# 并行计算

2022 年 5 月 25 日

# 目录

# 1 矩阵乘并行计算

## 1.1 矩阵基本性质

### 1.1.1 加法

可交换顺序、可分配可结合、A+(-A)=O

### 1.1.2 数乘

可交换顺序、可分配可结合。

### 1.1.3 乘法

不可交换顺序、可分配可结合。

### 1.1.4 转置

$(A+B)^T = A^T + B^T$
$(kA)^T = kA^T$
$(AB)^T = B^T A^T$
$(AT)^T = A$

### 1.1.5 共轭

$(A)_{i,j} = \overline{A_{i,j}}$
矩阵内实部不变，虚部取负。

### 1.1.6 注意

$(A+B)(A+B) = A^2 + AB + BA + B^2$

### 1.1.7 相关定义

- 行列式：是一个函数，其定义域为 det 的矩阵 A，取值为一个标量，写作 det(A) 或 |A|，A 为 n×n 的正方形矩阵。
- 余子式：n 阶行列式 D 中，把元素 $a_{oe}$ 所在的第 o 行和第 e 列划去后，留下来的 n-1 阶行列式叫做元素 $a_{oe}$ 的余子式，记作 $M_{oe}$。

k 阶余子式：行列式 D 中划去了 k 行 k 列，划去的交叉部分组成子式 A（即元素），称剩下的为行列式 D 的 k 阶子式 A 的余子式。

- 代数余子式：将余子式 $M_{oe}$ 再乘以-1 的 o+e 次幂记为 $A_{oe}$，$A_{oe}$ 叫做元素 $a_{oe}$ 的代数余子式。

  同理 k 阶代数余子式。此时 o、e 为行和列的序号累加。

- 特征值：对于 n 阶方阵 A，如果存在数 m 和非零 n 维列向量 x，使得 Ax=mx 成立，则称 m 是 A 的一个特征值（characteristic value）或本征值（eigenvalue）。

- 特征向量：对于一个给定的线性变换，它的特征向量（本征向量或称正规正交向量）v 满足经过该线性变换之后，得到的新向量仍然与原来的 v 保持在同一条直线上。

- 迹：n 阶方阵 A 的对角元素之和称为矩阵 A 的迹 (trace), 记作 tr(A)。

- 正定矩阵：n 阶方阵 A，如果对任何非零向量 z，都有 $z^TAz > 0$，其中 $z^T$ 表示 z 的转置，就称 M 为正定矩阵。大于等于 0 则为半正定矩阵，小于 0 则为负定矩阵。

#### 1.1.8 初等变换

初等变换：设 A 是 m×n 矩阵，进行倍乘、互换、倍加行（列）变换，统称为初等变换。包括：

- 倍乘：用非零常数 k 乘 A 的**某行 (列)** 的每个元素。

- 互换：互换 A 的某两行（列）的位置。

- 倍加行（列）：将 A 的某行（列）元素的 k 倍加到另一行（列）。

初等矩阵：单位矩阵经一**次**初等变换得到的矩阵称为初等矩阵。

等价矩阵：矩阵 A 经过有限次初等变换变成矩阵 B，则称 A 与 B 等价（可能有多个矩阵与 A 等价，其中等价的最简矩阵被称为 A 的等价标准型）

性质：用初等矩阵 P 左乘（右乘）A，其结果 PA(AP) 相当于对 A 作相应的初等行（列）变换。

### 1.2 传分块统方法

对于矩阵 a 被分成 2×2 四块的情况：

$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}$$

有：$c_{0,0} = a_{0,0} * b_{0,0} + a_{0,1} * b_{1,0}$

该情况下，每个线程（计算块）中都存储一行 A 和一列 B（矩阵块），（又要传递又要储存）大大增加了存储量，存储量由 O（n 平方）—>O(n 立方)

### 1.3 Cannon 方法

该算法在每次计算完成后让计算块内的数据有规律的传递移动，记为 M×M 矩阵，有：

1. A 总体上子块从右往左循环移动 1 步；$a_{i,j} -> a_{i,j-1}$
2. B 总体上子块从下往上循环移动 1 步；$b_{i,j} -> b_{i-1,j}$
3. 每个计算块正常相乘，并存入 C 块中，$c_{i,j} = c_{i,j} + c$；
4. 重复上述过程，累计 M 次；

最终得到相乘的结果。

## 2 线性方程组的并行求解

### 2.1 直接求解法

#### 2.1.1 LU 分解算法

对于矩阵形式的线性方程组：Ax=b，如果 A 满足为方阵且可逆，则 A 可以被分解为下三角矩阵 L(Lower Triangle Matrix) 和上三角矩阵 U(Uppder Triangle Matrix) 的乘积。即：

PA = LU

得：LUx=Pb

则方程可以被分解为 (L)y=(Pb) 和 (U)x=(y)，通过两个相似的步骤依次求解 y 和 x 即可。

基本原理是利用高斯消元，原理是在求解方程组 Ax=b 时将系数矩阵 A 和右向量 b 组成增广矩阵，对其进行行的初等变换，最终得到上三角初等矩阵，则自下而上可求得各个 x。

在不带入 b 的情况下，对 A 矩阵的初等变换操作可以以置换矩阵 $E_{ij}$ 来表示，表示交换第 i 行和第 j 行。而此矩阵乘以一个置换矩阵 P 即可化为下三角形式。

**置换矩阵 P：**

对于单位矩阵 E，交换其内部的行列，即可得到置换矩阵，与矩阵相乘时，对单位矩阵的操作（交换、乘加、乘（但仅交换属于置换矩阵））都会作用到相乘的矩阵上。

### 2.1.2 Gauss 直接消去

对矩阵：$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & & & \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$

- 从上往下，第二行减去乘以系数 $a_{2,1}/a_{1,1}$ 的第一行：

  $a_{2,k} = a_{2,k} - a_{1,k} * a_{2,1}/a_{1,1}$

  更新第二行的值 $a_{2,k}$。

- 第三行减去乘以系数 $a_{3,1}/a_{1,1}$ 的第一行，减去乘以系数 $a_{3,2}/a_{2,2}$ 的第二行：

  $a_{3,k} = a_{3,k} - a_{1,k} * a_{2,1}/a_{1,1}$

  $a_{3,k} = a_{3,k} - a_{2,k} * a_{3,2}/a_{2,2}$

- 对于第 i 行的处理，第 j 列元素有：

  $a_{i,j} = a_{i,j} - \sum_{k=1}^{k<i} (a_{k,j} * a_{i,k}/a_{k,k})$

  第 k 行乘的因子始终为：$a_{i,k}/a_{k,k}$

- 最终得到上三角初等矩阵 U。L 和 P 通过反推变换过程可以得到。

- 对向量 b 做相同变换，则可从下而上，逐渐求解出各个未知数。

**列主元 Gauss 消去方法：**

- 类似于直接方法，但避免了 $a_{j,j}$ 为 0 时无法消元的情况。将从第 j 列的 $a_{j,j}$ 及其以下的各元素中选取绝对值最大的元素，然后通过行变换将它交换到主元素 $a_{j,j}$ 的位置上，再进行消元。
- 对于第 j 列，首先找到该列中最大值的所在行 l($l \geq j$ 即可)，记最大值为 $a_{l,j}$。
- 然后将第 j 列第 j 行即主元 $a_{j,j}$ 的值，与 $a_{l,j}$ 进行对比：

  如果 $a_{l,j}$ 等于 0 就直接退出（该矩阵无法求解）；

  如果不相等，则将第 j 行与第 l 行进行交换，使第 j 列最大值在主元位置：$swap(a_{l,j}, a_{j,j})$。
- 对于$i > j$的每一行 i(k 表示列)，执行操作 $a_{ik} = a_{ik} - a_{ik} \times a_{ij}/a_{jj}$，注意 j 始终指当前的第 j 列。
- 然后依次处理$0 \leq j < n$各列。

### 2.1.3 Gauss 消去并行计算方法

- 并行部分以列为块分割，出于负载均匀的考虑，进程间交叉存在，例如第 1 列到第 6 列，线程序号按 1、2、3、1、2、3 排列。
- 进程间通信为若干个只从一个进程向下一个进程传递的因子组 $f_{i-j} = a_{j,i}/a_{i,i}, k < j < n-1$(表示第 j 行用的（减去）第 i 行乘的因子)；以及为了定位行的交换，还需要传递最大值所在行 $l$。
- 对列进行循环，对于落在某线程内的每一列，可以按序号 k 划分为 2 个部分 (其中 1<k<=i)，其会按顺序依次经过：
  - k<=i：则此部分共计接收之前线程的 i 个 send，表示了前 i 行对第 i 行 (从 0 行开始) 的操作 (乘加因子、交换行)($f_{j-i}, 0 <= j < i$,$l$)，并且将获得的发送给下一个线程 (i 个 send)。(**注意每次 i 循环只接受一个**)
  - k=i：计算此列对应的行 i 相对于接下来的行 j 的操作 ($f_{i-j}, i < j < n$,$l$)，进行本列内的行交换，并传递给下一个线程 (1 个 send)。

– 注意对于 k>i 的情况，可以算到进程执行到下一轮 i 中 k<=i 的情况。

注意，最后一个线程会向第一个线程发送之前已获得全部的数据，但不会发送紧接着的下一个线程的数据，因为这个数据就是从下一个线程传来的，避免重复发送 (这也是重要的终止 send-recv 条件)。

- 在循环内，按顺序进行行交换 (行整体)，由于每一个线程内，必然是先到 k<i 的情况，因此不用担心之前行的交换对接下来 k=i 情况下交换的影响；注意本线程内交换过的数不用重复交换。
- 在循环内，利用 $f_{i-j}, 0 < j < i$，计算 $a_{j,k}, i < j < n-1, i < k < n-1k$，非最终值，不断更新。**每次更新 (j,j) 以下的最大矩形块的数据，依据的是一个 f 因子组**。即用计算的行的 f 或者传递过来的 f 来更新全部的系数。
- 注意：**最终每一个线程内所得的矩阵都是分解后的上三角矩阵，也就是说每个线程在每个 i 下交换操作和更新系数操作是完全相同的**。注意线程是完全同步的，当 i 等于 1 时，所有的进程都在执行循环 i=1。

**三角矩阵的并行求解：**

以下三角矩阵为例：步骤：

$$\begin{bmatrix} a_{1,1} & 0 & 0 & ... & 0 \\ a_{2,1} & a_{2,2} & 0 & ... & 0 \\ a_{3,1} & a_{2,3} & a_{3,3} & ... & 0 \\ ... & & & & \\ a_{m,1} & a_{m,2} & a_{m,3} & ... & a_{m,n} \end{bmatrix}$$

- 并行的部分为列，但出于处理器负载均衡的考虑，每一个并行块并不是相邻的若干列，而是分开的，类似于 123，123，123 这样的交替排列，每一组大小为 P 列 (等于并行的数量)，称为卷帘。
- 并行部分原理：从 k=0 列开始：
- 如果属于第一个线程，就使 $u_i = b_i$(行 $i \subseteq [0, n-1]$)，并使 $v_i = 0$(行 $i \subseteq [0, p-2]$)。

  否则使 $u_i = 0$，行范围同上。

  (即只在第一个线程处对 u 赋右值、对 v 赋初值 0)

- 判断属于第 myid 线程，i 从 myid 开始增加，对每一个线程内的第 i 行 (0 到 p-1)，i 以 p 为步长递增，直到最后 n:

  $for \quad i = myid \quad step \quad p \quad to \quad n-1$

- 在 i=0 的情况下，不接受数据；否则接收 n 维向量 $v_{recv}$。
- 计算 $x_k = (u_i + v_0)/a_{ik}$。

  **整个 0 到 p-1 只有这一行即 k 行的 x 是求的，剩下的由接下来的线程求**。

- 更新传入的 v 向量:

  $v_j = v_{j+1} + u_{i+j+1} - a_{i+j+1} * x_k$ , j=0,...,p-3

  $v_{p-2} = u_{i+p-1} - a_{i+p-1} * x_k$ , 即 j=p-2

  **j 在这里指的是每一个 i 下，对应的行数 (范围 0 到 p-1)，随 i 的变化 v 向量不断更新**。

- 向下一线程发送 v 向量 $v_{send}$。
- 更新 [i+p,u-1] 行范围内的 u 向量:

  $u_j = u_j - a_{jk} * x_k$ , j=i+p,...,n-1

- k=K+1。
- 继续循环 i，最终完成全部求解。每次循环只处理一列!!

