Course 212017H: 并行计算

崔　涛

tcui@lsec.cc.ac.cn

中国科学院数学与系统科学研究院

并行算法设计过程

1. 分析物理（数学）问题，划分任务：
   - 任务尺寸尽量平均；
   - 任务间的数据通信尽可能少；
   - 任务数大于处理器数；
   - 任务数与问题规模成正比。
2. 分析任务之间的通信：
   - 尽可能少的全局通信；
   - 尽可能规则的通信；
   - 各任务的通信量尽量相当；
   - 各任务内部的通信和计算可同时执行。
3. 调整（合并、组合）任务：
   - 减少通信；
   - 减少冗余计算。
4. 将划分后的任务分配到所有进程（进程映射）：
   - 保证负载平衡。

Jacobi迭代求解二维Poisson方程 I

考虑长方形区域$\Omega = (0, a) \times (0, b)$上的Poisson方程:

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

采用均匀网格及5点差分格式离散上述方程. 设$h_x = a/n$, $h_y = b/m$为网格步长, $x_i = ih_x$, $y_j = jh_y$, $u_{i,j} = u(x_i, y_j)$, $f_{i,j} = f(x_i, y_j)$, $g_{i,j} = g(x_i, y_j)$, $i = 0, \ldots, n$, $j = 0, \ldots, m$. 则差分方程为:

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

Jacobi迭代求解二维Poisson方程 II

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:

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

Jacobi迭代的公式为:

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

程序实例中取$g(x,y) = -(x^2+y^2)/4$, $f(x,y) = 1$. 显然解析解为$u_a(x,y) = -(x^2+y^2)/4$, 并且$\forall n, m > 0$, 差分方程的解等于解析解.

串行程序jacobi0.f

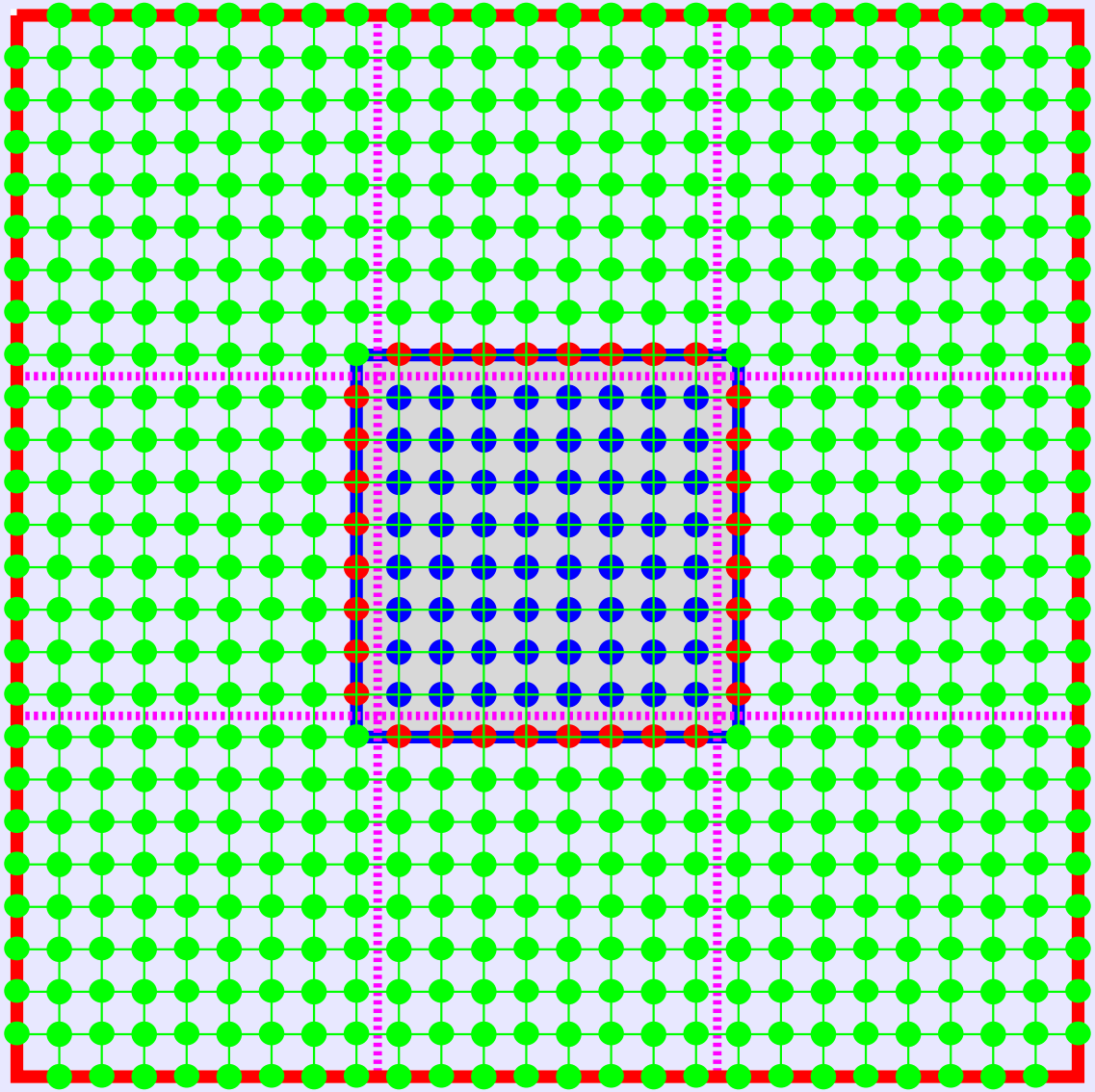
并行算法与程序 I

设处理器数为NPROCS, 将计算区域按二维方式划分成NP × NQ 个子区域, NP × NQ = NPROCS. 相邻子区域间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.

文件jacobi1.cmm中定义了一个公共块/PARMS/, 它包含一些重要的全局变量以及包含mpif.h的语句. 并行程序每个子程序的开始都包含jacobi1.cmm, 以避免通过参数表在子程序间传递大量的参数. 这些全局变量如下:

NPROCS	MPI_COMM_WORLD中的进程数.
NP, NQ	x方向和y方向的进程数.
MYRANK	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.
RANK_X, RANK_Y	x方向和y方向的进程坐标.
IOFFSET, JOFFSET	子区域原点(左下角)在整个网格中的坐标.

