第六讲：MPI进程组与通信器

陈俊清

jqchen@math.tsinghua.edu.cn

清华大学数学科学系

April 12, 2013

Part I

进程组与通信器

Outline

Outline

进程组

进程组是一组进程的有序集合，它定义了通信器中进程的集合及进程的序号。组中的每个进程都有一个唯一的rank加以标识。 组对象存在于通信子环境内，为通信器定义和描述通信参与进程，并提供表示和管理进程的若干功能。

MPI提供两个预定义的进程组句柄——MPI_GROUP_EMPTY和MPI_GROUP_NULL，前者表示由空进程组集合构成的进程组， 后者表示非法进程组。

进程组的句柄是进程所固有的，只对本进程有意义，因此进程组的句柄通过通信在进程间传递是无意义。

上下文（Context）

上下文是通信器的一个固有性质，它为通信器划分出特定的通信空间。消息都是在一个给定的上下文中进行传递。 不同上下文间不允许进行消息的收发，这样可以确保不同通信器中的消息不会混淆。
此外，MPI通常也要求点对点通信与聚合通信是独立的，它们间的消息不会互相干扰。
上下文对用户是不可见的，它是MPI实现中的一个内部概念。

通信器（Communicator）

- 域内通信器（Intracommunicator）： 域内通信器由进程组和上下文构成。一个通信器的进程组中必须包含定义该通信器的进程作为其成员。 此外，为了优化通信以及支持处理器实际的拓扑连接方式，通信器中还可以定义一些附加属性，如进程间的拓扑联接方式等。 域内通信器可用于点对点通信，也可以用于聚合通信。MPI预定义的域内通信器包括MPI_COMM_WORLD,MPI_COMM_SELF和MPI_COMM_NULL，其中MPI_COMM_NULL表示非法通信器（空通信器）。
- 域间通信器（Intercommunicator）：用于分属不同进程组的进程间的点对点通信。 一个域间通信器由两个进程组构成。域间通信器不能定义进程的拓扑联接信息，也不能用于聚合通信。

Outline

查询进程组大小和进程在组中的序号

```
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_rank(MPI_Group group, int *rank)
```

上述两个函数分别返回给定进程组的大小（包含的进程个数）和进程在其中的序号。 他们与MPI_Comm_size和MPI_Comm_rank完全类似。

两个进程组间进程序号的映射

```
int MPI_Group_translate_ranks(MPI_Group group1, int n,
        int *ranks1, MPI_Group group2, int *ranks2)
```

该函数给出进程组group1中的一组进程序号ranks1所对应的进程在进程组group2中的序号ranks2。 n给出进程组ranks1和ranks2中序号的个数。

比较两个进程组

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2,
        int *result)
```

比较两个进程组。返回时，如果两个进程组包含的进程及他们的序号完全一样则result=MPI_IDENT, 如果两个进程组包含的进程一样但序号不同，则result=MPI_SIMILAR,否则result=MPI_UNEQUAL.

进程组的创建与释放

获取通信器中的进程组:

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

在group参数中返回指定通信器的进程组。

进程组的创建与释放

进程组的并集:

```
int MPI_Group_union(MPI_Group group1,MPI_Group group2,
        MPI_Group *newgroup)
```

newgroup返回group1与group2的并集构成的进程组。 新进程组中进程序号的分配原则是先对属于group1的进程按group1的序号编号， 再对属于group2但不属于group1的进程按group2的序号编号。

进程组的创建与释放

进程组的交集:

```
int MPI_Group_intersect(MPI_Group group1,
        MPI_Group group2, MPI_Group *newgroup)
```

newgroup返回group1与group2的交集构成的进程组。新进程中进程的序号按group1的序号进行编排。

进程组的创建与释放

进程组的差集:

```
int MPI_Group_difference(MPI_Group group1,
        MPI_Group group2, MPI_Group *newgroup)
```

newgroup返回的新进程组由属于 group1但不属于 group2的进程构成，序号按group1中的序号进行编排。

进程组的创建与释放

进程组的子集:

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
        MPI_Group *newgroup)
```

新进程组newgroup由老进程组group中用数组ranks给出序号的进程组成， n是数组 ranks的元素个数。 新进程组中进程的序号亦由 ranks数组决定，即进程组 newgroup中序号为 i的进程是老进程组 group中序号为 ranks[i]的进程。
该函数也可以用来对进程组中的进程进行重新排序。当参数n=0时将创建一个空进程组 MPI_GROUP_EMPTY.