**u 数组与 v 数组的理解**

- v 数组在线程间传递，用于将本线程的计算结果对接下来的方程的影响 (本次 + 之前所有的) 传递给其他线程。**也就是说是其他线程的解对求解方程的影响**。
- v 数组的大小等于线程总数目-1；每次循环时都会变化，覆盖的列向未知数区域移动，每次移动一行。即每次由本向量结果新得到一个 v，舍弃一个 v，剩下的 v 向下平移并考虑本线程的影响。
- 注意：在 v 的更新中，更新后的 $v_0$ 指的是当前的 k 对应的 k+1 行。而最新产生的 v 并不会考虑其他线程的影响，则在下次循环到本线程

时，本线程的影响就会消失，因此在下一次循环到本线程时，计算 x 时需要考虑本线程的影响即 u。

(没有必要每次传完所有的影响 $a_{i,j} * x_i$，浪费性能)

- u 在本线程内使用，用于本地保存本线程已计算出的解 ($x_0$、...、$x_k$) 所对应的 $b_i - \sum_{i=0}^{i=k} x_i * a_{i,j}$，也就是本线程的解对求解方程的影响。
- u 在初始情况下即第一个线程时被初始赋值为 b，即为方程右值。

## 2.2 迭代解法

可以对每一行的方程迭代部分进行划分，每一个计算块处理若干行的迭代方程。

Gauss-Seidel 迭代方法：Ax=b，s 为迭代的 x 带入得到的过程量 ax。

- 行：类似于 LU 分解，每一个迭代区间内，对行进行循环：
  - 在非本线程的行下，每个线程接受其前一个线程的 x，并且向下一个线程发送 (除非一个恰好循环)；
  - 在本线程的行下，计算出 x 的值，并且向下一个线程发送。

  之后对 s 矩阵进行更新，第 i 行即更新了第 i 行的 s，并且每个线程内保有的 s 矩阵在每次行循环下保持一致。

  最终并行计算截至条件，如果满足就跳出循环。
- 列：类似于求解三角矩阵的方程组，每一个迭代区间内，对列 j 进行循环：
  - 在非本线程的行下，每个线程接受其前一个线程的 x，并且向下一个线程发送 (除非一个恰好循环)；
  - 在本线程的行下，计算出 x 的值，并且向下一个线程发送。

  之后对 s 矩阵进行更新，但注意的是第 i 列下的更新的是全部的 s 向量的第 j 列，而非一整个第 j 行。若以 j 表示列，则第 i 行 $s_i = b_i - \sum_{j=1}^{n}(a_{i,j} * x[j])$ 中，未更新到的 x 对应的部分仍然采用的是上一轮循环的 $x[j]$。

  最终并行计算截至条件，如果满足就跳出循环。

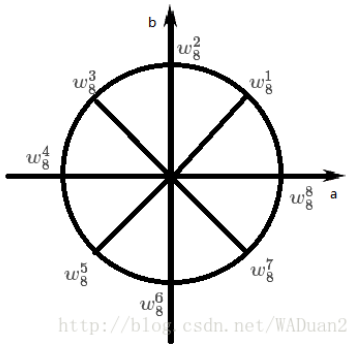
# 3 FFT 并行算法

## 3.1 复数基本知识

- 复数乘法的在复平面中表现为辐角相加，模长相乘；

  即 $(a_1, \theta_1) * (a_2, \theta_2) = (a_1 * a_2, \theta_1 + \theta_2)$

- 单位根：复数 w 满足 $w^n = 1$，称为 n 次单位根。如图所示：





总 n 次第 m 个根记为 $w_n^m$，其中 n 为 2 的整数倍，则满足性质：

$w_n^m = -w_n^{m+n/2}$

## 3.2 快速傅氏变换 FFT 原理

### 3.2.1 物理意义

**傅里叶变换：**

对于周期函数，是将 $f(t)$ 分解为无数个**不同频率**、**不同幅值**的正、余弦信号。用频谱函数表示，自变量是频率 $\omega$，因变量是幅值。函数是离散的，自变量都是基频 $\omega_0$ 的整数倍。

对于非周期函数，则是求频谱密度函数，自变量是 $\omega$ ，因变量是信号幅值在频域中的分布密度，即单位频率信号的强度。

**可以将频谱函数和频谱密度函数类比为离散概率分布和概率密度函数。**

**快速傅氏变换：**

是离散傅氏变换的快速算法，是对离散傅立叶变换的改进。可用于加速多项式的乘法，将复杂度从 $\Theta(n^2)$ 优化为 $\Theta(n \log n)$。

### 3.2.2 DFT

对于连续的傅里叶变换，已知：

$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt$

DFT 的目的是得到信号的频谱密度函数 $(t- > \omega)$，DFT 就是 $\omega$ 和 t 都为离散版的傅里叶变换。

由于计算机也只可能计算出有限个频率上对应的幅值密度，因此最终也需要转为离散的情况。

**基本原理：**

- 时域 $t$ 的离散化：

  由于计算机计算中时域不可能是连续的，则将其离散化，变为无限个时刻点，每个时刻点间隔为 $T_s$：

  $F(\omega) = \sum_{j=-\infty}^{\infty}[f(jT_s)e^{-i\omega jT_s}T_s]$

  实际上 k 是有限个的，由于 $e^{-i\omega t}$ 是以 $2\pi$ 为周期的连续函数，则可以把 $2\pi$ 分为 n 份，在其中均匀的取样，同时对时间间隔归一化 $(T_s = 1)$，则有：

  $F(\omega) = \sum_{j=0}^{n-1} f(j)e^{-i\frac{2\pi j\omega}{n}}$

- 频域 $\omega$ 离散化：

  对于 n 个采样点，可知最多能求解的 $\omega$ 点对应的 $F(\omega)$ 函数值同样为 n 个，即对 $\omega$ 进行离散化，将上式变成 n 个方程：

  $F(\omega k) = \sum_{j=0}^{n-1} f(j)e^{-i\frac{2\pi jk\omega}{n}}, \quad k = 0, 1, \ldots, n-1$

  记 $y_k = F(\omega k)$，$x_j = f(j)$，对 $\omega$ 同样进行归一化，则上式可以化为：

  $y_k = \sum_{j=0}^{n-1} x_j e^{-i\frac{2\pi jk}{n}} \quad k = 0, 1, ..., n-1$

  **k 是大循环，j 是小循环。**

- 注意：该方法的时间复杂度为 $\Theta(n^2)$。

### 3.2.3 FFT

用于在 DFT 的基础上，减少其复杂度。

**基本原理：**

- 记 $\omega(n)=e^{-i\frac{2\pi}{n}}$，则 $\omega(n)^k$ 恰好为方程 $x^n=1$ 的第 k 根。上式化为：

  $y_k=\sum_{j=0}^{n-1}x_j\omega(n)^{kj}\quad k=0,1,...,n-1$

- 同样有性质成立：

  性质 1：$\omega(n)^{2k}=\omega(n/2)^k$ 不同于：$[\omega(n)^k]^2=\omega(2n)^k$

  性质 2：$\omega(n)^{kn}=1$

  性质 3：$\omega(n)^{kn/2}=-1$

  性质 4：$\omega(n)^k=\omega(n)^{k+n}=-\omega(n)^{k+n/2}$

- 则可利用这些性质化简 DFT 方程。

  把 $\omega(n)^j$ 视为整体，首先考虑单个方程的化简。(化简内循环 j)

  将其拆分成奇数和偶数两部分相加，有：

  $y_k=\sum_{j=0}^{n/2-1}x_{2j}\omega(n)^{2jk}+\sum_{j=0}^{n/2-1}x_{2j+1}\omega(n)^{(2j+1)k}$

  记 n=2m，利用性质 1 和 4，化简为：

  $y_k=\sum_{j=0}^{m-1}x_{2j}\omega(m)^{jk}+\omega(n)^k\sum_{j=0}^{m-1}x_{2j+1}\omega(m)^{jk}$

- 考虑方程间的化简：(化简大循环 k)

  由性质 4：

  $(\omega(m)^{k+m})^j=\omega(m)^{kj}$、$(\omega(n)^{k+m})^j=(\omega(n)^{k+n/2})^j=-\omega(n)^{kj}$

  因此可得 $y_{k+m}$ 的表达式与 $y_k$ 几乎一样，区别仅在于第二部分的 $\omega(n)^{k+m}$ 由于对应方程级数仍然为 n，因此变化为 $-\omega(n)^k$。

  最终得到：

  $$\begin{cases} y_k=\sum_{j=0}^{m-1}x_{2j}\omega(m)^{kj}+\omega(n)^k\sum_{j=0}^{m-1}x_{2j+1}\omega(m)^{kj} \\ y_{k+m}=\sum_{j=0}^{m-1}x_{2j}\omega(m)^{kj}-\omega(n)^k\sum_{j=0}^{m-1}x_{2j+1}\omega(m)^{kj} \\ k=0,1,\ldots,m-1 \end{cases}$$

  可记为：

  $$\begin{cases} y_k=G(x^2)+xH(x^2) \\ y_{k+m}=G(x^2)-xH(x^2) \end{cases}$$

注意其中的 x 实际上指的是 $\omega(n)$，而非前式的 x。

也就是说，只要能够得到 $y_k$，就一定能够得到 $y_{k+m}$。因为每一个 m、k 下，H 和 G 总是相等的。

- 分治方法：

  完整的分治过程不仅包括利用 k+m 与 k 的关系不断对方程组进行减半的拆分，还包括对方程内的不同指数的项按奇偶进行拆分。

  对于 n=2m 的划分可以一直进行下去，但由于每一次方程数目减半，方程内也需要继续进行划分以减少系数，同时每一行 y 的表达式增加，直到最简单的形式：H 和 G 中不含有 x 即 $y = G + xH$。

- 复杂度：由于对于个点 n 而言，一共需要在 $log_2(n)$ 个位置建立方程求解 (m 对应的位置才需要)，而每一次需要进行 n 次乘法 (x 与 $\omega$ 相乘)，因此总的复杂度量级为 $nlog_2(n)$。

- 注意：方程数目或多项式系数 +1 必须为 $2^n$ 次方，否则需要补零。

- 注意：可见求解中需要计算全部的 $\omega(m)^{kj}$，m=2,4,...,n/2、k=0,1,...,n-1。但利用性质 1，k 计算到 n/2-1 就可以了。

- 注意对于 n 下的方程，实际上只有单独的 x 部分是 n 的，H(x) 和 G(x) 对于的全是 m。

## 3.3 多项式乘法与 FFT

### 3.3.1 多项式的表示方法

系数表示法：用一个多项式的各个项系数来表达该多项式。

点值表示法：把 n-1 阶多项式看成一个函数，从上面选取 n 个点，从而利用这 n 个点来唯一的表示这个函数。每个点记为 $(x_i, y(x_i))$。

DFT：多项式由系数表示法转为点值表示法的过程；

IDFT：把一个多项式的点值表示法转化为系数表示法的过程。

FFT 就是通过取某些特殊的 x 的点值来加速 DFT 和 IDFT 的过程。

### 3.3.2 点值表示法与 FTT 关系

在点值表示法下，单纯的多项式相乘复杂度为 $n$，因为在向量乘中，$x_k$ 保持不变，而仅仅需要将各项 $f(x_i)$ 和 $g(x_i)$ 相乘。

对于 DFT 和 IDFT 过程，复杂度则取决于这两个转化过程，对于 $y_i = a_0 + a_1 * x_i + a_2 * x_i^2 + ... + a_n * x_i^n$ 方程组：

- 已知 A 和 X 向量，求解 Y；
- 已知 Y 和 X 向量，求解系数向量 A；

最适合带入的 X 的值即为方程 $x^n = 1$ 的根，$\omega_n^k, k = 0, 1, .., n-1$。由于每个方程系数是一样的，这样就可以利用之前复数的周期性质，减少乘的数量，快速转化。

复杂度同 FFT 算法，为 $nlog_2(n)$ 量级。

带入 x，对于第 k 行方程，表示为：

$y_k = \sum_{j=0}^{n-1} a_j x_k^j = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j \quad k = 0, 1, ..., n-1$

参考 FFT 基本式：$y_k = \sum_{j=0}^{n-1} x_j \omega(n)^{kj} \quad k = 0, 1, ..., n-1$

则若视 $a_j$ 为 $x_j$，则两式完全相等。可以采用同样的方法进行化简。

编写程序时注意 $(e^{\theta i} = cos(\theta) + isin(\theta))$：对于 $\omega_n^k = e^{-i\frac{2\pi k}{n}}$，在传统意义上表示时域向频域的转化关系，但在多项式中则表示原项乘以了逆矩阵且扩大了 n 倍。多项式乘法中，如果只是单纯的相乘，应采用 $\omega_n^k = e^{i\frac{2\pi k}{n}}$。

### 3.3.3 FFT 加速 DFT

在 DFT 中，a 与 x 已知，求解 y；

**基本 FFT 实现：**

- 待乘式次数补 0，使满足 $n = 2^l$；
- 计算出所有的 x：$\omega(m)^k$、$m = 2, 4, ..., n/2$、$k = 0, 1, ..., n/2 - 1$。
- 利用 FFT 部分的奇偶分治，只考虑第 0 行方程，将其拆分到最终只剩两个与 $\omega$ 无关的参数 $a$：