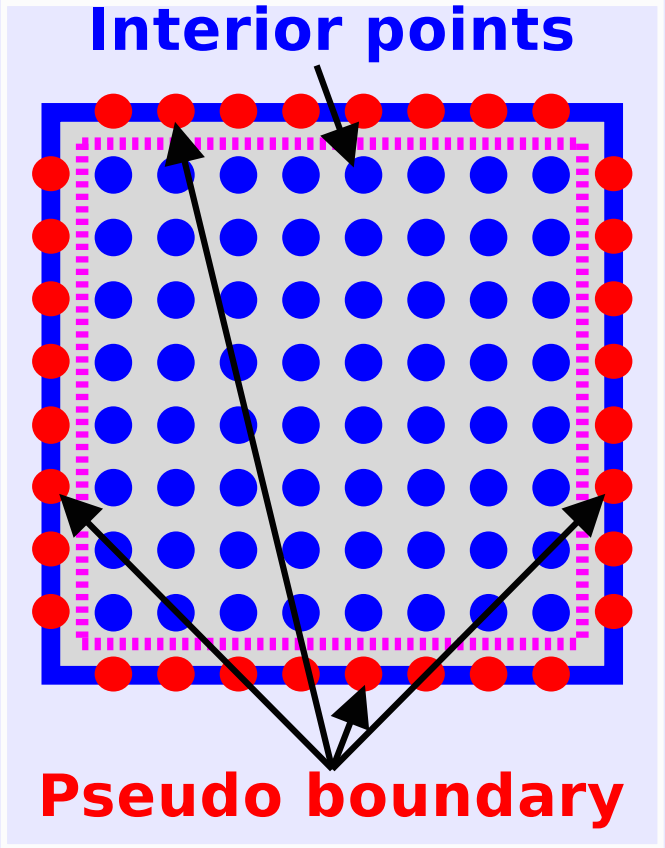
并行算法与程序 II



并行算法与程序 III

NGLOB和MGLOB表示整个网格规模. NLOCA和MLOCA 表示子区域的网格规模. 子区域上的网格点可表示为(0:NLOCA, 0:MLOCA), 其中(1:NLOCA-1, 1:MLOCA-1)为子区域的内点. 并行程序中每个进程仅负责子区域内点的计算. 网格点(0, 1:MLOCA-1), (NLOCA, 1:MLOCA-1), (1:NLOCA-1, 0), 和(1:NLOCA-1, MLOCA) 为子区域的“边界点”, 它们可能是整个计算区域的边界点(物理边界), 也可能是相邻子区域的内点, 后者通常称为“内边界点”(inner boundary points), “拟/伪边界点”(pseudo boundary points), 或“幽灵点”(ghost points).

并行算法与程序 IV



并行算法与程序 V

进行子区域划分后, 每个子区域内点的迭代公式与串行程序完全一样, 但每次迭代后必须通过消息传递来从相邻子区域获取非物理边界点上的新的函数近似值.

子区域划分按如下原则进行:

- 如果用户指定了处理器的划分方式(即用户给出了NP, NQ并且它们满足NP × NQ = NPROCS), 则采用用户指定的处理器的划分. 否则根据总的网格点数NGLOB和MGLOB 调用子程序PARTITION 计算出适当的NP, NQ, 使得子区域中两个方向上的网格点数尽量接近, 从而使得总的通信量达到最小. 子程序PARTITION 是用C语言编写的, 源代码在文件partition.c中.

并行算法与程序 VI

- 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, 它们与进程序号MYRANK的关系是:

$$\begin{cases} \texttt{RANK\_X} = \mathrm{mod}(\texttt{MYRANK}, \texttt{NP}) \\ \texttt{RANK\_Y} = \texttt{MYRANK}/\texttt{NP} \\ \texttt{MYRANK} = \texttt{RANK\_X} + \texttt{RANK\_Y} \times \texttt{NP} \end{cases}$$

- 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各子区域来确定. 以x方向为例, NLOCA 的计算公式如下:

$$\texttt{NLOCA} = \begin{cases} (\texttt{NGLOB} - 1)/\texttt{NP} + 1, & \text{如果NGLOB-1能被NP整除} \\ (\texttt{NGLOB} - 1)/\texttt{NP} + 1 + \delta, & \text{如果NGLOB-1不能被NP整除} \end{cases}$$

并行算法与程序 VII

令$r = \mathrm{mod}(\mathtt{NGLOB} - 1, \mathtt{NP})$, 则$\delta$定义如下:

$$\delta = \begin{cases} 1, & \text{if } \mathtt{RANK\_X} < r \\ 0, & \text{if } \mathtt{RANK\_X} \geq r \end{cases}$$

- 按上式确定子区域大小后, 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(如果定义了适当的通信器, 也可调用MPI_SCAN函数计算). 仍以x方向为例, 我们有:

$$\mathtt{IOFFSET} = \begin{cases} (\mathtt{NGLOB\text{-}1})/\mathtt{NP} \cdot \mathtt{RANK\_X} + \mathtt{RANK\_X}, & \text{if } \mathtt{RANK\_X} < r \\ (\mathtt{NGLOB\text{-}1})/\mathtt{NP} \cdot \mathtt{RANK\_X} + r, & \text{if } \mathtt{RANK\_X} \geq r \end{cases}$$

其中$r = \mathrm{mod}(\mathtt{NGLOB} - 1, \mathtt{NP})$.

以上参数的计算在子程序—INIT0—中完成.

子程序JACOBI中定义了两个数据类型T1和T2, 分别对应于数组的一列(x方向)和一行(y方向), 用于子区域内边界的消息传递.