进程组的创建与释放

进程组减去一个子集：

```
int MPI_Group_excl(MPI_Group, int n, int *ranks,
        MPI_Group *newgroup)
```

该函数将一个进程组进程的集合减去其中的一个子集而得到一个新进程组。减去的进程号由数组ranks给出，n给出减去的进程数。 新进程组的序号按进程在原进程组中的序号排列得出。（参看程序例子：ex6-0.c）

进程组的创建与释放

进程号范围构成的子集:

```
int MPI_Group_range_incl(MPI_Group group, int n,
        int ranges[][3],MPI_Group *newgroup)
```

将由一组进程序号构成的子集组成一个新进程组。每个范围由一个三元数对（起始序号，终止序号，步长）描述，n为范围个数。 即构成新进程组的进程集合由原进程组中序号属于下述集合的进程构成:

$$\{r| \quad r = ranges[i][0] + k * ranges[i][2],$$
$$k = 0, \ldots, \lfloor \frac{ranges[i][1] - ranges[i][0]}{ranges[i][2]} \rfloor, i = 0, \ldots, n-1\}$$

上式中所有算出的r必须互不相同，否则调用出错。新进程组中进程的序号按上式中先k后i的顺序编排。

进程组的创建与释放

进程序号范围构成的子集的补集:

```
int MPI_Group_range_excl(MPI_Group group, int n,
        int ranges[][3],MPI_Group *newgroup)
```

将老进程组减去由一组进程序号范围给出的进程而得到一个新的进程组，它相当于取函数MPI_Group_range_incl给出的子集的补集。 各参数含义与MPI_Group_range_incl类似。

进程组的创建与释放

进程组的释放:

```
int MPI_Group_free(MPI_Group *group)
```

函数返回时会将group置成MPI_Group_NULL以防以后被误用。实际上，函数只是将该进程组加上释放标志。 只有基于该进程组的所有通信器均已被释放后才会真的将其释放。

Outline

比较两个通信器

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2,
        int *result)
```

比较两个（域内）通信器。返回时，如果两个通信器代表同一个通信域（两个通信器中的进程可以互相通信） 则result = MPI_INDENT;如果两个通信器不代表同一个通信域，但他们的进程组相同，即他们包含相同的进程且进程的序号也相同， 则result=MPI_CONGRUENT;如果两个通信器包含的进程相同但进程的序号不同，则result=MPI_SIMILAR; 否则result=MPI_UNEQUAL。

通信器的创建与释放

新通信器必须在一个已有通信器的基础上创建。调用一个通信器创建函数时，所有属于老通信器的进程必须同时调用该函数。 返回时，属于新通信器的进程得到信通信器的句柄，而不属于新通信器的进程则得到MPI_COMM_NULL。

通信器的创建与释放

通信器的复制:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

生成一个与comm具有完全相同属性的新通信器newcomm。注意，新通信器newcomm与老通信器comm代表着不同的通信域，因此他们不能进行通信操作，即comm发送的消息不能用newcomm来接收，反之亦然。该函数通常用在并行库函数中：在库函数的开头将用户提供的通信器参数复制产生一个新通信器，在库函数中使用新通信器进行通信，而在库函数返回前将新通信器释放，这样可以确保库函数中的通信不会与用户的其它通信相互干扰。(两个通信器的比较：ex6-1.c)

通信器的创建与释放

通信器的创建:

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
        MPI_Comm *newcomm)
```

创建一个包含指定进程组group的新通信器newcomm，这个函数并不将comm的其他属性传递给newcomm， 而是为newcomm建立一个新的上下文。 返回时，属于group的进程中newcomm等于新通信器的句柄，而不属于进程组group的进程中newcomm则等于MPI_COMM_NULL. （参看例子程序：ex6-2.c）

通信器的创建与释放

分裂通信器:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

该函数按照由参数color给出的颜色将通信器中的进程分组，所有具有相同颜色的进程构成一个新通信器。 新通信器中进程的序号按参数key的值排序，两个进程的key值相同时则按照他们在老通信器comm中的序号排序。 返回时，newcomm等于进程所属的新通信器的句柄。color必须是非负整数或MPI_UNDEFINED.如果color = MPI_UNDEFINED。 则表示进程不属于任何新通信器，返回时newcomm=MPI_COMM_NULL.