  即 n=1 时：$G_0 = y_0^{(0)} = a_0 * \omega_1^0 = a_0$；$H_0 = y_1^{(0)} = a_1 * \omega_1^0 = a_1$。
- 然后在纵向和横向上不断“滚雪球”一般累加回各项或方程，顺序与拆分时相反。

  – 首先考虑横向的累加：

每轮累加时都满足：下一个偶数项/奇数项 = 上一个偶数项 $+\omega_n^k \times$ 上一个奇数项。其中 n 是当前方程下的 n。

则通过这样的步骤以翻倍的速度不断加回之前被分治的各奇偶项，直到得到最终的 y。由于 H 和 G 都是一轮一轮累加出来的，避免了利用求和公式累加幂导致 j 对 $\omega$ 影响。

– 然后考虑纵向的累加：

在每一轮横向累加时利用当前 n 值下的周期性，每一个 n 下增加对 $k+n/2$ 列的计算即可 (即蝴蝶操作)。但注意由于每一次的原方程并不完整，因此每一次纵向回滚时都需要重新计算所有的行方程 (更新 y)(就是用新的 G 和 H 变换加减号来计算)。

- 示例：

以 8 项式为例，首先考虑横向的分治和回滚，略去行系数 k：

$x_0$、$x_1$、$x_2$、$x_3$、$x_4$、$x_5$、$x_6$、$x_7$　　n=8

分治步骤：

第一次：$x_0$、$x_2$、$x_4$、$x_6$；$x_1$、$x_3$、$x_5$、$x_7$　　n=4

第二次：$x_0$、$x_4$；$x_2$、$x_6$；$x_1$、$x_5$；$x_3$、$x_7$　　n=2

第三次：$x_0$；$x_4$；$x_2$；$x_6$；$x_1$；$x_5$；$x_3$；$x_7$　　n=1

回滚：

最开始：$G_0 = a_0$；$H_4 = a_4$；$G_2 = a_2$；$H_6 = a_6$；$G_1 = a_1$；$H_5 = a_5$；$G_3 = a_3$；$H_7 = a_7$；　　n=1

滚 1 次：$G_{04} = G_0 + \omega(n)H_4$；$H_{26} = G_2 + \omega(n)H_6$；$G_{15} = G_1 + \omega(n)H_5$；$H_{37} = G_3 + \omega(n)H_7$；　　n=2

滚 2 次：$G_{0426} = G_{04} + \omega(n)H_{26}$；$G_{1537} = G_{15} + \omega(n)H_{37}$　　n=4

滚 3 次：$Y = G_{0426} + \omega(n)H_{1537}$　　n=8

得到了最终的 y 值。

然后考虑纵向的回滚，记一开始为第 0 行：

第一次：0->1　　n=2,m=1

第二次：0->2,1->3　　n=4,m=2

第三次：0->4,1->5,2->6,3->7　　n=8,m=4

最终求得了每一行的 y 值。

注意：除非最后一次 (n) 计算，其他次 (n) 的计算都是不完整的，因此每一个 n 下都需要重新计算所有的其余列。

- 递归程序参考：

```
RECURSIVE-FFT(a)
    n=a.length
    if n==1
        return a
    E={a[0],a[2],...,a[n-2]}
    O=(a[1],a[3],...,a[n-1]}
    y_E=RECURSIVE-FFT(E);
    y_O=RECURSIVE-FFT(O);
    for k=0 to n/2-1
        w=e^(2πki/n)
        y[k]=y_E[k]+w*y_O[k]
        y[k+n/2]=y_E[k]-w*y_O[k]
    return y
```

- 此过程复杂度为 $nlog_2n$。

**高效 FFT 实现：**

之前的 FFT 实现中，在行之间的计算顺序每次 (n) 都是按 0 到 n 增加的，但是行内则是对每一项进行了重新排列，在递归方法下需要花费大量空间用于创建和维护数组。而如果一开始每一行的项就是已经是排列后的，则可以利用迭代法来求解，提高 FFT 效率。

重要规律：在原始的顺序下，每个项序号用二进制表示 (四位)，然后把每个数的二进制顺序翻转一下，就是最终拆分完全后每个数的序号。

蝴蝶变换：输入 $x_1$、$x_2$，通过 $y_1 = x_1 + x_2$、$y_2 = x_1 - x_2$，使最终输出 $x_1 = y_1$、$x_2 = y_2$ 的方法。

迭代法 FFT 实现：

- 翻转多项式所有的系数 $a_i$，变化为需要的排列顺序；
- 以 a 的次序为处理顺序；
- 进行主循环，记 step，从 1 开始每次自身乘 2 递增直到 n/2-1；

(记 step 个系数的 H+xG 的值为大单元，则 step 表示分治下各部分系数数量为 step 的情况 (不管行))

- 计算当前 step 下的 $\omega_n^1 = e^{i\frac{2\pi}{n}}$；(由于因为这是逆操作，step 始终是只为原来一半的即表示 m，因此利用当前的 H+xG 求解新 H 或 G 时，在 x 中需要将 step 乘 2，即 $n = e^{i\frac{\pi}{n}}$)

- 进行中循环，记 j，从 0 开始每次增加 2 倍的 step 直到 n-1；(将向量 a 按当前 step 大小全分割 (总计 n/step 个)，每一次循环处理两个大单元 (序号间隔 step)，最终得到全部更新的 a 向量)

  (j 表示当前处理的 a 向量范围，每一个中循环内 $[j, j + 2step)$)

  可以视为对单行方程的分割。随 step 增加 j 取值不断减少。当 step=n/2 时，不分割，对应的 $a_j$ 即 j 行的计算值。

- 进行小循环，记 k，从 j 开始 +1 增加 step 个数为止；(利用之前 step 下计算的上一轮的 a 向量值，来得到可求的每一行方程内对应传入的 step 位置的 G+xH 的值，即更新一部分 a 向量 (2step 个))

  (k 即表示 a 向量的序号，每次小循环会更新 j 即 2step 的部分 a 向量，直到全部更新完成)(在第一次传入 j 中表示行的序号，从 0 到 step；而在之后传入的 j 下，a 向量的序号并不对应于列，需要减去之前的序号 (即 $k - j$ 或 $k - 2 * step * l$ 才表示列))

  可以视为单方程内分割下的方程间分割的分别计算。随 step 增加 k 取值不断增加，当 step=n/2 时，计算量最大，恰好为全部的 a 数。

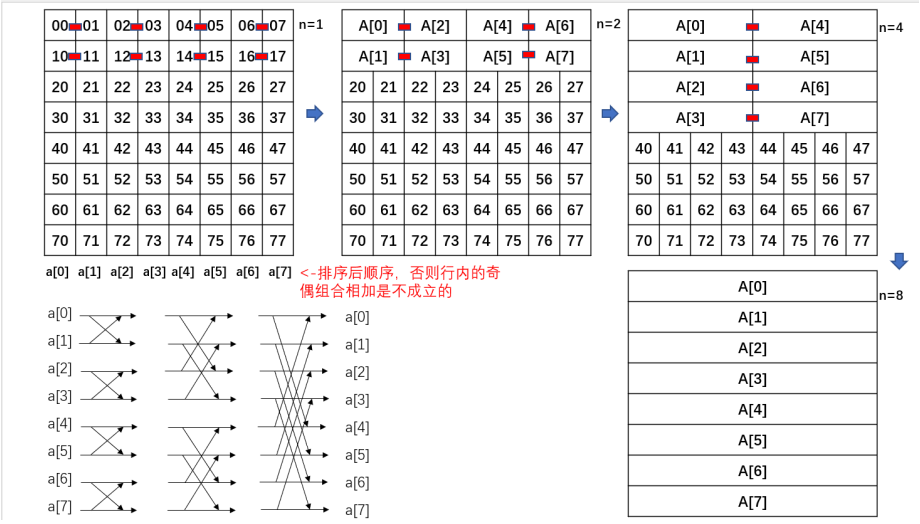
- 注意：传入的 a 矩阵为按顺序排列的系数向量，由于采用了重复赋值更新，在一轮大循环后，a 的意义就已经发生变化了，a 向量按一定的规律在不同的 step 下排列，但最后会表示为每一行的累加值。

- 程序：

  注意需要考虑不同行中 k 的影响，这是通过小循环中从第 0 行开始，每次循环 wnk 项乘以一个 wn 来的 ($\omega_{2n}^{0+1+1+\cdots}$)。

- 蝴蝶变换示意图：

```cpp
typedef complex<double> cd;//C++ 自带复数类，需要头文件complex
void fft(cd *a,int n)
{
    for(int i=0;i<n;i++) if(i<rev[i]) swap(a[i],a[rev[i]]);
    for(int step=1;step<n;step<<=1)
    {
        cd wn=exp(cd(0,PI/step));//exp：e的幂，此处计算单位根
        for(int j=0;j<n;j+=step<<1)
        {
            cd wnk(1,0);//cd构造函数：cd(实数部分,虚数部分/i);
            for(int k=j;k<j+step;k++)
            {//蝴蝶操作
                cd x=a[k];
                cd y=wnk*a[k+step];
                a[k]=x+y;
                a[k+step]=x-y;
                wnk*=wn;
            }
        }
    }
}
```




### 3.3.4 FFT 加速 IDFT

在 IDFT 中，y 与 x 已知，求解 a。

将方程写为如下矩阵形式：(之前也是这种形式，只不过 a 向量乘进去

了)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

需要在等式两边左侧乘以 $\omega$ 的逆矩阵，即可变为 $A = \omega^{-1}y$ 的格式，与之前的 y=ax 形式完全一样，可用相同方法求解。即输入为 y 向量，返回的为新的系数向量 a。

可以证明，对于矩阵每一项取倒数再除以 n 就是该矩阵的逆矩阵。

注意为保证输出 a 向量的顺序，也需要在计算前对 y 向量进行翻转。

则程序基本上与 DFT 过程相同，区别在于对系数的处理中，将 y 向量每一项除以 n；然后在 H+xG 处理中，x 的初始定义由 $\omega_n^k = e^{2\pi ik/n}$ 变为 $\omega_n^k = e^{-2\pi ik/n}$(e 的系数加个负号)。

完整的程序可以写为一个函数，除上述操作外其他部分完全一样。

### 3.4 二维串行 FFT 算法

可以由两个方向的一维 FFT 来完成。

## 4 MPI 并行程序设计基础

**与 pthread 区别**

定义：Massage Passing Interface: 是消息传递函数库的标准规范。是一种新的库描述, 不是一种语言。

mpi 是基于分布式内存系统，而 openmp 和 pthread 基于共享内存系统；

即 mpi 之间的数据共享需要通过消息传递，因为 mpi 同步的程序属于不同的进程，甚至不同的主机上的不同进程。相反由于 openmp 和 pthread 共享内存，不同线程之间的数据就无须传递，直接传送指针就行。

同时 mpi 不同主机之间的进程协调工作需要安装 mpi 软件（例如 mpich）来完成。

## 4.1 并行相关分类

**计算机架构：**

- SMP：SMP 是对称多处理技术。具有多个 CPU，所有的 CPU 共享一个内存，使用相同的地址空间。所有的 CPU 通过一条总线 (bus) 和内存以及 IO 设备 (硬盘等) 连接。总线同一时刻只能处理一个请求，当有多个 CPU 的访存访问请求时，只能一个一个处理。
- MMP：类似于集群，但 MMP 使用了更多定制化的组件，包括网络、处理器、操作系统等；而 cluster 运行通用操作系统，互连网络使用商业标准的 IB 和以太网设备连接，存储为 SAN、NAS 和并行文件系统。
- Cluster：集群，它至少将两个系统连接到一起，使两台服务器能够像一台机器那样工作或者看起来好像一台机器。基本特征是具备多个 CPU 模块，每一个 CPU 模块由多个 CPU 组成，而且具备独立的本地内存、I/O 槽口等。
- MMP 与 Cluster 区别：
  - MPP 实际上是一台机器，这台机器有使用高速网络紧密连接的成千上万个处理器，只有一个操作系统。
  - cluster 实际上是有多台机器，每个机器有自己的操作系统（一般都是一样的）、硬盘、内存等，这些机器使用一些普通网络的一些变体连接起来，使用某些系统帮助分配任务给这些主机。

**并行计算机系统结构编程模型 (Flynn 分类法)：**

- 单指令单数据 (SISD)：SISD 是标准意义上的串行机，具有如下特点：1）单指令：在每一个时钟周期内，CPU 只能执行一个指令流；2）单数据：在每一个时钟周期内，输入设备只能输入一个数据流；3）执行结果是确定的。这是最古老的一种计算机类型。
- 单指令多数据 (SIMD)：SIMD 属于一种类型的并行计算机，具有如下特点：1）单指令：所有处理单元在任何一个时钟周期内都执行同一条指令；2）多数据：每个处理单元可以处理不同的数据元素；3）非常适合于处理高度有序的任务，例如图形/图像处理；4）同步（锁步）及确定性执行；5）两个主要类型：处理器阵列和矢量管道。