矩阵乘法 I

$$C = A * B, A \in R^{m \times l}, B \in R^{l \times n}$$

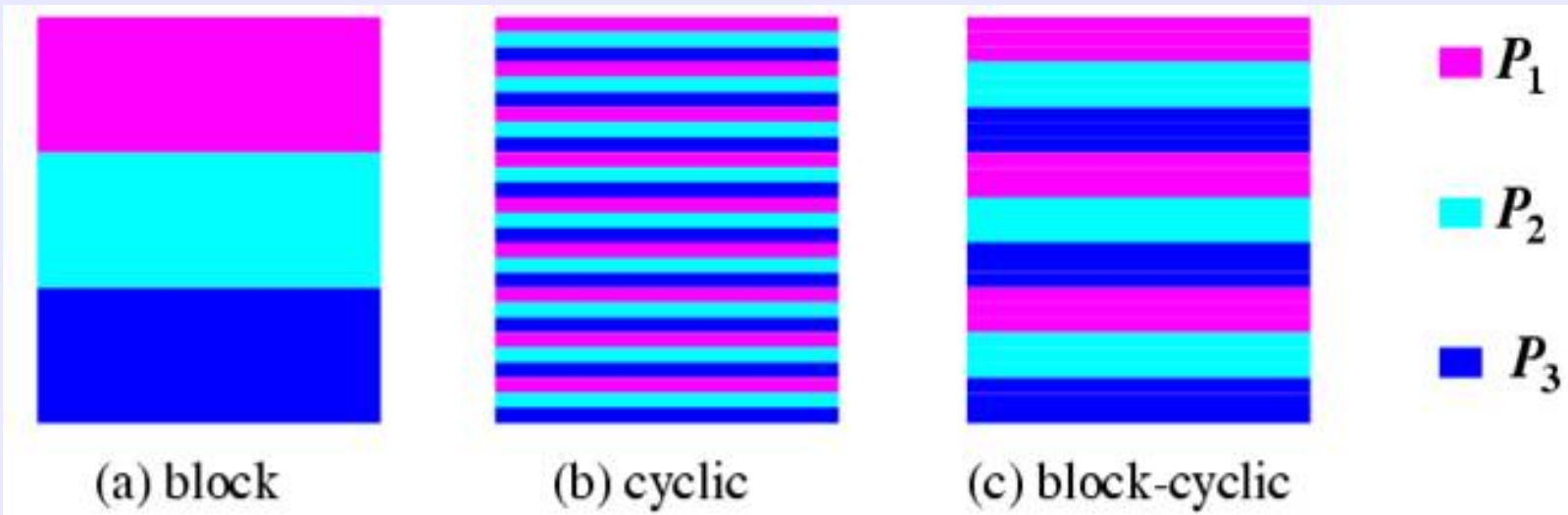
串行算法:i-j-k循环,算法复杂度$O(n^3)$

```
for i=1 to m
  for j=1 to l
    C(i,j)=0
    for k=1 to n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    end for
  end for
end for
```

矩阵并行乘法 I

$$C = A * B, A \in R^{m \times l}, B \in R^{l \times n}$$

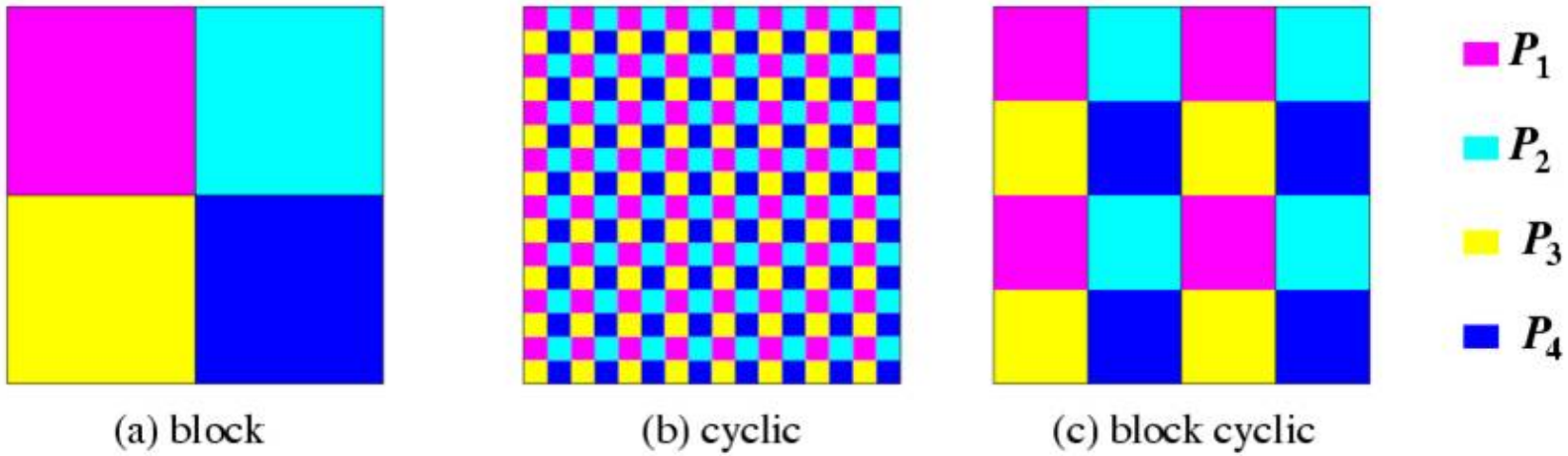
- 假定: m, l, n均能被p整除，其中p为进程数
- 基于分而治之思想的两种划分策略：条形划分和块状划分，
  - 条形（行或列）划分:



Striped row-major mapping of a 27×27 matrix on $p = 3$ processors.

矩阵并行乘法 II

- 块状划分



Checkerboard mapping of a 16×16 matrix on $p = 2 \times 2$ processors.

矩阵并行乘法: 行列划分 I

$$AB = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} \begin{bmatrix} B_0 & B_1 & \cdots & B_{p-1} \end{bmatrix} = \begin{bmatrix} A_0B_0 & A_0B_1 & \cdots & A_0B_{p-1} \\ A_1B_0 & A_1B_1 & \cdots & A_1B_{p-1} \\ \vdots & & \ddots & A_0B_{p-1} \\ A_{p-1}B_0 & A_{p-1}B_1 & \cdots & A_{p-1}B_{p-1} \end{bmatrix}$$

```
for i=0 to p-1
  j=(i+myid) mod p
  C_j=A*B
  src = (myid+1) mod p
  dest = (myid-1+p) mod p
  if (i!=p-1)
    send(B,dest)
    recv(B,src)
  end if
end for
```

矩阵并行乘法: 行列划分 II

本算法中：`C_j` $= C_{myid,j}$, `A` $= A_{myid}$。

思考:

- 行行划分: $A = \begin{bmatrix} A_0 \\ \vdots \\ A_p \end{bmatrix}$、$B = \begin{bmatrix} B_0 \\ \vdots \\ B_p \end{bmatrix}$、$C = \begin{bmatrix} C_0 \\ \vdots \\ C_p \end{bmatrix}$
- 列列划分$A = [A_0 \cdots A_p]$、$B = [B_0 \cdots B_p]$、$C = [C_0 \cdots C_p]$
- 列行划分$A = [A_0 \cdots A_p]$、$B = \begin{bmatrix} B_0 \\ \vdots \\ B_p \end{bmatrix}$、$C =$???

