / x; // 如果 n 是负数，将 x 变为其倒数，并将 n 变为其相反数
    }
    double ans = 1.0;
    while (n) {
        if (n % 2 == 1) {
            ans *= x; // 如果 n 是奇数，将 x 乘到答案中
        }
        x *= x; // 将 x 的平方赋值给 x
        n /= 2; // 将 n 除以 2
    }
    return ans;
}

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
```

```
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += mpow(digit, digits);
        temp /= 10;
    }
    return sum == n;
}

int main() {
    int count = 0;
    omp_set_num_threads(NUM);
    for (int n = 3; n <= 4; n++) {
        int min = mpow(10, n-1);
        int max = mpow(10, n);
        #pragma omp parallel
        {
            int local_count = 0;
            #pragma omp for
            for (int i = min; i < max; i++) {
                if (is_narcissistic(i)) {
                    #pragma omp critical
                    {
                        printf(" %d ", i);
                        local_count++;
                    }
                }
            }
            #pragma omp atomic
            count += local_count;
        }
        printf("\n");
    }
    printf("Count: %d\n", count);
    return 0;
}
```

【直接把数据展开，然后均分分割即可】

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

#define NUM 8

double mpow(double x, int n) {
    if (n == 0) {
        return 1.0; // 任何数的 0 次方等于 1
    }
    if (n < 0) {
        n = -n;
        x = 1 / x; // 如果 n 是负数，将 x 变为其倒数，并将 n 变为其相反数
    }
    double ans = 1.0;
    while (n) {
        if (n % 2 == 1) {
            ans *= x; // 如果 n 是奇数，将 x 乘到答案中
        }
        x *= x; // 将 x 的平方赋值给 x
        n /= 2; // 将 n 除以 2
```

```
    }
    return ans;
}

// 计算一个数的位数
int get_digits(int n) {
    int digits = 0;
    while (n) {
        digits++;
        n /= 10;
    }
    return digits;
}

// 判断一个数是否是水仙花数
bool is_narcissistic(int n) {
    int digits = get_digits(n);
    int sum = 0;
    int temp = n;
    while (temp) {
        int digit = temp % 10;
        sum += mpow(digit, digits);
        temp /= 10;
    }
    return sum == n;
}
int main() {
    int count = 0;
    int ss = 0;
    omp_set_num_threads(NUM);
    int min = mpow(10,2);
    int max = mpow(10,25);

    #pragma omp parallel for reduction(+:count)
    for (int i = min; i < max; i++) {
        if (is_narcissistic(i))
        {
            printf(" %d ", i);
            count++;
        }
    }


    printf("Count: %d\n", count);
    return 0;
}
```

完全数

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int is_perfect_number(int n) {
    int sum = 0;
    for (int i = 1; i < n; i++) {
        if (n % i == 0) {
            sum += i;
        }
```

```
    }
    return sum == n;
}

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n_min = 6;
    int n_max = 33550337;

    // 平均分配n的范围
    int n_start = n_min + rank * ((n_max - n_min) / size);
    int n_end = n_min + (rank + 1) * ((n_max - n_min) / size);
    if (rank == size - 1) {
        n_end = n_max;
    }

    // 计算每个进程分配的范围内的完全数并输出
    int count = 0;
    for (int n = n_start; n <= n_end; n++) {
        if (is_perfect_number(n)) {
            printf("%d ", n);
            count++;
            if (count == 8) {
                break;
            }
        }
    }
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

不需要求连续序列，可以离散化分片：

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int is_perfect_number(int n) {
    int sum = 0;
    for (int i = 1; i < n; i++) {
        if (n % i == 0) {
            sum += i;
        }
    }
    return sum == n;
}

int main() {

    int n_max = 33550337;
    int count = 0;
    #pragma omp parallel for reduction(+:count)
    for (int n = 6; n < n_max; n++) {
        if (is_perfect_number(n)) {
            printf("%d ", n);
            count++;
```

```
            if (count == 8) {
                break;
            }
        }
    }
    printf("\n");

    return 0;
}
```