- 多指令单数据 (MISD)：MISD 属于一种类型的并行计算机，具有如下特点：1）多指令：不同的处理单元可以独立地执行不同的指令流；2）单数据：不同的处理单元接收的是同一单数据流。这种架构理论上是有的，但是工业实践中这种机型非常少。

- 多指令多数据 (MIMD)：MIMD 属于最常见的一种类型的并行计算机，具有如下特点：1）多指令：不同的处理器可以在同一时刻处理不同的指令流；2）多数据：不同的处理器可以在同一时刻处理不同的数据；3）执行可以是同步的，也可以是异步的，可以是确定性的，也可以是不确定性的。这是目前主流的计算机架构类型。

**并行程序类型：**

- 主从式 M-S：即 Master/Slaver 模式。核心思想是基于分而治之，将一个原始任务分解为若干个语义等同的子任务，并由专门的工作者线程来并行执行这些任务，原始任务的结果是通过整合各个子任务的处理结果形成的。各子任务互不相干。

- 对称式 SPMD：(Single Program Multiple Data) 指单程序多数据。类似于 SIMD，但在 SPMD 中，虽然各处理器并行地执行同一个程序，但所操作的数据不一定相同 (即各处理器只在需要时进行同步，而不是同步地执行每一条指令)。

- 自主式 MPMD：(Single Program Multiple) 指多程序多数据。相比于 SPMD，相当于各自进程执行各自的程序。SPMD 和 MPMD 的表达能力是相同的，只是针对不同的问题编写难易而已。MPI 是可以写 SPMD 和 MPMD 的并行程序的。

  重点在 SPMD。

## 4.2 并行程序基本结构

MPI 采用了 SPMD 模型，即每一个 mpi 进程都是一个独立的程序，相互之间通过 MPI communicator 通信交换数据，他们之间通过 rank 编号来区分。如果没有使用 rank 编号区分而直接定义的话，那么该变量在每个进程里都一样的。但是实际中的大部分数据，都会根据进程编号而不同。

也就是说，直接定义的变量，必然是同时相同的存在于多个线程内的。

- 进入 MPI 环境。产生通讯子 (进程序号、进程数)。
- 程序主体。
- 退出 MPI 环境。

## 4.3 MPI 数据类型

| MPI 数据类型 | C 中对应数据类型 |
|---|---|
| MPI_SHORT | short int |
| MPI_INT | int |
| MPI_LONG | long int |
| MPI_LONG_LONG | long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | char |

## 4.4 MPI 通讯子 (通信域) 基础

定义及功能：

通讯子定义了一组能够互相发消息的进程。在这组进程中，每个进程会被分配一个序号，称作秩 (rank)，进程间显性地通过指定秩来进行通信。

内容：

- 上下文 (context)：提供了一个相对独立的通信区域，不同的信息在不同的上下文中传递，不同的上下文的信息互不干扰，上下文可以区分不同的通信。
- 进程组 (group)：组是一个进程的有序集合，在实现中可以看作是进程标识符的一个有序集。一个通信域对应一个进程组。

组内的每个进程与一个整数 rank 相联系，称为序列号，从 0 开始并且是连续的。

- 虚拟处理器拓扑 (topology)：…

附注：进程：一个进程对应一个 pid 号，同一个进程可以属于多个进程组 (每个进程在不同进程组中有个各自的 rank 号)，因此也可以属于不同的通信域。

默认 (最大范围)：MPI_COMM_WOLRD，这是 MPI 已经预定义好的通讯子，是一个包含所有进程的通讯子。最大集。

参考链接：http://scc.ustc.edu.cn/zlsc/cxyy/200910/MPICH/mpi52.htm

**通信域产生方法：**

- 在已有通信域基础上划分获得：MPI_Comm_split
- 在已有通信域基础上复制获得：MPI_Comm_dup
- 在已有进程组的基础上创建获得：MPI_Comm_Create

**进程组产生方法：**

可以当成一个集合的概念，可以通过“子、交、并、补”各种方法。所有进程组产生的方法都可以套到集合的各种运算。

## 4.5 进程通信原理

通信的基础建立在不同进程间发送和接收操作。一个进程可以通过指定另一个进程的秩以及一个独一无二的消息标签 (tag) 来发送消息给另一个进程。接受者可以发送一个接收特定标签标记的消息的请求（也可以不管标签，接收任何消息），然后依次处理接收到的数据。这样的涉及一个发送者以及一个接受者的通信被称作点对点通信。

如果某个进程需要跟所有其他进程通信。则可用专门的接口来处理这类所有进程间的通信，称为集体性通信。

## 4.6 MPI 基本函数

### 4.6.1 并行环境管理函数

**MPI_Init(&argc, &argv)**

- 功能：初始化 MPI 环境。产生一个通讯子 (称 MPI_COMM_WORLD)
- 参数：
  - 就是 C++main 函数传入的参数，形式如上。
- 备注：必须在调用该函数后，才能调用其他的 MPI 函数。不关心返回值。

**MPI_Finalize()**

- 功能：结束 MPI 环境。
- 参数：无
- 备注：任何 MPI 程序结束时，都需要调用该函数。不关心返回值。

### 4.6.2 MPI 通讯子操作函数

**MPI_Comm_rank 函数**

```
int MPI_Comm_rank(
    MPI_Comm comm, //[传入] 当前进程所在的通讯子
    int *rank //[传出] 进程号
)
```

- 功能：获得当前进程的进程标识 (进程号)。
- 返回值：不关心。
- 备注：在调用该函数时，需要先定义一个整型变量如 myid，不需要赋值。将该变量传入函数中，会将该进程号存入 myid 变量中并返回。

**MPI_Comm_size 函数**

```
int MPI_Comm_size(
    MPI_Comm comm, //[传入](不一定本进程的) 通讯子。如果通讯子为 MP_Comm_WORLD，即获取总进程数
    int *size //[传出] 进程数目
)
```

- 功能：是获取该通讯子内的总进程数。

- 返回值：不关心。
- 备注：用法类似前一个。

**MPI_Comm_dup 函数**

int MPI_Comm_dup(
MPI_Comm comm,//[传入] 要复制的通讯子
MPI_Comm *newcomm //[传出] 新的通讯子，具有相同的组和从源复制的任何缓存信息，但它包含新的上下文信息
)

- 功能：复制现有通讯子及其所有缓存的信息
- 返回值：不关心。
- 备注：无。

**MPI_Comm_compare 函数**

int MPI_Comm_compare(
MPI_Comm comm1,//[传入] 要比较的通讯子 1
MPI_Comm comm2 //[传入] 要比较的通讯子 2
)

- 功能：比较两个通讯子
- 返回值：
  - MPI_IDENT：两个通讯子的组和上下文相同。
  - MPI_CONGRUENT：上下文不同、组相同。
  - MPI_SIMILAR：上下文不同，组的成员相同但次序不同。
  - MPI_UNEQUAL：都不相同。
  - 失败：错误代码。
- 备注：无。

**MPI_Comm_create 函数**

int MPI_Comm_Create(
MPI_Comm comm, //[传入] 源通讯子
MPI_Group group, //[传入] 定义从源通讯子中得到的进程子集(进程组)
MPI_Comm *newcomm //[传出] 新的通讯子
)

- 功能：在通过组操作得到子进程组后，利用该函数得到一个子通讯子，子通讯子与子进程组对应。
- 返回值：返回成功时为 MPI_SUCCESS，否则为错误代码。
- 备注：
  - 创建新的，老的还在；
  - group 的产生可参考第 9 章；
  - 可以通过此方法得到 split 函数无法获得的子通讯子。

**MPI_Comm_split 函数**

int MPI_Comm_split(
MPI_Comm comm,//[传入] 要拆分的通讯子。也就是被划分的范围
int color,//[传入] 相同的 color 的通讯子会被划分成同一个子通讯子
int key,//[传入] 新通讯子中调用进程的相对等级 (rank)。进程在新的通讯子中按参数键的值定义的顺序排列
_Out_ MPI_Comm *newcomm //[传出] 新的通讯子
)

- 功能：用于将指定的单个通信的进程组划分为任意数量的子组。
- 返回值：返回成功时为 MPI_SUCCESS，否则为错误代码。
- 备注：将原有的通讯子拆分了，新的组成老的。且子组的数量由在所有进程中指定的 color 数量确定。生成的通信器不重叠。

**MPI_Comm_free 函数**

```
int MPI_Comm_free(
    MPI_Comm *comm //[输入] 指向要释放的通讯子的指针
);
```

- 功能：释放通过 dup、create 或 split 创建的通讯子。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：

  此操作将通讯子标记为释放。句柄设置为 MPI_COMM_NULL。任何使用此通讯子的挂起操作都将正常完成。直到没有对对象的活动引用时，对象才会被释放。

  这一功能既适用于内部通讯子，也适用于外部通讯子。

  所有缓存属性的删除被回调函数以不确定的顺序调用。

# 5 点到点通信函数

## 5.1 阻塞式

阻塞式：发送或接受完数据后该 rank 进程才会继续执行。而且必须发送成功 (但不一定接收成功)。

### 5.1.1 MPI_Send 函数

```
int MPI_Send(
    void* data,//[传入] 发送缓冲区地址 (要发送数据的所在地址)
    int count,//[传入] 要发送数据的大小
    MPI_Datatype datatype,//[传入] 数据的类型
    int dest,//[传入] 目标的进程编号
    int tag,//[传入] 消息标记 (用于区分不同类型的消息)
    MPI_Comm send_comm//[传入] 目标的通讯子
)
```

- 功能：执行标准模式发送操作，并在可以安全地再利用发送缓冲区时返回 (直到缓存为空)。

- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：为非本地函数，成功完成取决于是否存在匹配的接收函数。

### 5.1.2 MPI_Recv 函数

int MPI_Recv(
void *buf,//[传出] 接收缓冲区地址 (要接收数据的存放地址)
int count,//[传入] 接收的数据大小
MPI_Datatype datatype,//[传入] 数据的类型
int source,//[传入] 指定来源的进程编号,若为 MPI_ANY_SOURCE 表示任意来源
int tag,//[传入] 指定来源的消息标记，若为 MPI_ANY_TAG 表示任意标签都接受
MPI_Comm recv_comm,//[传入](接收方) 通讯子，需要与 send 中的相同。通常情况下 send 和 recv 均为 MPI_COMM_WOLRD
MPI_Status *status //[传出] 接受状态，即指向描述已完成操作的 MPI_Status 对象 (结构体) 的指针。如果不需要该状态信息，直接赋常量 MPI_STATUS_IGNORE 即可
);

- 功能：执行接收操作，并且在收到匹配的消息之前不返回 (直到缓存被填充)。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 接收消息的长度必须小于或等于接收缓冲区的长度。如果所有传入数据都不适合接收缓冲区，则此函数将返回溢出错误。
  - 发送和接收操作之间存在不对称性。接收操作可以接受来自任意发送方的消息，但发送操作必须指定唯一的接收方。
  - 注意防止死锁。

**函数成功接受的必要条件**

- send_comm==recv_comm(都是要为接收方的通讯子)

- send_tag==recv_tag
- send_dest==recv_rank(接收方进程编号)
- send_rank(发送方进程编号)==recv_source

#### 5.1.3 MPI_Sendrecv 合成函数

```
int MPI_Sendrecv(
    void *sendbuf,//[传入] 发送缓冲区地址
    int sendcount,//[传入] 数据大小
    MPI_Datatype sendtype,//[传入] 信息的数据类型
    int dest,//[传入] 目标的进程编号
    int sendtag,//[传入] 消息标记
    void *recvbuf,//[传出] 接收缓冲区地址
    int recvcount,//[传入] 接收的数据大小
    MPI_Datatype recvtype,//[传入] 指定来源的数据类型
    int source,//[传入] 指定来源 (接收) 的进程编号
    int recvtag,//[传入] 指定来源的消息标记
    MPI_Comm comm,//[传入](接收方) 通讯子
    MPI_Status *status//[传出] 接受状态，同上
);
```

- 功能：发送和接收消息。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：send、recv、sendrecv 互相兼容，sendrecv 既可以接受 send 的数据，也可以给 recv 发送数据。

#### 5.1.4 MPI_Sendrecv_Replace 合成函数

```
int MPI_Sendrecv_replace(
    void* buffer,//[传入传出] 发送和接收缓冲区的初始地址
    int count,//[传入传出] 数据的大小
    MPI_Datatype sendtype,//[传入传出] 数据的类型
    int dest,//[传入] 目标的进程编号 rank
```

int sendtag,//[传入] 发送的信息的消息标记
int source,//[传入] 指定来源的进程编号
int recvtag,//[传入] 指定来源的消息标记
MPI_Comm comm,//[传入](接收方) 通讯子
MPI_Status*status//[传出] 接受状态，同上
)