块状划分：Cannon 算法 I

$$A = \begin{bmatrix} A_{00} & \cdots & A_{1m} \\ \vdots & A_{ij} & \vdots \\ A_{m0} & \cdots & A_{mm} \end{bmatrix}, B = \begin{bmatrix} B_{00} & \cdots & B_{1m} \\ \vdots & B_{ij} & \vdots \\ B_{m0} & \cdots & B_{mm} \end{bmatrix}$$

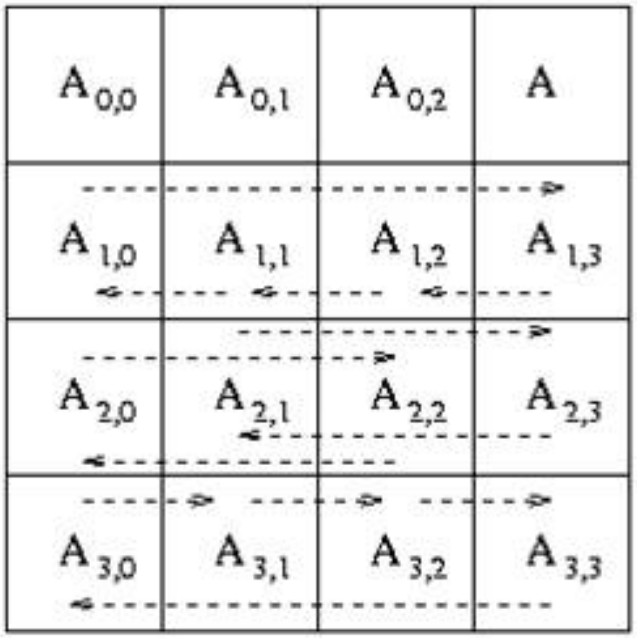
$$C = \begin{bmatrix} C_{00} & \cdots & C_{1m} \\ \vdots & C_{ij} & \vdots \\ C_{m0} & \cdots & C_{mm} \end{bmatrix}$$