通信器的创建与释放

通信器的释放:

```
int MPI_Comm_free(MPI_Comm *comm)
```

函数返回时会将comm置成MPI_COMM_NULL以防以后被误用。实际上，该函数只是将通信器加上释放标志。只有所有引用该通信器的操作均已完成后才会真的将其释放。(一个安全通信的例子：ex6-3.c)

Part II

进程拓扑结构

Outline

Outline

进程拓扑结构

进程拓扑结构是(域内)通信器的一个附加属性，它描述一个进程组各进程间的逻辑联接关系。 进程拓扑结构的使用一方面可以方便、简化一些并行程序的编制， 另一方面可以帮助MPI系统更好地将进程映射到处理器以及组织通信的流向，从而获得更好的并行性能。
MPI的进程拓扑结构定义为一个无向图，图中结点（node）代表进程，而边（edge）代表进程间的联接。 MPI进程拓扑结构也被称为虚拟拓扑结构，因为它不一定对应处理器的物理联接。

- 笛卡尔拓扑结构：具有网格形式的进程拓扑结构，该结构中进程可以用笛卡尔坐标来标识。

笛卡尔拓扑结构

创建函数:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
        int *dims,int *periods, int reorder,
        MPI_Comm *comm_cart)
```

该函数从一个通信器comm_old出发，创建一个具有笛卡尔拓扑结构的新通信器comm_cart。
ndims给出进程网格的维数。数组dims给出每维中的进程数。 数组periods则说明进程在各个维上是否具有周期性，即该维中第一个进程与最后一个进程是否相联， 周期的笛卡尔拓扑结构也称为环面（torus）结构，periods[i]=true表明第i维是周期的，否则是非周期的。
reorder指明是否允许在新通信器comm_cart中对进程进程重新排序。 在某些并行机上，根据处理器的物理联接方式及所要求的进程拓扑结构对进程重新排序有助于提高并行程序的性能。

笛卡尔拓扑结构

comm_cart中各维的进程乘积必须不大于comm_old中的进程数，即：

$$\Pi_{i=0}^{ndims-1} dims[i] \leq NPROCS$$

其中NPROCS为comm_old中的进程数。
当$\Pi_{i=0}^{ndims-1} dims[i] < NPROCS$时，一些进程将不属于comm_cart，这些进程的comm_cart参数将返回MPI_COMM_NULL.

笛卡尔拓扑结构

辅助函数:

```
int MPI_Dims_create(int nnodes,int ndims,int *dims)
```

该函数当给定进程数及维数时自动计算各维的进程数，使得他们的乘积等于总进程数，并且各维上进程数尽量接近。 确切地说，给定nnodes和ndims，计算整数dims[i],i=0,. . . ,ndims-1,使得$\Pi_{i=0}^{ndims-1} dims[i] = nnodes$并且dims[i]的值尽量接近。 该函数要求输入时dims中元素的值为非负整数， 并且它仅修改dims中输入值为0的元素。因此用户可以指定一些维的进程数而仅要求计算其它维的进程数。
局限：没有考虑实际数据在各维上的大小。

笛卡尔拓扑结构

将笛卡尔拓扑结构分割成低维子结构:

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
        MPI_Comm *newcomm)
```

该函数将一个具有笛卡尔拓扑结构的通信器comm中指定的维抽取出来， 构成一个具有低维笛卡尔结构（子网格）的新通信器newcomm。数组remain_dims的元素指定相应的维是否被包含在新通信器中，如果remain_dims[i]=true表示子网格包含第i维， 否则子网格不包含第i维。

笛卡尔拓扑结构

查询笛卡尔拓扑结构的维数:

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

MPI_Cartdim_get在ndims中返回通信器comm中的笛卡尔拓扑结构的维数。

笛卡尔拓扑结构

查询笛卡尔拓扑结构的详细信息:

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
        int *periods, int *coords)
```

MPI_Cart_get返回通信器comm的笛卡尔拓扑结构的详细信息。 数组dims，periods和coords分别返回各维的进程数、是否周期及当前进程的笛卡尔坐标。 参数maxdims给出数组dims，periods和coords的长度的上界。

笛卡尔拓扑结构

笛卡尔坐标到进程序号的映射:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

给定一个进程在通信器comm中的笛卡尔坐标coords，该函数在rank中返回进程在comm中的进程号。 如果某维具有周期性，则coords中对应的坐标值允许“越界”，即小于 0 或大于等于该维的进程数。

笛卡尔拓扑结构

进程序号到笛卡尔坐标的映射:

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
        int maxdims, int *coords)
```