- 功能：使用单个缓冲区发送和接收消息。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：与 Sendrecv 相比，不同之处在于使用同一个缓冲区来接收和发送数据 (因此前三个参数是一样的)。也正因为如此，效率相比于 Sendrecv 低下。

#### 5.1.5 消息查询函数 MPI_Probe

int MPI_Probe(
int source,//[传入] 查询的进程编号
int tag,//[传入] 查询的消息标记
MPI_Comm comm,//[传入] 查询的通讯子
MPI_Status *status//[传出] 接受状态，同上
);

- 功能：探测接收消息的内容，但不影响实际接收到的消息。
- 返回值：函数执行成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：为阻塞型探测，直到有一个符合条件的消息到达才返回，**即函数成功返回则一定接收到了信息**。

#### 5.1.6 消息查询函数 MPI_IProbe

int MPI_IProbe(
int source,//[传入] 查询的进程编号
int tag,//[传入] 查询的消息标记
MPI_Comm comm,//[传入] 查询的通讯子

int *flag,//[传出] 标记，如果找到了符合要求的信息 (source、tag、comm)，就为 1，否则为 0

MPI_Status *status//[传出] 接受状态，同上

);

- 功能：同 Probe，但为非阻塞型。
- 返回值：函数执行成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：为非阻塞型探测，无论是否有一个符合条件的消息到达，都立刻返回。通过 flag 判断是否找到。

#### 5.1.7 消息查询函数 MPI_Get_Count

int MPI_Get_count(

MPI_Status *status,//[传出] 接收状态

MPI_Datatype datatype,//[传入] 每个接收缓冲区元素的数据类型

int *count//[传出] 接收到的元素数目

);

- 功能：获得接收到的某一种类型的元素数量。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：通过对 count 的值进行修改。

### 5.2 非阻塞式

**非阻塞式 (异步通信) 与阻塞式 (同步通信) 区别**

- 同步通信中在接受到数据和发送完数据之前，进程处于挂起状态，完成后才允许进程继续执行下一语句。
- 异步通信中发送方在将要发送的消息在消息缓冲区中后, 即可返回; 接受方不管消息缓冲区中是否已有发送原语发送的消息，都将返回。当消息被确切地发出或收到时，系统将用中断信号 (非阻塞式会产生独有的句柄 request) 通知发送方或接受方。在此之前，它们可以周期性地查询、暂时挂起或执行其它计算，以实现计算与通信的重叠。

- 异步通信需要系统提供一个消息缓冲区，同步通信则不需要。
- 在同步通信中，通信可以是异步开始的，但必将是同步结束的；在异步通信中，通信可以是异步开始的，也可以是异步结束。

### 5.2.1 MPI_Isend 函数

int MPI_Isend(
void* data,//[传入] 发送缓冲区地址
int count,//[传入] 数据大小
MPI_Datatype datatype,//[传入] 信息的数据类型
int dest,//[传入] 目标的进程编号
int tag,//[传入] 消息标记 (用于区分不同类型的消息)
MPI_Comm send_comm,//[传入] 目标的通讯子
MPI_Request *request //[传出]所请求的通信操作的句柄，用来描述非阻塞发送或接收的完成情况。是提供给后面的非阻塞通信检测/等待函数用的。
)

- 功能：启动标准模式发送操作，在调用后不用等待通信完全结束就可以返回。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：本地函数，此函数可以在将消息发送到缓冲区之前返回。也就是说，在执行完该函数后，数据并没有完全写入缓冲区就已经开始执行接下来的命令了，并不会阻塞在这里等待发送完成，因此函数返回后不能立刻修改 data 中的内容，否则发送的数据也跟着变化，需要执行消息请求完成函数 MPI_Wait 保证发送完毕。

### 5.2.2 MPI_Irecv 函数

int MPI_Irecv(
void *buf,//[传出] 接收缓冲区地址
int count,//[传入] 接收的数据大小
MPI_Datatype datatype,//[传入] 信息的数据类型

int source,//[传入] 指定来源的进程编号,若为 MPI_ANY_SOURCE 表示任意来源

int tag,//[传入] 指定来源的消息标记，若为 MPI_ANY_TAG 表示任意标签都接受

MPI_Comm recv_comm,//[传入](接收方) 通讯子，需要与 send 中的相同。通常情况下 send 和 recv 均为 MPI_COMM_WOLRD

MPI_Request *request//[传出]所请求的通信操作的句柄，同之前

);

- 功能：执行接收操作，在调用后不用等待通信完全结束就可以返回。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：此函数是本地的。此函数可以在将消息接收到缓冲区之前返回。即该函数返回后，数据同样并没有写入完全缓冲区，接收并没有完成，同样需要执行消息请求完成函数 MPI_Wait 保证接收完毕后才能操作这部分数据。
- 在调用 wait 和 test 后，已经完成的通信操作的句柄 request 将不可用。

#### 5.2.3 消息请求完成函数 MPI_Wait

int MPI_Wait(

MPI_Request *request,//[传入] 通信对象的句柄

MPI_Status *status//[传出] 接收状态，同之前

);

- 功能：用于等待 MPI 请求完成。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - **此函数成功返回时，表示 request 指针表示的通信操作完成，否则通信操作函数就不会结束**。也就是说程序会卡住直到函数执行完毕。
  - 用于一个非阻塞通信。

- 非本地操作，成功完成可能取决于其他进程的匹配操作，即一直等到相应的通信完成后才成功返回。
- 通信操作为空时，立即返回空状态。
- 往往与 isend 和 irecv 连用，与阻塞式区别在于程序在执行到通信时会不会挂起。
- 若接收消息的进程先接收、发送消息的进程再发送，则 MPI 就只会填充程序提供的缓冲区 (发送，接收缓冲区)。而无需使用 MPI 提供的消息缓冲区。这可以减少在接收进程端的内存拷贝操作，提高性能。

#### 5.2.4 消息请求完成函数 MPI_Waitany

int MPI_Waitany(
int count,//[传入] 通信对象的数目
MPI_Request *array_of_requests,//[传入](未完成的) 通信对象句柄的数组指针：定义为 MPI_Request request[count]
int *index,//[传出] 指向一个整数的指针，整数表示通信完成的通信对象在数组中的索引
MPI_Status *status//[传出] 接受状态
);

- 功能：用于等待非阻塞通信对象数组中的任意一个通信对象的完成。当存在多个通信对象时，一旦有一个通信完成，就返回该通信所对应通信对象的序号 index，并释放该通信对象，并把该通信的相关信息存放在 status 中。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：

  注意区分 *request 和 *array_of_requests：

  - *request:指的是单个通信对象句柄的地址，调用时传入 &，同 Ised 和 Irecv。
  - *array_of_requests：指的是通信对象的集合，调用时传入集合即数组的名字。

注意区分 *index 和 *array_of_index：

- *index:在 wait、waitany 这种输出为 1 个通信对象的情况下，地址为该通信对象在数组中的编号；
- *array_of_index：在 waitsome 这种输出为多个通信对象的情况下，为数组指针，表示第 I 个通信对象对应的通信完成信息存放在 array_of_statuses[I] 中，其中完成为 1，未完成为 0；
- waitall 中没有 index。

### 5.2.5 消息请求完成函数 MPI_Waitall

```
int MPI_Waitall(
    int count,//[传入] 通信对象的数目
    MPI_Request *array_of_requests,//[传入] 通信对象句柄的数组指针
    MPI_Status *array_of_statuses//[传出] 接受状态
);
```

- 功能：用于等待非阻塞通信对象数组中的所有通信对象的完成。当所有的通信完成时才返回，并释放所有的通信对象。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：无。

### 5.2.6 消息请求完成函数 MPI_Waitsome

```
int MPI_Waitsome(
    int incount,//[传入] 通信对象的数目
    MPI_Request *array_of_requests,//[传入] 通信对象句柄的数组指针
    int outcount,//[传出] 信完成的通信对象的数量
    int *array_of_index,//[传出] 数组指针，表示通信完成的全部通信对象在数组中的索引
    MPI_Status *array_of_status//[传出] 接受状态
);
```

- 功能：用于等待非阻塞通信对象数组中的任意数量的通信对象的完成。当任意数目的通信完成时就返回，并释放通信完成的通信对象。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：完成非阻塞通信的数目记录在 outcount 中，相应的非阻塞通信对象的下标存放在下标数组中，对应通信的相关信息存放在 array_of_statuses 中。

#### 5.2.7 消息请求检查函数 MPI_Test

**test 与 wait 区别：**

- wait 要一直等到相应的非阻塞通信完成后才成功返回，而 test 在调用后会立刻返回。
- wait 在 any、some 的情况下，会给出具体的完成的通信对象的编号；而 test 在 any 的情况下则不会，只会给出当前是否满足通信完成的判断，仅 some 情况下给出编号。
- 相比于 wait 中的 index 指示完成的通信对象的编号，在 test 中用 flag 来取代，仅用于判断是否满足要求。

```
int MPI_Test(
    MPI_Request *request,//[传入] 通信对象句柄的指针
    int *flag,//[传出] 指示请求是否已完成的判断指针，通信完成为 1，否则为 0
    MPI_Status *status//[传出] 接受状态
);
```

- 功能：用于测试非阻塞通信中通信完成的情况，立即返回。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：函数执行后立刻返回，不会等待。

#### 5.2.8 消息请求检查函数 MPI_Testany

int MPI_Testany(
int count,//[传入] 通信对象的数目
MPI_Request *request,//[传入] 通信对象句柄的数组指针
int *flag,//[传出] 指示请求是否已完成的判断指针，对应索引位置的通信完成则为 1，否则为 0
MPI_Status *status//[传出] 接受状态
);

- 功能：用于测试非阻塞通信数组中是否有任何一个对象已经完成，若有对象完成 **(若有多个，任取一个)**，令 flag=1 立即返回，并释放该对象；否则令 flag=0 并立即返回。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：如果调用时没有完成的通信，flag 仍然为 0。

#### 5.2.9 消息请求检查函数 MPI_Testall

int MPI_Testall(
int count,//[传入] 通信对象的数目
MPI_Request *request,//[传入] 通信对象句柄的数组指针
int *flag,//[传出] 指示请求是否已完成的判断指针，全部通信完成则为 1，否则为 0
MPI_Status *status//[传出] 接受状态
);

- 功能：当非阻塞通信数组中有任意一个非阻塞通信对象对应的非阻塞通信没有完成时，令 flag=false 并立即返回；当所有通信都已经完成时，令 flag=true 并立即返回。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：无。

#### 5.2.10 消息请求检查函数 MPI_Testsome

int MPI_Testsome(
int incount,//[传入] 通信对象的数目
MPI_Request *request,//[传入] 通信对象句柄的数组指针
int outcount,//[传出] 通信完成的通信对象的数目
int *array_of_indices,//[传出] 数组指针，表示通信完成的全部通信对象在数组中的索引
MPI_Status *status//[传出] 接受状态
);

- 功能：立即返回，有几个非阻塞通信已经完成，就令 outcount 等于几，且将完成对象的下标记录在数组中。若没有非阻塞通信完成，则返回 outcount=0。
- 返回值：功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**此函数没有 flag 判断参数，是通过 outcount 参数是否为 0 来发挥原 flag 作用的**。

### 5.3 持久通讯

#### 5.3.1 消息请求检查函数 MPI_Send_init

int MPI_Send_init(
void *buf,//[传入] 发送缓冲区地址
int count,//[传入] 数据大小
MPI_Datatype datatype,//[传入] 信息的数据类型
int dest,//[传入] 目标的进程编号
int tag,//[传入] 消息标记
MPI_Comm comm,//[传入] 目标的通讯子
MPI_Request *request//[传出] 所请求的通信操作的句柄，同之前
);

- 功能：执行发送操作，但请求是持久的，即产生的通信句柄可以重复使用，但每次使用时需要激活和取消占用。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。

- 备注：如果另外一个通信在一个并行计算的内部循环中不断地以**同样的参数**被执行，则没必要每次重新创建通信句柄，即可把这些通信参数一次性捆绑到一个持久通信请求，然后不断用该请求初始化 (激活) 和释放 (取消占用) 消息。

- 注意：**在激活的区间内，该通信就等于一个非阻塞通信**。

  **释放有两种方法，一种是 free 释放，另一种是 wait 或者 test 成功返回，且不可以同时采用**。

#### 5.3.2 MPI_Recv_init 函数

```
int MPI_Recv_init(
    void *buf,//[传出] 接收缓冲区地址
    int count,//[传入] 接收的数据大小
    MPI_Datatype datatype,//[传入] 信息的数据类型
    int source,//[传入] 指定来源的进程编号,若为 MPI_ANY_SOURCE 表示任意来源
    int tag,//[传入] 指定来源的消息标记，若为 MPI_ANY_TAG 表示任意标签都接受
    MPI_Comm recv_comm,//[传入](接收方) 通讯子，需要与 send 中的相同。通常情况下 send 和 recv 均为 MPI_COMM_WOLRD
    MPI_Request *request//[传出] 所请求的通信对象的句柄
);
```