$$C_{ij} = \sum_{k=0}^{m} A_{ik} B_{kj}$$

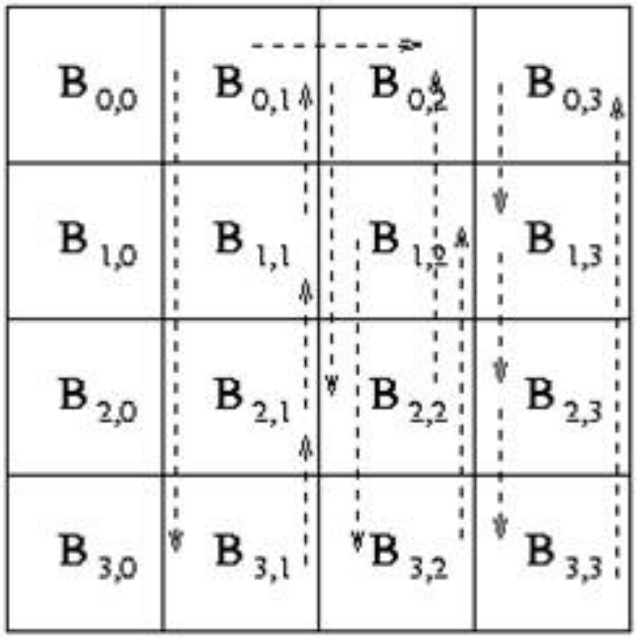
cannon_algorithm.txt

算法示例：

块状划分：Cannon 算法 II

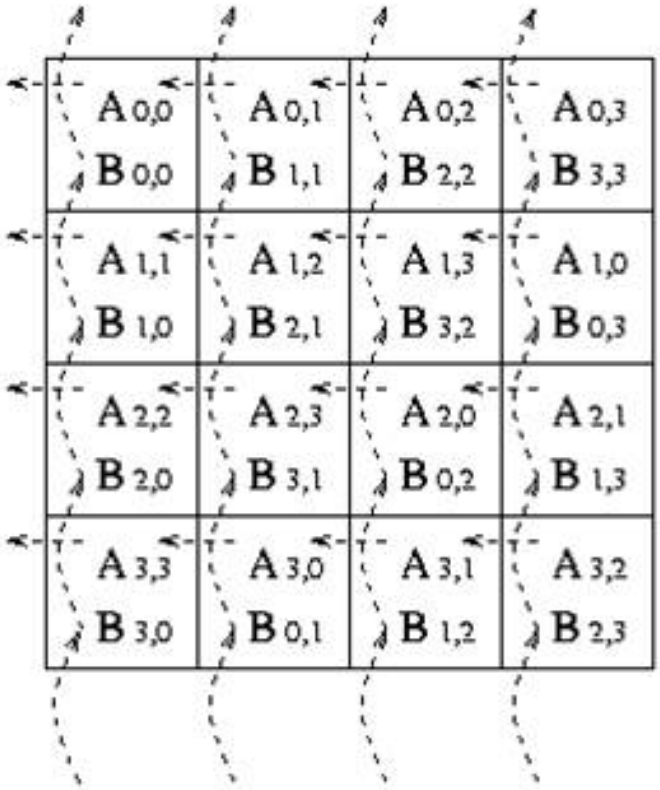


(a) Initial alignment of A

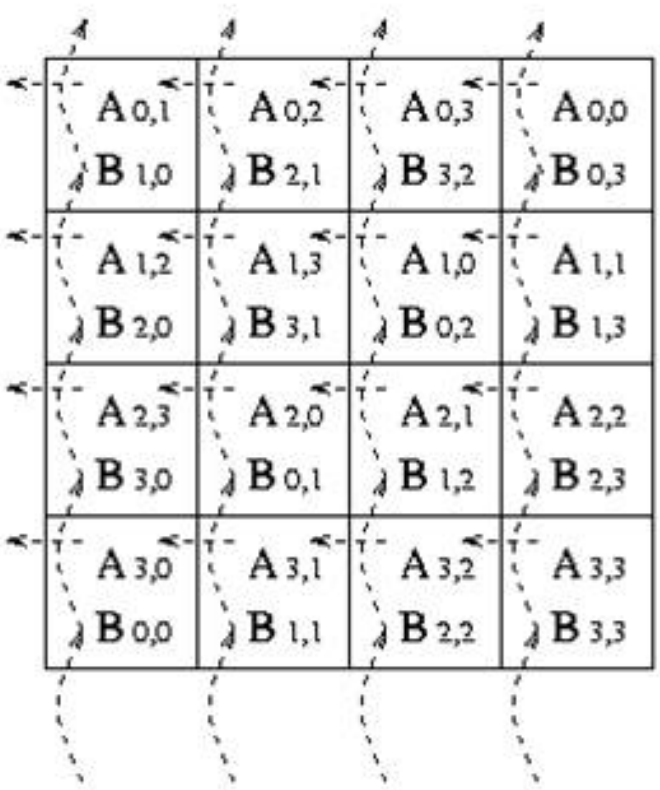


(b) Initial alignment of B

块状划分：Cannon 算法 III

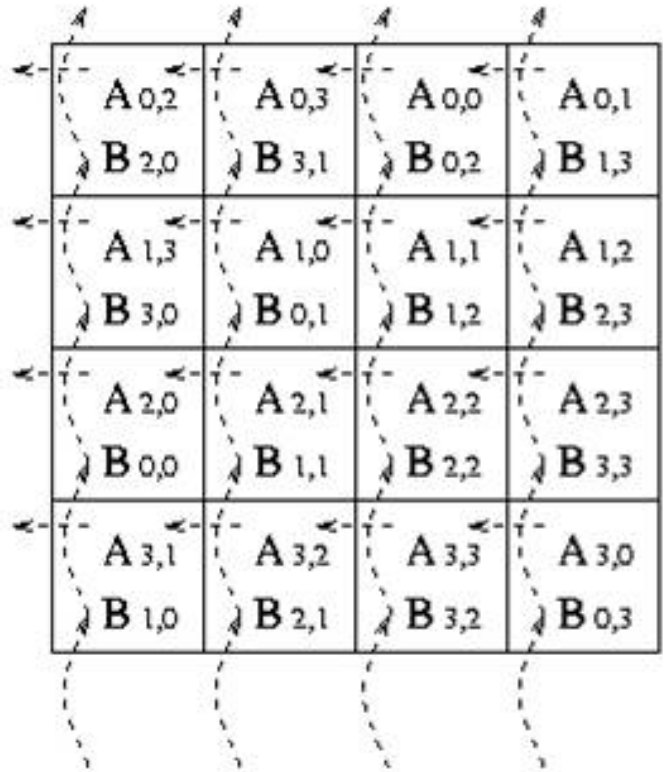


(c) A and B after initial alignment



(d) Submatrix locations after first shift

块状划分：Cannon 算法 IV



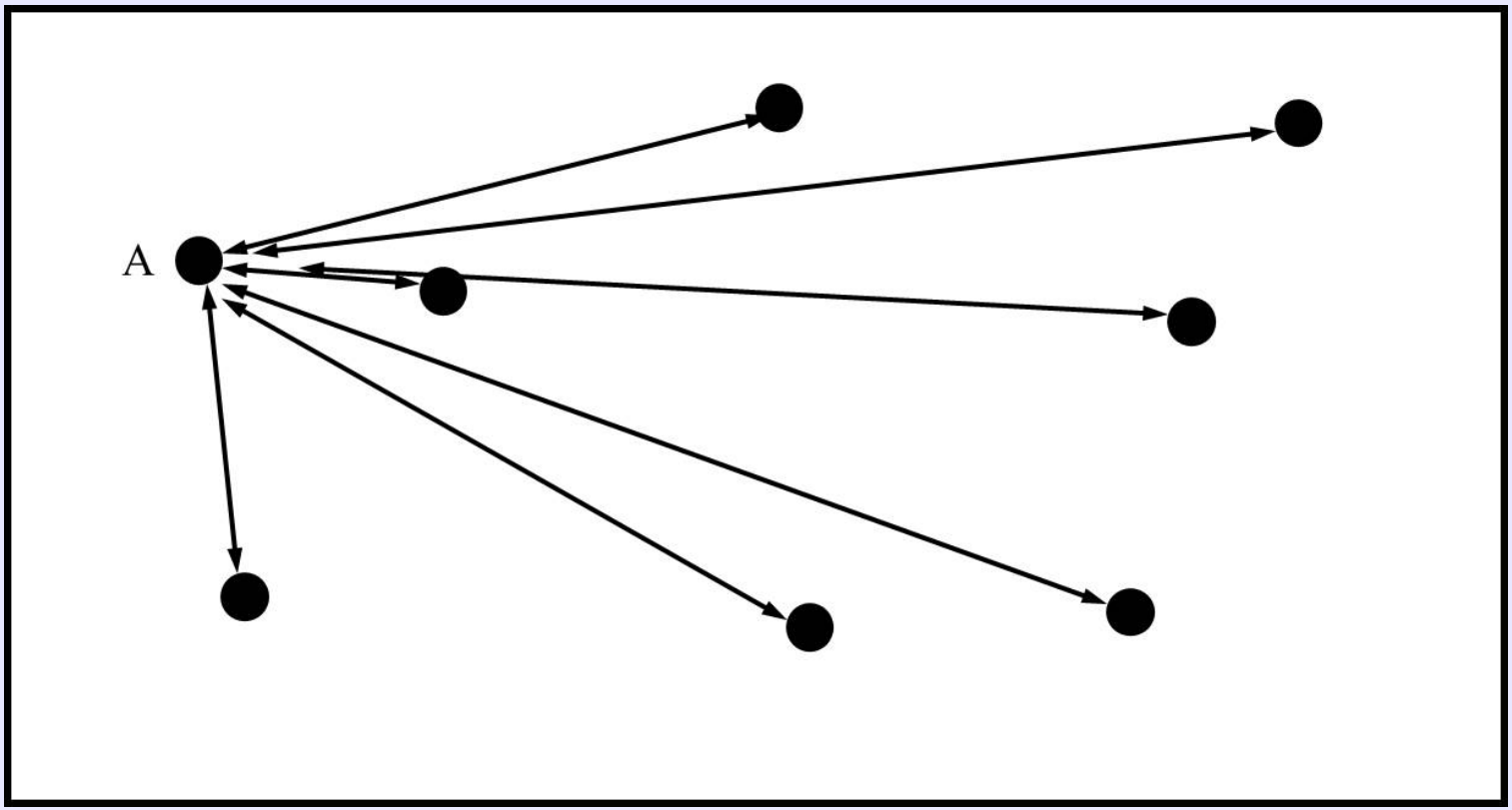
(e) Submatrix locations after second shift

A 0,3 B 3,0	A 0,0 B 0,1	A 0,1 B 1,2	A 0,2 B 2,3
A 1,0 B 0,0	A 1,1 B 1,1	A 1,2 B 2,2	A 1,3 B 3,3
A 2,1 B 1,0	A 2,2 B 2,1	A 2,3 B 3,2	A 2,0 B 0,3
A 3,2 B 2,0	A 3,3 B 3,1	A 3,0 B 0,2	A 3,1 B 1,3

(f) Submatrix locations after third shift

NBody问题 I

研究N个质点相互之间在万有引力作用下的运动规律，又称N体问题。这个问题作为研究天体系统运动的一种力学模型，是天体力学和一般力学中的一个基本问题。



NBody问题 II

N-Body Solver: 通过模拟手段确定相互作用的天体/粒子在一段时间后的位置和速度

假设体系中粒子总数为n,粒子q的质量为m_q, 粒子q在t时刻的位置为$s_q(t)$。

- 根据万有引力定律粒子q在t时刻所受外力$F_q(t)$为:

$$F_q(t) = -Gm_q \sum_{j=0, j\neq q}^{n-1} \frac{m_j}{|s_q(t) - s_j(t)|^3}(s_q(t) - s_j(t))$$