给定一个进程在通信器中的进程号rank，该函数在coords中返回进程在comm中的笛卡尔坐标。maxdims给出数组coords的最大长度。

笛卡尔拓扑结构

数据平移（shift）操作中源地址与目的地址的计算:
在一个具有笛卡尔拓扑结构的通信器中经常在一个给定维上对数据进行平移(shift)，如用MPI_Sendrecv将一块数据发送给该维上后面一个进程，同时接收从该维上前面一个进程发送来的数据。MPI提供了一个函数来方便这种情况下目的地址和源地址的计算。

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
        int disp, int *rank_source, int *rank_dest)
```

输入参数direction是进行数据平移的维号($0 \leq direction < ndims$),disp给出数据移动的“步长”（绝对值）和“方向”（正负号）。输出参数rank_source和rank_dest分别是平移操作的源地址和目的地址。

笛卡尔拓扑结构

假设指定维上的进程数为d，当前进程该维的坐标为i，源进程rank_source该维的坐标为i_s，目的进程rank_dest该维的坐标为i_d，如果该维是周期的，则：

$$i_s = i - \text{disp} \bmod d$$
$$i_d = i + \text{disp} \bmod d.$$

否则：

$$i_s = \begin{cases} i - \text{disp}, & \text{if } 0 \le i - \text{disp} < d \\ \text{MPI\_PROC\_NULL}, & \text{otherwise.} \end{cases}$$
$$i_d = \begin{cases} i + \text{disp}, & \text{if } 0 \le i + \text{disp} < d \\ \text{MPI\_PROC\_NULL}, & \text{otherwise.} \end{cases}$$

Outline

一般拓扑结构

创建图（graph）拓扑结构:

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes,
        int *index, int *edges, int reorder,
        MPI_Comm *comm_graph)
```

从通信器comm_old出发，创建一个具有给定的图结构的新通信器comm_graph.

一般拓扑结构

新通信器的拓扑结构由参数nnodes，index和edges描述：nnodes是图的结点数（如果nnodes小于通信器comm_old的 进程数， 则一些进程将不属于新通信器comm_graph,这些进程中参数comm_graph的返回值将为MPI_COMM_NULL）， index[i](i=0,. . . ,nnodes-1)给出结点0,. . . ,i的邻居数之和:
index[0]表示第0个节点的邻居数，第i($i > 0$)个节点的邻居数：$index[i] - index[i-1]$。

一般拓扑结构

edges则顺序给出所有结点的邻居的序号。用Neighbor[i]表示第i个结点的邻居的序号集合，则:

$$\text{Neighbor}(0) = \{edges[j] | 0 \leq j < index[0]\}$$
$$\text{Neighbor}(i) = \{edges[j] | index[i-1] \leq j < index[i]\}$$
$$i = 1, \ldots, nnodes - 1$$

参数reorder指明是否允许在新通信器中对进程重新编号。

一般拓扑结构

查询拓扑结构类型:

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

查询拓扑结构类型。返回时，如果comm具有笛卡尔拓扑结构，
则status=MPI_CART, 如果comm具有图拓扑结构
则status=MPI_GRAPH,否则status=MPI_UNDEFINED.

一般拓扑结构

查询拓扑结构的结点数与边数:

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes,
        int *edges)
```

该函数在参数nnodes 中返回通信器comm的拓扑结构图的结点数（等于comm中的进程数）。 在参数edges中，返回通信器comm的拓扑结构边数。

一般拓扑结构

查询拓扑结构的详情:

```
int MPI_Graph_get(MPI_Comm comm, int maxindex,
        int maxedges, int *index, int *edges)
```

该函数返回通信器comm的拓扑结构中的index和edges数组。参数maxindex和maxedges分别限定数组index和edges的最大长度。

一般拓扑结构

查询指定进程的邻居数:

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank,
        int maxneighbors, int *neighbors)
```

该函数在数组neighbors中返回通信器comm中序号为rank的进程的所有邻居的序号。maxneighbors限定数组neighbors的最大长度。

底层支持函数

查询指定笛卡尔结构下理想的进程编号方式:

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims,
        int *periods, int *newrank)
```

该函数在newrank中返回给定笛卡尔拓扑结构下当前进程的建议编号，参数ndims，dims和periods的含义与函数MPI_Cart_create中相同。

底层支持函数

查询指定图结构下理想的进程编号方式:

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index,
        int *edges, int *newrank)
```

该函数在newrank中返回给定图结构下当前进程的建议编号。参数nnodes，index和edges的含义与函数MPI_Graph_create中相同。

Outline

二维poisson并行有限差分方法

我们考虑长方形区域$\Omega = [0, a] \times [0, b]$上的Poisson方程:

$$\begin{cases} -\Delta u = f, & \text{in } \Omega \\ u = g, & \text{on } \partial\Omega. \end{cases}$$

二维poisson并行有限差分方法

采用均匀网格5点中心差分格式，设$h_x = a/n, h_y = b/m$为网格步长，

$$x_i = i * h_x, y_j = j * h_y,$$

$$u_{i,j} = u(x_i, y_j),$$

$$f_{i,j} = f(x_i, y_j), g_{i,j} = g(x_i, y_j), i = 0, \ldots, n, j = 0, \ldots, m,$$

则差分方程为：

$$\begin{cases} \frac{2u_{i,j}-u_{i+1,j}-u_{i-1,j}}{h_x^2} + \frac{2u_{i,j}-u_{i,j+1}-u_{i,j-1}}{h_y^2} = f_{i,j} \\ u_{i,0} = g_{i,0}, u_{i,m} = g_{i,m}, u_{0,j} = g_{0,j}, u_{n,j} = g_{n,j} \\ i = 1, \ldots, n-1, j = 1, \ldots, m-1 \end{cases}$$

二维poisson并行有限差分方法

令$d = 1/(2/h_x^2 + 2/h_y^2), d_x = d/h_x^2, d_y = d/h_y^2$，则差分方程可以写成：

$$\begin{cases} u_{i,j} - d_x \cdot (u_{i+1,j} + u_{i-1,j}) - d_y \cdot (u_{i,j+1} + u_{i,j-1}) = d \cdot f_{i,j} \\ u_{i,0} = g_{i,0}, u_{i,m} = g_{i,m}, u_{0,j} = g_{0,j}, u_{n,j} = g_{n,j} \\ i = 1, \ldots, n-1, j = 1, \ldots, m-1 \end{cases}$$

Jacobi迭代公式为：

$$u_{i,j}^{new} = d \cdot f_{i,j} + d_x \cdot (u_{i+1,j}^{old} + u_{i-1,j}^{old}) + d_y \cdot (u_{i,j+1}^{old} + u_{i,j-1}^{old})$$
$$i = 1, \ldots, n-1, j = 1, \ldots, m-1$$

程序中取$g(x,y) = -(x^2 + y^2)/4, f(x,y) = 1$。显然解析解为$u_a(x,y) = -(x^2 + y^2)/4$。

二维poisson并行有限差分方法

- 串行程序参看jacobi0.c
- 并行算法与程序:
  设处理器数为nprocs，将计算区域按2维划分成np×nq个子区域，np×nq=nprocs。相邻两个子区域有一个网格步长的重叠，以便于子区域间的数据交换。 每个子区域包含大致相等的网格内点数，每个进程负责一个子区域的计算。

二维poisson并行有限差分方法

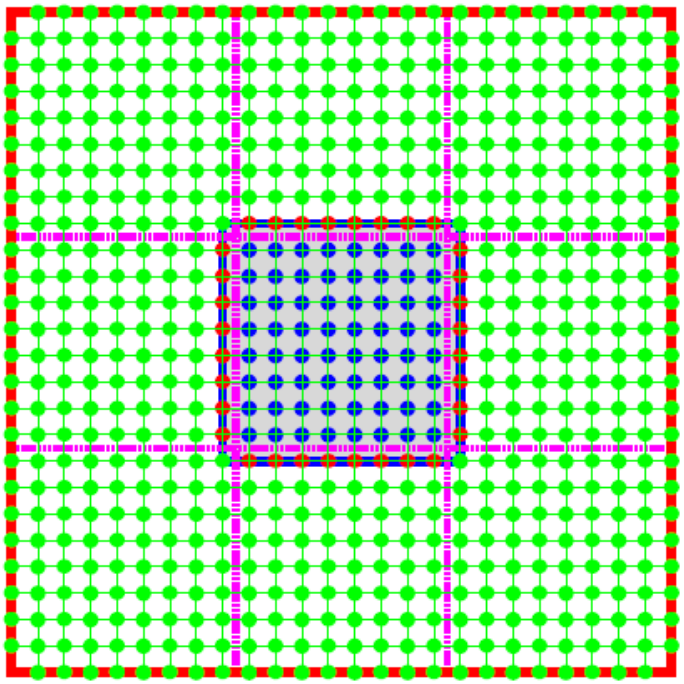
下面是一些变量的说明:

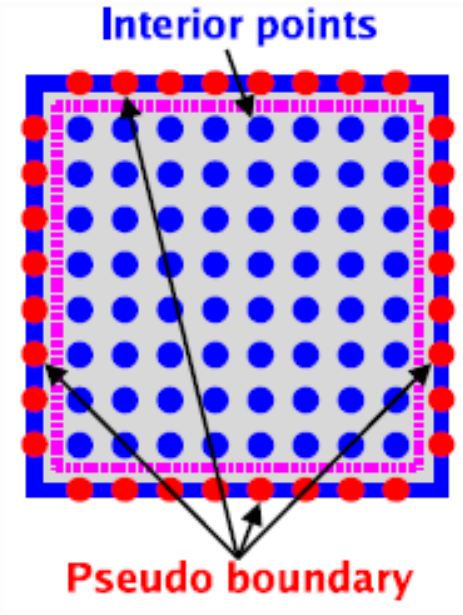
`nprocs`	MPI_COMM_WORLD中的进程数
`np, nq`	x方向和y方向的进程数
`rank_x,rank_y`	进程在MPI_COMM_WORLD中的进程号
`nglob, mglob`	整个区域的网格点数
`nloca, mloca`	子区域的网格点数
`ioffset, joffset`	子区域原点（左下角）在整个网格中的坐标

子区域上的网格点可以表示为（0：nloca，0：mloca），其中（1：nloca－1，1：mloca－1）为子区域内点。并行程序中每个进程仅负责子区域内点的计算。

二维poisson并行有限差分方法

（0，1：nloca－1），（nloca，1：mloca－1），（1：nloca－1，0），（1：nloca－1，mloca）为子区域的“边界点”。





二维poisson并行有限差分方法

子区域的划分：

- 指定处理器的划分，给出np，nq，使得他们满足np×nq=nprocs.
- 为方便子区域间边界数据的交换，用二维坐标（rank_x,rank_y）来标识子区域和进程，他们与进程序号的关系是

$$\begin{cases} \text{rank\_x} = \text{mod(myrank,np)} \\ \text{rank\_y} = \text{myrank/np} \\ \text{myrank} = \text{rank\_x+rank\_y} \times \text{np} \end{cases}$$

二维poisson并行有限差分方法

- 给定处理器划分后，子区域的大小通过将整个计算网格的内点尽量均匀地分配给每个子区域来确定。 以x方向为例，nloca的计算公式为:

$$nloca = \begin{cases} (nglob-1)/np+1, & \text{如果nglob-1能被np整除} \\ (nglob-1)/np+1+\delta, & \text{如果nglob-1不能被np整除} \end{cases}$$

令$r = mod(nglob-1, np)$，则δ定义如下(rank_x<r的进程每个多分一个):

$$\delta = \begin{cases} 1, & \text{if rank\_x } < r \\ 0, & \text{if rank\_x } \geq r \end{cases}$$

二维poisson并行有限差分方法

- 定义完子区域大小后，子区域的原点坐标（在全局网格中的网格点编号）很容易直接计算出来， 以x方向为例，我们有:

$$\text{ioffset} = \begin{cases} \text{(nglob-1)/np*rank\_x} + \text{rank\_x}, & \text{if rank\_x} < r \\ \text{(nglob-1)/np*rank\_x} + \text{r}, & \text{if rank\_x} \geq r \end{cases}$$

  其中$r = mod(nglob - 1, np)$.

参看程序:jacobi1.c,jacobi2.c

作业

- 修改例子jacobi1.c,jacobi2.c程序，用持久通信函数来代替Sendrecv函数(目的：通信与计算的重叠)。
- 修改或重写例子程序jacobi1.c，jacobi2.c，处理两个子区间之间的重叠为两个网格步长的情况， 并且迭代两步交换一下数据。看看jacobi迭代达到同样收敛精度时与jacobi1.c相比运行时间有何变化。