- 功能：执行接收操作，但请求是持久的，同样需要激活和取消占用。

- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。

- 备注：同上可用于化简接收循环。

#### 5.3.3 MPI_Start 函数

```
int MPI_Start(
    MPI_Request *request //[传入] 通信对象的句柄
);
```

- 功能：激活持久通信产生的句柄 request 所对应的通信。

- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：每次使用持久通信对象时都需要执行。

#### 5.3.4 MPI_Startall 函数

int MPI_Startall(
int count,//[传入] 通信对象的数目
MPI_Request *array_of_requests, //[传入] 通信对象的句柄的数组指针
);

- 功能：激活持久通信产生的句柄所对应的通信集合。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：每次使用持久通信对象时都需要执行。

#### 5.3.5 MPI_Request_free 函数

int MPI_Request_free(
MPI_Request *requests, //[传入] 通信对象的句柄
);

- 功能：释放通信对象 (及所占用的内存资源)。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：每次使用的持久通信对象完毕时都需要执行。只能一个一个的释放。

  **若该通信请求相关联的通信操作尚未完成，则等待通信的完成再返回，因此通信请求的释放并不影响该通信的完成**。

  并不是一个激活对应一次释放，每次激活 (start) 表示调用一次发送或接受，该函数返回后调用完毕不用释放，不用每次释放只需要最后释放一次。

  该函数成功返回后 request 被置为 MPI_REQUEST_NULL，与 wait、test 类似。

### 5.3.6 MPI_Cancel 函数

int MPI_Cancel(
MPI_Request *request //[传入] 通信对象的句柄
);

- 功能：非阻塞型，用于取消一个尚未完成的通信请求。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：原理是在 MPI 系统中设置一个取消该通信请求的标志后立即返回，具体的取消操作由 MPI 系统在后台完成。

  **若相应的通信请求已经开始，则它会正常完成，不受取消操作的影响；若相应的通信请求还没开始，则可以释放通信占用的资源。**

  仍需用 MPI_WAIT，MPI_TEST 或 MPI_REQUEST_FREE 来释放该通信请求。

注意：free 和 cancel 都是针对非阻塞通信的，用于结束它们 (前者释放后者取消)。

### 5.3.7 MPI_Test_cancelled 函数

int MPI_Test_cancelled(
MPI_Status *status,//[传入] 接收状态
int *flag //[传出] 通信对象通信情况，1 表示请求成功被取消，否则为 1
);

- 功能：测试以查看请求是否已取消。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：传入的为接受状态，想调用此函数不可将接受状态设置为不需要。此函数往往和 MPI_Cancel 函数联用以查看取消情况。

### 5.4 高维进程

通常先定义一组按顺序序号从小到大的行通讯子,然后利用 MPI_Comm_split 函数对每个通讯子进行拆分，从而产生列通讯子，最终组成二维进程。

方法：在定义行通讯子内将传入进程的 color 按一定的规律赋予不同的 color 值用于将进程划分给不同的通讯子、以及赋予不同的 key 值用于指定进程在通讯子内的优先级，然后利用 split 函数划分即可。

## 6 派生数据类型

通信时必须传递某一类型的数据，可以通过该方法来创建需要传递的某一类数据。

数据类型描述方法：类型图

**类型图 =< 基类型 0 偏移 0>,< 基类型 1 偏移 1>,...,< 基类型 n-1 偏移 n-1>**

- 基类型可以是预定义类型或派生类型;
- 偏移可正可负，没有递增或递减的顺序要求;
- 偏移 i 是指第 i 块类型的起始位置;
- lb：下界，所有的块中偏移量最小的值，即数据类型起始位置。
- ub：上界，所有的块中偏移量 + 块大小最大的值，由于实际上不同类型数据在内存中是顺序存放而非并行存放的，其也等价于**整个数据类型的大小**。
- extent：extent=ub-lb+e，其中 e 是能够使类型图的跨度满足该类型的类型表中的所有的类型都能达到下一个对齐要求所需要的最小非负整数值。

  跨度，该数据类型的类型图中从第一个基类型到最后一个基类型间的距离 (到末尾结束位置)。

### 6.1 连续数据类型 CONTIGUOUS

#### 6.1.1 MPI_Type_contiguous 函数

int MPI_Type_contiguous(

int count,//[传入] 新数据类型中的元素数 (块大小)。
MPI_Datatype oldtype,//[传入] 每个元素的 MPI 数据类型。
MPI_Datatype *newtype//[传出] 新数据类型名字
);

- 功能：定义一个新的数据类型，该数据类型是现有数据类型的许多元素的串联。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 新数据类型的大小取决于旧类型的大小。
  - **块内元素类型相同、元素连续**。
  - 存放示例：$a_1$、$a_2$、$a_3$、$a_4$、$a_5$、...
  - 得到的新类型是将一个已有的数据类型按顺序依次连续进行复制后的结果。还需要赋值的。

## 6.2 向量数据类型 VECTOR

### 6.2.1 MPI_Type_vector 函数

int MPIAPI MPI_Type_vector(
int count,//[传入] 块数：向量中块的数量
int blocklength,//[传入] 块长度：每个块中的元素数量
int stride,//[传入] 步长：块间距 (元素数据类型长度为单位)
MPI_Datatype oldtype,//[传入] 每个元素的 MPI 数据类型
MPI_Datatype *newtype//[传出] 新数据类型名字
);

- 功能：定义由指定大小的指定数量的块组成的新数据类型。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**块内元素类型相同、元素连续；块之间的大小相等、间距相等、元素类型相等**。

  所有的块间距指的都是一个块的开始和下一个块的开始之间的距离，或元素数或字节数。也称为位移，第一个块的位移就是 0。

原理是复制一个数据类型到含有相等大小块的空间，每个块内都是 blocklength 个旧数据类型的拷贝，块间距均为 stride 倍的元素 (旧元素类型大小)。

### 6.2.2 MPI_Type_create_hvector 函数

int MPIAPI MPI_Type_create_hvector(
int count,//[传入] 块数
int blocklength,//[传入] 块长度
MPI_Aint stride,//[传入] 步幅：块间距 (字节为单位)，注意步幅必须是旧数据类型范围的倍数
MPI_Datatype oldtype,//[传入] 每个元素的 MPI 数据类型
MPI_Datatype *newtype//[传出] 新数据类型名字
);

- 功能：同前一个。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**与前一个类似，但间距的步幅为字节数而非元素数目**。

## 6.3 索引数据类型 INDEX

### 6.3.1 MPI_Type_create_hindexed 函数

MPI_Type_indexed 函数目前还在用，马上被替换了。区别在于传递的两个数组原来传递的是指针的形式，现在是传数组名字了；原来是以元素数目为单位，现在以字节为单位。

int MPI_Type_create_hindexed(
int count,//[传入] 块数
int array_of_blocklengths[],//[传入] 每个块的元素数数组
int array_of_displacements[],//[传入] 每个块的间距数组 (步幅，字节为单位)，同上
MPI_Datatype oldtype,//[传入] 每个元素的 MPI 数据类型
MPI_Datatype *newtype//[传出] 新数据类型名字
);

- 功能：定义由不指定大小指定数量的块组成的新数据类型。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**块内元素类型相同、块内元素连续；块之间的大小无需相等、间距无需相等、元素类型相等**。

  原理是复制一个旧数据类型到含有大小不一定相等的块的空间，第 i 个块内旧数据类型的拷贝数目为 array_of_blocklength[i] 个，第 i 个块间距为 array_of_displacements[i](字节为单位)。

## 6.4 结构体数据类型 STRUCT

### 6.4.1 MPI_Type_create_struct 函数

MPI_Type_create_struct(
int count,//[传入] 块数
int array_of_blocklengths[],//[传入] 每个块的元素数数组
int array_of_displacements[],//[传入] 每个块的间距数组 (步幅，字节为单位)，同上
MPI_Datatype array_of_types[],//[传入] 每个块的元素 MPI 数据类型数组
MPI_Datatype *newtype//[传出] 新数据类型名字
);

- 功能：定义一个新的数据类型，每个数据块具有指定的数据类型、位移和大小。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**块内元素类型相同、块内元素连续；块之间的大小无需相等、间距无需相等、元素类型无需相等**。

  原理是复制若干种个旧数据类型到一个块序列中 (每个块内旧数据类型相同)，块之间数据类型可以不同，第 i 个块内旧数据类型的拷贝数目为 array_of_blocklength[i] 个，第 i 个块间距为 array_of_displacements[i]。
- 其他几种数据类型都可以通过 struct 得到。

### 6.4.2 派生数据类型使用

比如将数组的不同位置的地址赋值给派生数据类型，就可以实现在数组的基础上将数组划分为若干需要的部分，从而方便管理，并且加速处理。

## 6.5 数据类型辅助函数

### 6.5.1 MPI_Type_commit 函数

```
int MPI_Type_commit(
    MPI_Datatype *datatype //创建的数据类型
);
```

功能：提交自定义的数据类型，定义的数据类型必须先提交才能使用。

### 6.5.2 MPI_Type_free 函数

```
int MPI_Type_free(
    MPI_Datatype *datatype //要释放的数据类型
);
```

功能：释放数据类型。

### 6.5.3 MPI_Type_get_extent 函数

```
int MPI_Type_get_extent(
    MPI_Datatype datatype,//需判断长度的数据类型
    MPI_Aint *lb,//[传出] 返回最小数据类型长度 (字节)
    MPI_Aint *extent//[传出] 返回数据类型的范围 (字节)
);
```

功能：获取数据类型的最小数据类型长度和长度。

最小数据类型长度即下界；数据类型的范围即跨度。

### 6.5.4 MPI_Address 函数

```
int MPI_Address(
    void* location,//[传入] 调用者的内存位置
    MPI_Aint *address//[传出] 位置的对应起始地址 (字节)
);
```

功能：获取内存中某个位置的地址，字节为单位。

虽然在向量、索引数据类型的情况下，间距可以通过数组内元素序号的相减得到，但在多种数据组成的结构体数据类型的情况下，间距是无法直接得到的，因此需要通过这个函数来分别得到两个需要的块的起始位置的绝对值并相减，从而得到间距。第一个块直接是 0 即可。

## 6.6 特殊数据类型与绝对原点

### 6.6.1 派生数据类型的大小与延伸

对于一个定义好了的派生数据类型，当以其来定义某开辟后的空间的数据类型时，该数据类型的数据就在空间内从起始位置开始，按照数据类型定义的规则存在，包括**包括块数目、大小、间距，块内的数据类型**等。当该数据类型的数据大小大于 1 时，则在第一个类型的结尾处 (即与起始位置距离为跨度大小的位置) 继续取处数据类型的第二个数据。

称数据类型的延伸尺寸为跨度。

### 6.6.2 MPI_UB 和 MPI_LB

- 其为大小为 0 字节的特殊数据类型，会不增加原有的数据的长度。
- 其能够改变已有数据类型的跨度，UB 使跨度延伸上界的长度，LB 使跨度延伸下界的长度。
- 可用于 vector 数据类型下传递对角块矩阵，需要派生数据类型下一次取的数据在原跨度的基础上再**增加一个步幅**的长度 (称为相对位移)，即可将定义好的数据类型与 MPI 定义的数据类型 MPI_UB，一起构成一个新的数据类型，那么就可以按这样的跨度传递特殊数据类型了。
- **需要在一开始构建 vector 的时候一块构建，UB 指的是相邻块头到头的间距，不可和一个构建好的 vector 构建，否则偏移长度会太大**。
- 也可以直接通过 index 创建块非等距的数据类型，但其大小为 1 且无法变大小。
- 注意，派生数据类型及缓冲区中所有的传入起始地址均为数据的地址，不可以为地址的地址。

### 6.6.3 绝对原点

MPI_BOTTOM：派生类型起始地址，称为绝对原点。在 MPI_Address 中返回的地址就是是相对于绝对原点的。

在不使用相对位移的情况下，发送接收中可以使用 MPI_BOTTOM 作为发送和接收数据的首地址 (反正和数据的地址都是一样的)。

## 6.7 数据的打包与拆包

对于不同数据类型的数据和非连续的数据来说，可以采用之前的函数来新建一个数据类型，从而将这些数据做为整体一次发送。也可以将这些数据打包到一块，然后再发送，然后在接收端接收后再解包分开。