NBody问题 III

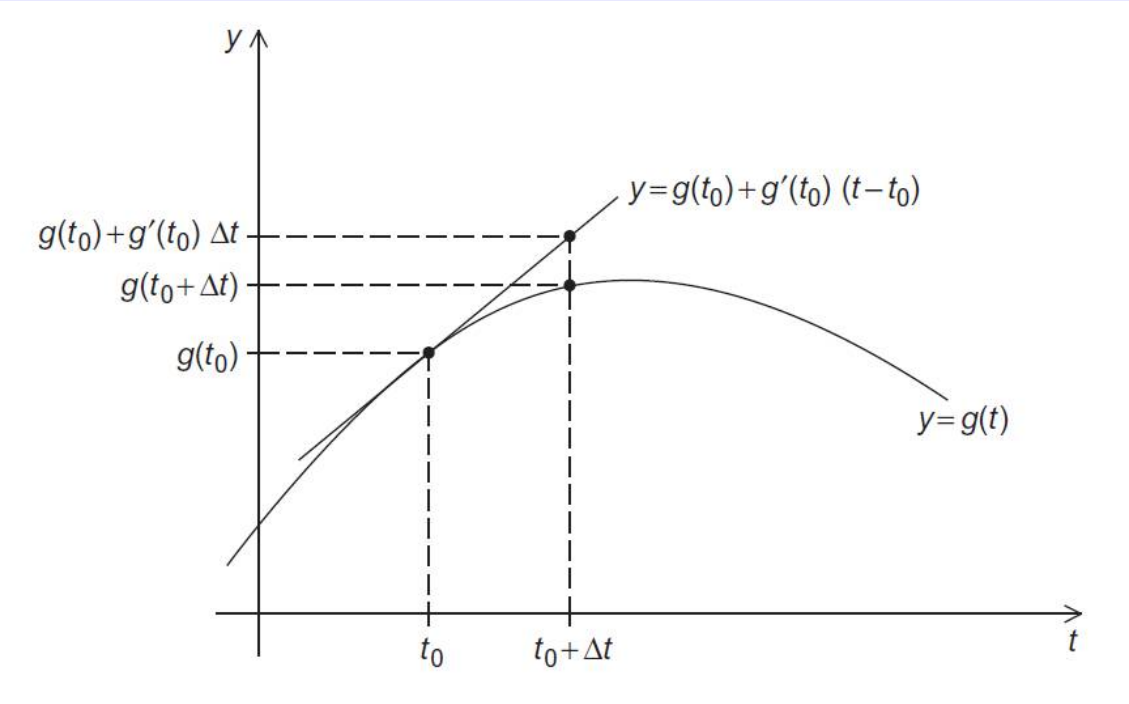
- 粒子q在t时刻的速度为:

$$v_q(t) = \frac{\partial s_q(t)}{\partial t}$$

- 牛顿第二运动定律：

$$F_q(t) = m_q \frac{\partial v_q(t)}{\partial t} = m_q \frac{\partial^2 s_q(t)}{\partial t^2}$$

串行算法 I



$$s_q(t+\Delta t) = s_q(t) + \Delta t v_q(t)$$

$$v_q(t+\Delta t) = v_q(t) + \Delta t \frac{F_q(t)}{m_q}$$

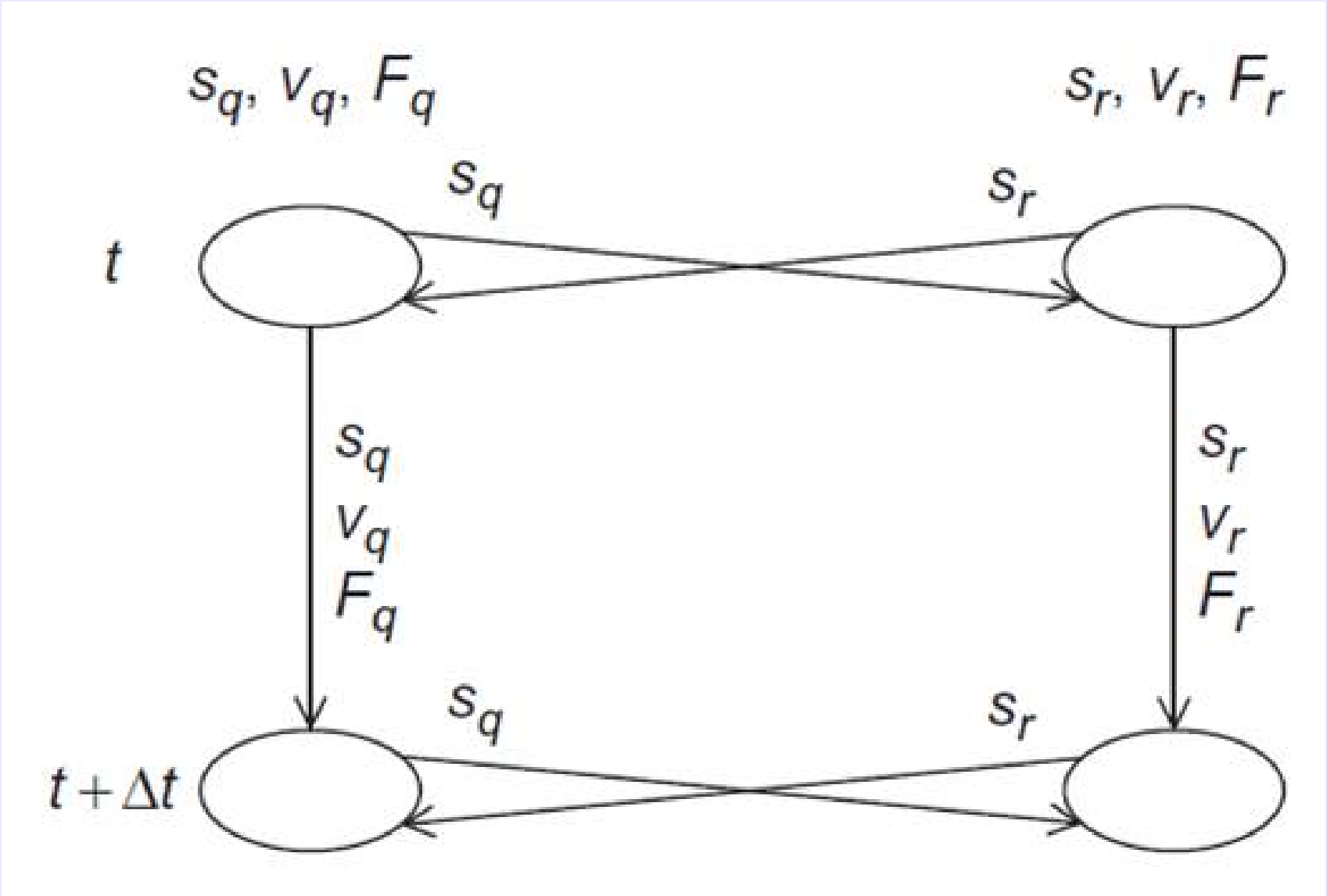
串行算法 II

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

力的计算:

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

通讯 I



OpenMP并行化 I

```
#   pragma omp parallel
    for each timestep {
        if (timestep output) {
#           pragma omp single
            Print positions and velocities of particles;
        }
#       pragma omp for
        for each particle q
            forces[q] = 0.0;
#       pragma omp for
        for each particle q
            Compute total force on q;
#       pragma omp for
        for each particle q
            Compute position and velocity of q;
    }
```

MPI并行化 I

数据结构设定

- 每个进程都维护一份包含所有粒子质量和位置的全局数组。
- 每个进程只存储own的粒子的力和速度。
- 每个进程通过位置指针指向自己own的粒子在全局数组中的起始位置。

```
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions into global pos array;
}
Print positions and velocities of particles;
```

编号问题 I

- 问题描述
  假设通信器MPI_COMM_WORLD中包含p个进程，每个进程中有一个长度为n、取值为非负整数的数组obj[]。进一步，假定同一进程中obj[]数组的分量值互不相等，但不同进程中的obj[]数组的分量值则可能相等。
  在实际应用中，obj[]代表一组分布存储在各个进程中的对象的编号，同一个对象可以同时存在于多个进程中，其在不同进程中的编号必须一样。例如，并行有限元程序中，obj[]可以是顶点、边、面或单元的全局编号。
  试设计解决下述计算问题的并行算法和MPI并行程序。

  问题一: 统计数目。统计所有进程中所包含的不同对象数N。
  问题二: 指定属主。为每个对象指定一个唯一的属主进程(owner process)。
  问题三: 重新编号。重新对所有对象进行编号，编号范围为0,...,N-1。

编号问题 II

- 算法
  1. 收集算法：收集所有对象到进程0。
     优点：实现简单。
     缺点：负载不平衡，并且可能超出进程0的最大内存。
  2. 循环算法：将所有进程按进程号首尾相连排成一个有向环(最后一个进程与首个进程相连)，让obj[] 在进程环上依次轮转同时进行比较、处理。
     缺点：通信量大，且需要p步才能完成。
  3. 分配算法：给定一个从obj[]的值域到$\{0,\ldots,p-1\}$的映射map，各进程将obj[i] 发送到进程map(obj[i]) 进行比较、处理。可以通过不同map的选择来优化通信和负载平衡。作为一个特例，如果取map$(i)\equiv 0$，则该算法等同于收集算法。
- 程序实例：

  程序一：统计对象数。 numbering-count.c

  程序二：指定属主并统计对象数。 numbering-owner.c

编号问题 III

程序三: 指定属主、统计对象数并对对象重新编号。 numbering-renumber.c

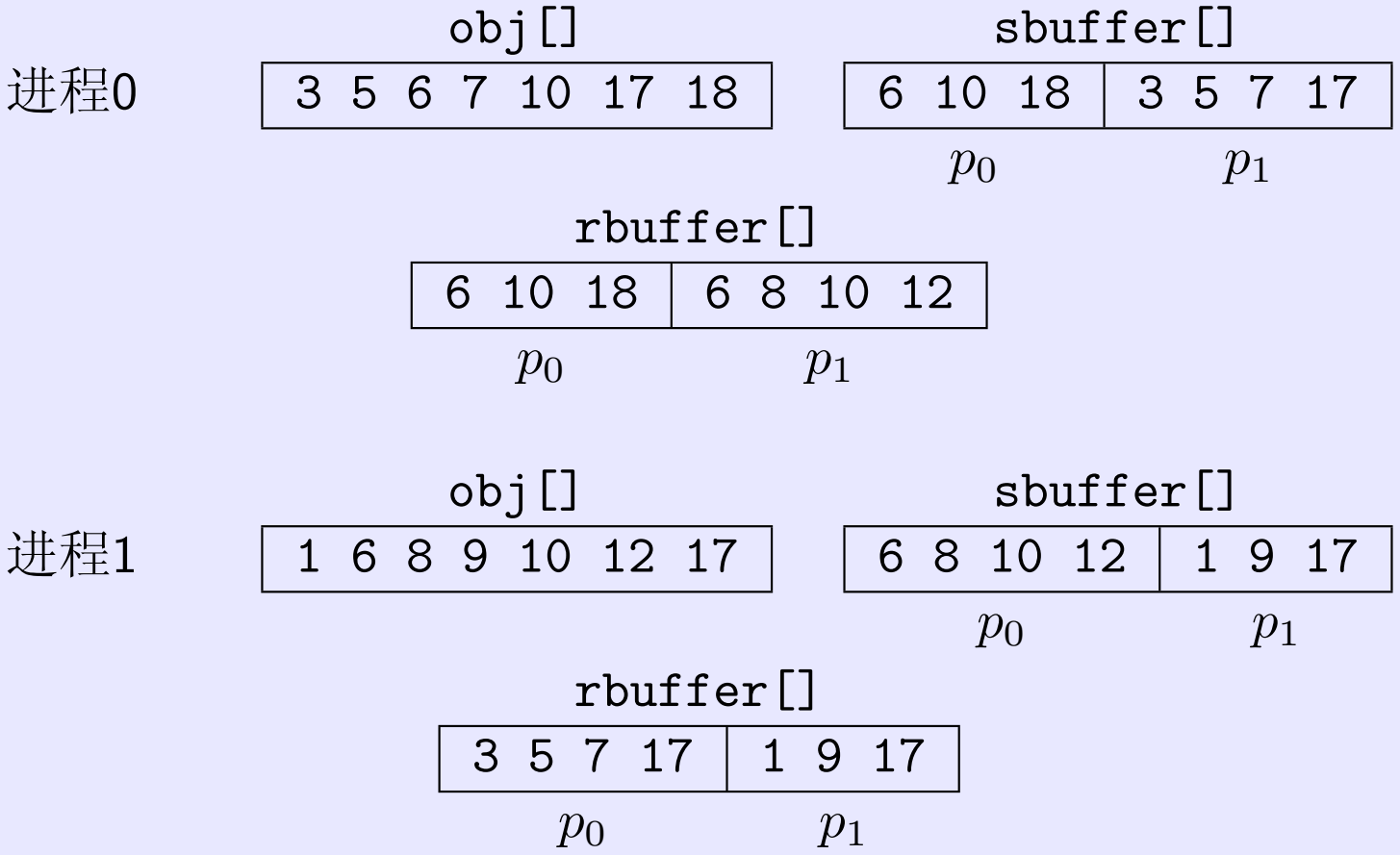


Figure : 通信示例(两个进程，map$(i) = i\%p$)