打包和解包操作是为了发送不连续的数据在发送前显式地把数据包装到一个连续的缓冲区再一次发送，在接收之后从连续缓冲区中解包。

### 6.7.1 MPI_Pack 函数

```
int MPI_Pack(
    void *inbuf,//[传入] 指向待打包数据的指针 (可选数据类型)
    int incount,//[传入] 要打包数据个数，可为空 (则默认全部的某类型数据)
    MPI_Datatype datatype,//[传入] 要打包的数据类型
    void *outbuf,//[传出] 打包后数据的位置 (即缓冲区)(可选数据类型)
    int outsize,//[传入] 缓冲区大小 (字节)，可为空 (则自动分配大小)
    int *position,//[传入传出] 缓冲区中第一个用于打包的位置 (地址偏移量)(字节)
    MPI_Comm comm //[传入] 当前通讯子
);
```

- 功能：单个函数是将某个位置的若干个某种数据类型的数据打包到一个连续的内存空间位置。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：

- – 在发送数据前该函数可以多次使用，将不同的类型的数据打包到同一个内存。
  – 定义的 *outbuf 必须足够大，以存放全部被包入的数据，且不用每次打包时修改。
  – 定义的 *position 在每次打包时会自动更新偏移量的值，不用每次打包时修改。

### 6.7.2 MPI_Unpack 函数

int MPI_Pack(
void *inbuf,//[传入] 指向待解包缓冲区的指针 (可选数据类型)
int insize,//[传入] 缓冲区大小 (字节)，可为空 (则自动获得大小)
int *position,//[传入传出] 缓冲区中第一个用于打包的位置 (地址偏移量)(字节)，
void *outbuf,//[传出] 解包后数据的位置 (可选数据类型)
int outcount,//[传入] 解包元素个数，可为空 (则默认全部该类型数据)
MPI_Datatype datatype,//[传入] 要解包的数据类型
MPI_Comm comm //[传入] 当前通讯子
);

- 功能：将某个位置的打包数据中的某种数据类型的数据解压缩到某个连续内存中。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  – 在接收数据后该函数可以多次使用，直到全部解包，将同一个内存内不同类型的数据解包到指定位置。
  – 定义的 *outbuf 要能够容纳对应的解压数据。
  – 定义的 *position 在每次打包时会自动更新偏移量的值，不用每次解包时修改。

### 6.7.3 MPI_Pack_size 函数

```
int MPI_Pack_size(
    int incount,//[传入] 数据个数
    MPI_Datatype datatype,//[传入] 数据类型
    MPI_Comm comm,//[传入] 当前通讯子
    int *size//[传出] 输出所需尺寸 (字节)
);
```

- 功能：计算打包某种类型数据所需的缓冲区大小。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：同样只能处理一种数据类型的大小，不可累加。

# 7 聚合通信

与点到点通信不同，其是两个进程之间的通信。而在聚合通信下，通讯子内所有的进程均参与通讯。

## 7.1 障碍同步 MPI_Barrier

```
int MPI_Barrier(
    MPI_Comm //[传入] 进程所在通讯子
);
```

- 功能：线程运行到此函数时等待，直到通讯子内所有进程都运行到该函数时再一起返回，继续运行。用于将一个通讯子内所有的进程同步。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：无。

## 7.2 广播 MPI_Bcast

```
int MPI_Bcast(
    void* buffer,//[传入] 发送缓冲区地址 (要发送的数据，任意类型)
    int count,//[传入] 数据个数
```

MPI_Datatype datatype,//[传入] 数据类型
int root,//[传入] 发送广播的序列号 (根进程)
MPI_Comm comm //[传入] 进程所在通讯子
);

- 功能：将序列号为 root 的进程的数据广播到通讯子 comm 内所有进程 (包括本身)。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：为阻塞式广播，全部接收后才返回。
- 接收方每个进程接收的数据是完全一样的。

### 7.3 收集 MPI_Gather

MPI_Gather(
void* send_data,//[传入] 该进程需要聚合的数据
int send_count,//[传入] 该进程内要聚集的数据个数 (必须相等)
MPI_Datatype send_datatype,//[传入] 聚集的数据类型
void* recv_data,//[传出] 聚合后数据的存放位置
int recv_count,//[传入] 根进程从该进程接收的数据长度 (必须相等)
MPI_Datatype recv_datatype,//[传入] 接收的数据类型
int root,//[传入] 聚集数据汇入的 (根进程) 号
MPI_Comm communicator //[传入] 进程所在通讯子
);

- 功能：将一个进程组 (通讯子) 内所有的的发送缓冲区内参数 (包括自身的) 根据发送这些数据的进程的序列号将它们依次存放到自已的消息缓冲区中。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 效果等同于一个组中的 n 个进程 (包括根进程在内) 都执行了一个 send，同时根进程执行了 n 次 recv。

  - **每一个函数都包括了发送和接受部分，只不过只有根进程接收。**
  - 每个进程都需要调用一次该函数。
  - 在每个进程和根进程之间，发送的数据量必须和接收的数据量相等，但发送方和接收方之间的不同数据类型映射仍然是允许的。
  - 是阻塞的。
  - MPI_Gatherv 函数：与 MPI_Gather 意义相同，但多了一个 int *displs[] 参数，其允许各进程传入大小不相等的数据。
  - 接收的数据是按进程顺序依次存放在 root 进程的接收缓冲区内的。
  - MPI_Allgather 函数：与 MPI_Gather 意义相同，少一个 root 参数，但此时是所有的进程都将接收结果，而不是只有根进程接收结果。

### 7.4 散播 MPI_Scatter

```
int MPI_Scatter(
    void* sendbuf,//[传入] 要发送的数据
    int sendcount, //[传入] 要发送的元素个数
    MPI_Datatype sendtype,//[传入] 数据类型
    void* recvbuf,//[传出] 接收消息的接收缓冲区地址
    int recvcount, //[传入] 接收的元素个数
    MPI_Datatype recvtype,//[传入] 接收的元素类型
    int root, //[传入] 发送进程的 (根进程) 号
    MPI_Comm comm //[传入] 进程所在通讯子
);
```

- 功能：将某个进程的发送缓冲区内参数发送给本进程组内的全部进程 (包括自身的) 的消息缓冲区中。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 效果等同于根进程执行了 n 次 send 操作，同时通讯子内每个进程执行一次 recv 操作。

- **每一个函数都包括了发送和接受部分，只不过只有根进程发送。**
- 每个进程都需要调用一次该函数。
- 发送的数据个数必须和接收的数据个数相等，但发送方和接收方之间的不同数据类型映射仍然是允许的。而相似函数 MPI_Scatterv 函数则允许向各进程发送大小不相等的数据。
- 是阻塞的。
- 与 MPI_Bcast 不同，root 进程发送后，每个进程接收的数据是不一样的，按顺序分别得到 sendbuf 的每一个元素。
- MPI_ALLGATHER：可使用 MPI_Alltoall 替代。

## 7.5 全交换 MPI_Alltoall

```
int MPI_Alltoall(
    constvoid *sendbuf,//[传入] 要发送的数据
    int sendcount,//[传入] 要发送的元素个数
    MPI_Datatype sendtype,//[传入] 数据类型
    void *recvbuf, //[传出] 接收消息的接收缓冲区地址
    int recvcount,//[传入] 接收元素个数
    MPI_Datatype recvtype,//[传入] 接收的元素类型
    MPI_Comm comm//[传出] 进程所在通讯子
);
```

- 功能：每个进程都可以向其他若干的进程按进程次序发送数据，又同时按进程次序接受其他节点的数据 (均包括自身)。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 每个进程发给其他进程的内容都是不同的，就是将各自发送缓冲区里的内容按序发给其他进程，每一个进程只会发送给另一个进程一个元素、而仅从另一个进程接受一个元素。
  - **发送元素个数 n 如果小于进程数，则只会将数据中的 n 个元素按进程号顺序发送至 0 到 n-1 个进程，同理接收。**

- 相对于每个进程都执行 sendcount 次 send，然后每个进程执行 recvcount 次 recv。
- 在每个进程和根进程之间，发送的数据量必须和接收的数据量相等 (都是指发给每个线程/接受每个线程的元素的数目，不是总的)，但发送方和接收方之间的不同数据类型映射仍然是允许的。
- 是阻塞的。

# 8 规约操作

指的是这种函数调用一次，则在该通讯子内所有的进程都安装运算的要求来操作。使用者可以不考虑这方面的并行实现，直接调用函数即可。

相当于一个函数同时完成了对各个进程参数的收集并计算，以及将结果输出回线程。避免了先收集再统一计算再返回的并行问题。

## 8.1 规约

### 8.1.1 规约 MPI_Reduce

```
int MPI_Reduce(
    void* sendbuf, //[传入] 发送的消息起始地址
    void* recvbuf, //[传出] 接收消息的起始地址
    int count,//[传入] 发送的消息数据个数
    MPI_Datatype datatype,//[传入] 发送消息的数据类型
    MPI_Op op, //[传入] 归约操作符 (句柄)
    int root,//[传入] 接收消息的 (根进程) 序列号
    MPI_Comm comm //[传入] 当前进程通讯子
);
```

- 功能：将组内每个进程输入缓冲区中的数据直接按 op 操作组合起来，并将其结果返回到序列号为 root 的进程的输出缓冲区中。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：

- 每个进程都需要调用该函数，所有组成员都用同样的参数 count、datatype、op、root 和 comm 来调用此函数，其所提供的所有进程的输入和输出缓冲区都满足长度相同、元素类型相同。
- 注意个数是指每个进程内上传的，而非总的，个数是几就进行几次 op 操作。
- 发送消息的数据类型不一定要与输入的数据相同，但必须和 op 操作函数匹配，而输出会与定义的类型相同。
- 是阻塞的。
- 用户也可以定义自己的作用于几种数据类型的操作 (op)，即可以是基本的也可以是派生的。
- 操作 op 始终满足结合律，且所有 MPI 定义的操作满足交换律，而用户自定义的操作是不满足交换律的。**(区别在于满足交换律，则在规约计算时计算顺序可以变，而非固定地按进程序列号升序方式进行，即从序列号为 0 的进程开始)**

### 8.1.2 规约广播 MPI_Allreduce

```
int MPI_Alleduce(
    void *input_data, //[传入] 发送的消息起始地址
    void *output_data, //[传出] 接收消息的起始地址
    int count,//[传入] 发送的消息数据个数
    MPI_Datatype datatype,//[传入] 数据类型
    MPI_Op operator,//[传入] 规约操作函数
    MPI_Comm comm //[传入] 当前进程通讯子
);
```

- 功能：类似于 MPI_Reduce，但所有进程均**接收相同的信息**而不止 root 进程。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：等价于先进行一次规约操作再进行一次广播。

### 8.1.3 前缀扫描 MPI_Scan

int MPI_Scan(
void* sendbuf, //[传入] 发送的消息起始地址
void* recvbuf, //[传出] 接收消息的起始地址
int count,//[传入] 发送的消息数据个数
MPI_Datatype datatype,//[传入] 数据类型
MPI_Op op,//[传入] 规约操作函数
MPI_Comm comm //[传入] 当前进程通讯子
);

- 功能：类似于 MPI_Reduce，但所有进程均接收信息而不止 root 进程。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 常用于对分布于组中的数据作前置归约操作。
  - 与 MPI_Allreduce 不同，对于第 i 个线程，其接收的结果为序号小于等于 i 的线程上传的数据之和，即进程号越大，参与 op 运算的数据就越多。
  - MPI 的归约和扫描操作允许每个进程贡献向量值，而不只是标量值。

### 8.1.4 规约散播 MPI_Reduce_scatter

int MPI_Reduce_scatter(
void* sendbuf, //[传入] 发送的消息起始地址
void* recvbuf,//[传出] 接收消息的起始地址
int *recvcounts,//[传入] 整数数组，各进程接收规约结果的元素个数。**所有进程的该数组都必须相同**
MPI_Datatype datatype, //[传入] 数据类型
MPI_Op op, //[传入] 规约操作函数
MPI_Comm comm //[传入] 当前进程通讯子
);

- 功能：将各个线程的发送缓冲区值聚集起来，完成 op 之后，按 recvcounts 将结果返回给各线程。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 与 MPI_Alleduce 不同之处在于，对于 op 输出的结果，操作的结果被按通讯子中进程的数量切割成 i 个**不重叠**的部分，第 i 个部分被发送到其接收缓冲区中的第 i 个进程。
  - 每个线程读取的元素个数为 recvcounts[i]，计算得到的结果个数则为 $max\{recvcounts[i]\}$，并将结果依据 recvcounts[i] 分割并按序列号顺序分配至各线程。
  - 一共进行 $max\{recvcounts[i]\}$ 次 op 操作，对于第 recvcounts[i] 个的 op 操作，如果某个线程不含对应输入缓冲区的参数，则不纳入 op 计算中。
  - 等价于：一个 MPI_Reduce 操作，后面跟一个 MPI_Scatterv 操作，但直接实现运行得更快一些。

## 8.2 运算种类与可用数据类型

### 8.2.1 MPI 定义的归约操作函数

| 名字 | 含义 | 允许数据类型 |
|---|---|---|
| MPI_MAX | 最大值 | C 整数、浮点数 |
| MPI_MIN | 最小值 | C 整数、浮点数 |
| MPI_SUM | 求和 | C 整数、浮点数、复数 |
| MPI_PROD | 求积 | C 整数、浮点数、复数 |
| MPI_LAND | 逻辑与 | C 整数、逻辑型 |
| MPI_BAND | 按位与 | C 整数、逻辑型 |
| MPI_LOR | 逻辑或 | C 整数、逻辑型 |
| MPI_BOR | 按位或 | C 整数、逻辑型 |
| MPI_LXOR | 逻辑异或 | C 整数、逻辑型 |
| MPI_BXOR | 按位异或 | C 整数、逻辑型 |
| MPI_MAXLOC | 最大值且相应位置 | 复合数据类型 |
| MPI_MINLOC | 最小值且相应位置 | 复合数据类型 |

**基本数据类型组参考**

| | |
|---|---|
| C语言中的整型 | MPI_INT MPI_LONG MPI_SHORT MPI_UNSIGNED_SHORT MPI_UNSIGNED MPI_UNSIGNED_LONG |
| Fortran语言中的整型 | MPI_INTEGER |
| 浮点数 | MPI_FLOAT MPI_DOUBLE MPI_REAL MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE |
| 逻辑型 | MPI_LOGICAL |
| 复数型 | MPI_COMPLEX |
| 字节型 | MPI_BYTE |

### 8.2.2 复合数据类型

针对 op 操作中的 MPI_MINLOC 操作符和 MPI_MAXLOC 操作符对应的数据类型。

MPI_MINLOC 操作符用于计算全局最小值和这个最小值的索引号；

MPI_MAXLOC 操作符用于计算全局最大值和这个最大值的索引号；

与规约函数一起使用，可计算一个全局最小值 (最大值) 和这个值所在的进程序列号。

**数据类型特点：**

- 其数据类型为两个数的组合 $(w, k)$，前者为值，后者为值所在进程序号；
- 在 MPI_MINLOC 的情况下，如果存在最小值相等的情况，则 k 取进程序号的最小值；
- 在 MPI_MAXLOC 的情况下，如果存在最大值相等的情况，则 k 取进程序号的最小值；
- 传入规约函数 (即给 op 函数) 的数据类型也应该是复合数据类型；
- 传入的数据也应该满足第一个值为要比较的值第二个为进程序号，类型为二维数组结构体皆可，传出也是写到类似的数据结构中，**传入传出缓冲区不要求为复合数据类型**。

**复合数据类型分类**

| 名字 | 描述 |
| --- | --- |
| MPI_FLOAT_INT | 浮点型和整型 |
| MPI_DOUBLE_INT | 双精度和整型 |
| MPI_LONG_INT | 长整型和整型 |
| MPI_2INT | 整型值对 |
| MPI_SHORT_INT | 短整型和整型 |
| MPI_LONG_DOUBLE_INT | 长双精度浮点型和整型 |

### 8.2.3 自定义规约操作函数

**MPI_Op_create 函数**

```
int MPI_Op_create(
    MPI_User_function *function,//[传入] 用户自定义的函数 (函数)
    int commute,//[传入] 可交换属性，是则为 true, 否则为 false
    MPI_Op *op //[传出] 操作名
);
```

- 功能：将用户自定义的操作和 op 绑定在一起，从而可以被规约函数调用。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 可以用于 8.1 的规约函数中。
  - 默认是满足结合律的，交换律根据 commute 属性确定。

**用户自定义规约操作函数的描述方法：**

```
typedef void MPI_User_function(
    void *invec, //[传入] 传入函数的数组数据，要操作的数据，每一个值来自不同的进程
    void *inoutvec,//[传出] 输出的数据数组，如果返回每一个进程按进程号元素依次返回
    int *len, //[传入] 是传入传出数据数组的大小，一般等于进程数目
    MPI_Datatype *datatype,//[传入] 数据类型
);
```

- 必须具备四个参数: invec, inoutvec, len 和 datatype。
- 也可以对 inoutvec 传入数据并计算，至少其在函数输出时会被覆盖。

**MPI_Op_free 函数**

```
int MPI_op_free(
    MPI_Op *op//[传入] 用户自定义的归约操作
);
```

- 功能：将用户自定义的归约操作撤消。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：自定义操作完毕后最好调用撤销。

# 9 组操作

类似于数学中的集合计算，通过对组进行一系列操作，达到构建不同用途的通讯子的目的 (通讯子操作比较难做到)。

## 9.1 进程组创建

### 9.1.1 MPI_Comm_group 函数

```
int MPI_comm_group(
    MPI_Comm comm,//[传入] 组所基于的通讯子
    MPI_Group *group //[传出] 通讯子对应的组 (句柄)
);
```

- 功能：用来建立一个通信子对应的新进程组 (句柄)，之后就可以对此进程组进行需要的操作组。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 本地操作。不依赖其他进程。
  - 不同的进程可以定义不同的组。一个进程可以定义不包含自身的组，下同。

– 不提供从头开始构建组的机制，必须从现有组构建组。定义所有其他组的基础组是与初始通讯子 MPI_COMM_WORLD 关联的组，下同。

### 9.1.2 MPI_Group_union 函数

```
int MPI_Group_union(
    MPI_Group group1,//[传入] 已有的组 1
    MPI_Group group2,//[传入] 已有的组 2
    MPI_Group *newgroup //[传出] 新的组
);
```

- 功能：从两个现有组的并集创建新组。(全算上)
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**次序是第二组跟在第一组后面**。

### 9.1.3 MPI_Group_intersection 函数

```
int MPI_Group_intersection(
    group1,//[传入] 已有的组 1
    MPI_Group group2,//[传入] 已有的组 2
    MPI_Group *newgroup //[传出] 新的组
);
```

- 功能：从两个现有组的交集创建新组。(共同的)
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**次序是交集中元素次序同第一组**。

### 9.1.4 MPI_Group_difference 函数

```
MPI_Group_difference(
    group1,//[传入] 已有的组 1
    MPI_Group group2,//[传入] 已有的组 2
    MPI_Group *newgroup //[传出] 新的组
);
```

- 功能：从两个现有组的差集创建新组。**(在第一组但不在第二组中的所有元素)**
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：**差集中的元素次序同第一组**。

#### 9.1.5 MPI_Group_incl 函数

int MPI_Group_incl(
MPI_Group group,//[传入] 已有的组
int n,//[传入] 进程组组中选取的进程数目
int *ranks,//[传入] 将在新进程组中出现的旧进程组中的编号
MPI_Group *newgroup //[传出] 由 ranks 定义的顺序导出的新进程组
);

- 功能：从一个已有的进程组中，选择一部分进程导出新的进程组。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：新进程组中进程的次序来源于老进程组中的次序。

#### 9.1.6 MPI_Group_excl 函数

int MPI_Group_excl(
MPI_Group group,//[传入] 已有的组
int n,//[传入] 进程组组中剔除的进程数目
int *ranks,//[传入] 在新进程组中不出现的旧进程组中的编号
MPI_Group *newgroup//[传出] 旧进程组中不在 ranks 里的元素组成的新进程组
);

- 功能：从一个已有的进程组中，剔除一部分进程剩余进程组成的新的进程组。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：次序来源于老进程组。

### 9.1.7 MPI_Group_range_incl 函数

int MPI_Group_range_incl(
MPI_Group group,//[传入] 已有的组
int n,//[传入] 进程组中选取的三元组数数目
int ranges[][3],//[传入] 将在新进程组中出现的旧进程组中的编号
MPI_Group *newgroup //[传出] 由 ranks 定义的顺序导出的新进程组
);

- 功能：类似于 MPI_Group_incl 函数，但在选取时是根据三元组数来选取的。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：
  - 三元组整体：二维数组，可视为若干个大小为 3 的一维数组，每个一维数组格式为 $first_i, last_i, stride_i$。分别表示起始进程序号，结束进程序号，步长 (分割块数)。
  - 共计选取 n 个这样的三元组，则抽取的进程序号为：
    $first_i, first_i + j * \frac{last_i - first_i}{stride_i}, (0 < j <= stride_i), last_i, (0 <= i < n)$
    共计 $\sum_{i=0}^{i<n}(stride_i + 1)$ 个数据。
  - 注意选取的进程序号不能重复，否则报错。
  - 步长可以为负数，相对应的 $first_i < last_i$。

### 9.1.8 MPI_Group_range_excl 函数

int MPI_Group_range_excl(
MPI_Group group,//[传入] 已有的组
int n,//[传入] 进程组组中剔除的三元组数目
int *ranges[][3],//[传入] 在新进程组中不出现的旧进程组中的编号
MPI_Group *newgroup//[传出] 旧进程组中不在 ranks 里的元素组成的新进程组
);

- 功能：类似于 MPI_Group_excl 函数，但在剔除时是按照三元组的规律来剔除的。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：参考 MPI_Group_range_incl 的三元组相关。

## 9.2 进程组管理

### 9.2.1 MPI_Group_size 函数

```
int MPI_Group_size(
    MPI_Group group,//[传入] 要判断大小的组
    int *size //[传出] 进程数目
);
```

- 功能：得到组内进程的数目。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：无。

### 9.2.2 MPI_Group_rank 函数

```
int MPI_Group_rank(
    MPI_Group group,//[传入] 要获得序号的所在组的名字
    int *rank//[传出] 进程序号
);
```

- 功能：获得当前进程在指定进程组内的进程序号。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：当前进程需要在组内。如果进程不是进程组中的成员，则返回值 RANK 为 MPI_UNDEFINED

### 9.2.3 MPI_Group_translate_ranks 函数

```
int MPI_Group_translate_ranks(
    MPI_Group group1,//[传入] 进程组 1
    int n,//[传入] 要查询的进程的个数
    int *ranks1,//[传入] 进程组 1 中有效编号组成的数组
    MPI_Group group2,//[传入] 进程组 2
    int *ranks2 //[传出]ranks1 中的元素在进程组 2 中的对应编号
);
```

- 功能：在已知若干个进程在进程组 1 中的序号，通过此函数确定相同的进程在进程组 2 中的序号。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：如果这个进程不属于进程组 2，则在 ranks2 中对应 ranks1 的位置返回值为 MPI_UNDEFINED。

### 9.2.4 MPI_Group_free 函数

```
int MPI_Group_free(
MPI_Group *group//[传入] 派生的组
);
```

- 功能：释放一个定义的进程组。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。
- 备注：无。

### 9.2.5 MPI_Group_compare 函数

```
int MPI_Group_compare(      MPI_Group group1,//[传入] 进程组 1
    MPI_Group group2,//[传入] 进程组 2
    int *result//[传出] 比较结果
);
```

- 功能：比较两个组的成员特性。
- 返回值：成功时返回 MPI_SUCCESS，否则返回错误代码。

- 备注：result 的值：
  - MPI_IDENT：两个通讯子具有同样的进程组（各进程组的组成员和序列号次序都相同）和相同的上下文，即为同一对象的句柄；
  - MPI_CONGRUENT：两个通讯子具有同样的进程组，但上下文不相同；
  - MPI_SIMILAR：两个通讯子的组成员相同但序列号次序不同；
  - MPI_UNEQUAL：两个通讯子组成员也不相同。

# 10 附录

## 10.1 数据、指针、与地址

- 变量名、数组名表示一个或一组数据，是有类型的，其中数组的类型由元素的类型和数组长度共同构成。
- 指针指向变量的地址。
- 指针的类型实际上指的是其指向某种数据的类型，其本身无类型，都是指向某一个地址。因此 sizeof 指针得到的是指针大小，而数组名则是数组大小。

  **指针的类型是用来在从地址取数据时，判断每一个元素需要提取多少字节（不同类型数据的元素长度不一样）**。

  如果对指针（即已指向变量的地址）进行类型转化，则可能由于新的类型过大导致溢出、或者新的类型小导致数据变化。所以只能对数据直接进行转化。
- 数组名表示的是数组的起始地址，不是一个指针。
- 向函数传递参数时，可以通过传入数据的地址来进行，但计算机无法分辨传入的 type 类型的地址对应的值是 type 类型的值，还是另一个地址，只能通过程序员编写时注意。而如果采用指针，则会明确的指出是一个 type 类型变量的地址 (*) 还是另一个 type 类型指针的地址 (**)。

- 指针的解引用：若定义指针 int * p；赋地址后，通过 *p 即可得到地址对应的值，称为解引用。(指针类型就是在这种情况下确定从地址读取多少字节数据的)
- 变量取地址：&。
- 一级指针指向的必须是变量的地址，二极及以上指针指向的必须是另一个指针的地址。
- void 型指针：表示通用指针，可以用来存放任何数据类型的引用。直接对其进行数据类型转化不会导致数据丢失或溢出。