编号问题 IV

- 思考:
  这里所介绍的分配算法中的通信一般情况下是“稠密”的，即每个进程需要与所有其它进程通信，通信复杂度为$O(p)$，通信量为$O(n)$。但对一些特殊情况，通过精心选取map函数，可以使得通信是“稀疏”的，即每个进程只与$O(1)$个其它进程间有通信，并且通信量 $\ll n$。
- 习题
  1. 试用循环算法解决问题二，并比较计算效率。
  2. 当如果去掉“同一进程中obj[]数组的分量值互不相等”的条件，程序一、程序二和程序三各需做哪些改动?
  3. 改进属主选择策略，以求达到更好的负载平衡效果(属于各进程的对象数尽量相等)。
  4. 假设obj[]数组的值域为$\{0, \ldots, M-1\}$。试改变程序三中map的定义，使得重新编号后新编号保持老编号的顺序。

并行程序性能评价 I

并行程序执行时间（execution time）等于从并行程序开始执行，到所有进程执行完毕，墙上时钟走过的时间，也称之为墙上时间（wall time）,包括：

计算CPU 时间 进程指令执行所花费的CPU 时间，它可以分解为两个部分，一个是程序本身指令执行占用的CPU 时间，即通常所说的用户时间（user time），主要包含指令在CPU 内部的执行时间和内存访问时间，另一个是为了维护程序的执行，操作系统花费的CPU 时间，即通常所说的系统时间（system time），主要包含内存调度和管理开销、I/O 时间、以及维护程序执行所必需要的操作系统开销等。通常地，系统时间可以忽略。

通信CPU 时间 包含进程通信花费的CPU 时间。

同步开销时间 包含进程同步花费的时间。

并行程序性能评价 II

进程空闲时间 当一个进程阻塞式等待其他进程的消息时，CPU 通常是空闲的，或者处于等待状态。进程空闲时间是指并行程序执行过程中，进程所有这些空闲时间的总和。

在处理器资源独享的前提下，假设某个串行应用程序在某台并行机单处理器上的执行时间为T_S，而该程序并行化后，P 个进程在P 个处理器并行执行所需要的时间为T_P.，则该并行程序在该并行机上的加速比S_P 可定义为：

$$S_P = \frac{T_S}{T_P}$$

效率E_P定义为:

$$E_P = \frac{S_P}{P} = \frac{T_S}{T_P * P}$$

这里，需要说明的是，T_1 指处理器个数为1 时，并行程序的执行时间。通常情形下，T_1 大于T_S，因为并行程序往往引入一些冗余的控制和管理开销。

并行程序性能评价 III

将并行程序的墙上时间分解为:

$$T_P = C_i + D_i \tag{1}$$

其中，C_i为第i个进程花费的CPU 时间，D_i 为第i 个进程的空闲时间。

$$C_i = L_i + O_i \qquad i = 1, 2, \cdots, P \tag{2}$$

其中，L_i 为第i 个进程数值计算指令执行花费的CPU 时间，O_i 为第i 个进程通信、同步花费的CPU 时间。

并行计算粒度进程 指令数值计算时间与墙上时间的比值，即：

$$\gamma_i = \frac{L_i}{T_P}$$

非数值冗余 由于并行引入的额外非数值计算开销,

$$W_i = D_i + O_i \qquad i = 1, 2, \cdots, P$$

并行程序性能评价 IV

负载平衡 为了减少无谓的空闲时间，各个进程分配的CPU 时间尽量相等，为此，定义负载平衡效率如下：

$$\eta_P = \frac{\sum_{i=0}^{P-1} C_i}{P \times \max_{i=1,2,\cdots,P} C_i}$$

根据$C_T = \sum_i C_i, D_T = \sum_i D_i$, $C_T + D_T = T_P \times P$,则效率公式

$$E_P = \frac{S_P}{P} = \frac{T_S}{C_T} \times \frac{C_T}{C_T + D_T}$$

分别定义：

数值效率$NE_P = \frac{T_S}{C_T}$

并行效率$PE_P = \frac{C_T}{C_T + D_T}$

并行程序性能评价 V

数值效率反映了并行计算引入的额外CPU 时间开销，这种开销来自两个方面。一方面，并行执行时，随着处理器个数的增长，各个进程的cache 命中率将提高，有助于缩短总的计算CPU 时间，从而提高数值效率；另一方面，并行计算可能引入额外的开销，例如并行算法增加计算量、并行通信CPU 时间和同步开销等，它们将延长计算CPU 时间，从而降低数值效率。

并行效率反映了并行程序具体执行的并行性能，它依赖于并行机网络的通信性能，以及并行程序的负载平衡等方面，并行效率总是小于1的。

如果数值效率大于1，则可能效率将大于1，也就是，并行程序的加速比将大于处理器的个数，此时，称之为超线性加速比。由以上的分析可知，需要辩证地看待超线性加速比。如果串行程序能够很好地发挥单处理器的峰值性能，则并行程序几乎不可能获得超线性加速比。反之，如果出现超线性加速比，说明串行程序需要进一步的性能优化。

并行程序性能评价 VI

可扩展分析 给定并行算法（程序）和并行机，如何调整参与并行计算的处理器个数P 和求解问题的计算规模W，使得随着处理器个数的增长并行计算的效率可以保持不变，称之为并行程序和并行机相结合的可扩展分析。通常地，它具有四个目的：

1. 选择合理的算法与结构组合
2. 性能预测
3. 最优性能选择
4. 指导性能优化

程序性能优化 I

串行程序性能的优化是并行程序性能优化的基础,在基于微处理器的高性能计算机上，提高程序单机性能的关键是改善程序的访存性能、提高cache 命中率、以及充分挖掘CPU 多运算部件、流水线的处理能力。

- 调用高性能库
- 选择适当的编译器优化选项
- 合理定义数组维数
- 注意嵌套循环的顺序
- 数据分块
- 循环展开

在并行算法确定之后，影响并行程序效率的主要因素是通信开销、由于数据相关性或负载不平衡引起的进程空闲等待、以及并行算法引入的冗余计算。在设计并行程序时，可以采用多种技术来减少或消除这些因素对并行效率的影响。

- 减少通信量、提高通信粒度

程序性能优化 II

- 全局通信尽量利用高效聚合通信算法
  - 优点：函数往往经过专门优化
  - 缺点: 无法在通信的同时进行计算工作
- 挖掘算法的并行度，减少CPU 空闲等待
- 负载平衡
- 通信、计算的重叠
- 通过引入重复计算来减少通信