第 8 章　二维 Poisson 方程

本章介绍一个采用 5 点差分格式、结合点 Jacobi 迭代求解二维 Poisson 方程的 MPI Fortran 程序实例。这个算法本身比较简单，效率也很低，但是它包含了规则网格上基于区域分解的偏微分方程计算 MPI 程序的典型结构和通信模式，非常有代表性。本章介绍的程序实例稍加修便可以用于许多其他类似问题的计算。此外，该程序亦不难推广到三维问题以及非定常问题 (主要适用于显式格式及某些隐式格式)。

本章将首先简单介绍二维区域上 Poisson 方程的 5 点差分离散以及如何用 Jacobi 迭代来求解所导出的线性方程组。在此基础上给出基于区域分解的并行算法和 MPI 并行程序 Fortran 代码。同时，为该算法建立并行模型对算法的并行效率进行分析。最后，从几个不同的角度对 MPI 并行程序进行改进。

考虑定义在二维规则区域上的 Poisson 方程：

$$\begin{cases} -\Delta u(x,y) = f(x,y) \quad (x,y) \in \Omega = (0,W) \times (0,H) \\ u(x,y)|_{\partial\Omega} = g(x,y) \end{cases} \tag{8.1}$$

其中，$f(x,y)$ 和 $g(x,y)$ 为已知函数，分别定义在区域 Ω 的内部和边界。

沿坐标轴 x 和 y 方向，分别取步长

$$h_x = \frac{W}{\texttt{IM}}, \qquad h_y = \frac{H}{\texttt{JM}} \tag{8.2}$$

将区域 Ω 离散成规模为 $\texttt{IM} \times \texttt{JM}$ 的网格，其中 IM 和 JM 分别为沿坐标轴 x 和 y 方向的网格单元个数。

假设方程 (8.1) 的离散解 $u(x,y)$ 定义在所有网格结点上，且用如下未知量表示

$$\begin{cases} u_{i,j} = u(i \times h_x, j \times h_y) & 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1 \\ u_{i,j} = g_{i,j} = g(i \times h_x, j \times h_y) & i = 0 \text{ 或 } i = \mathtt{IM} \text{ 或 } j = 0 \text{ 或 } j = \mathtt{JM} \end{cases} \tag{8.3}$$

用二阶中心差商近似导数：

$$\begin{cases} u_{xx}(i \times h_x, j \times h_y) \approx \dfrac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} \\ u_{yy}(i \times h_x, j \times h_y) \approx \dfrac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \end{cases} \tag{8.4}$$

并记

$$f_{i,j} = f(i \times h_x, j \times h_y) \tag{8.5}$$

将以上公式代入方程 (8.1)，便得到了它的 5 点差分离散。而问题则转化为求解稀疏线性代数方程组：

$$2(h_x^2 + h_y^2)u_{i,j} - h_y^2(u_{i-1,j} + u_{i+1,j}) - h_x^2(u_{i,j-1} + u_{i,j+1}) = h_x^2 h_y^2 f_{i,j}, \quad 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1 \tag{8.6}$$

该方程组包含 $(\mathtt{IM}-1)\times(\mathtt{JM}-1)$ 个未知量 $u_{i,j}$，$i = 1, \ldots, \mathtt{IM}-1$，$j = 1, \ldots, \mathtt{JM}-1$。

这里，采用众所周知的 Jacobi 点迭代算法求解方程组 (8.6)。从任意一个初始近似解

$$u_{i,j}^0, \quad i = 1, \ldots, \mathtt{IM}-1, \quad j = 1, \ldots, \mathtt{JM}-1$$

出发，迭代计算

$$\begin{cases} u_{i,j}^k = \dfrac{h_x^2 h_y^2 f_{i,j} + h_y^2(u_{i-1,j}^{k-1} + u_{i+1,j}^{k-1}) + h_x^2(u_{i,j-1}^{k-1} + u_{i,j+1}^{k-1})}{2(h_x^2 + h_y^2)} \\ i = 1, \ldots, \mathtt{IM}-1, \quad j = 1, \ldots, \mathtt{JM}-1 \end{cases}$$

$k = 1, 2, \ldots$，直到近似解 $u_{i,j}^k$ 满足误差要求。

在程序实例中取 $f(x, y) = -4$，方程 (8.1) 的解析解为 $u(x, y) = x^2 + y^2$，此时离散方程和原方程的解是一样的。由于离散方程的精确解已知，因此程序中直接比较近似解与精确解之间的误差来判断近似解是否满足误差要求，当不知道离散方程的精确解时，可以计算近似解的余量来判断是否达到收敛要求。初始近似解取为 $u_{i,j}^0 = 0$，$i = 1, \ldots, \mathtt{IM} - 1$，$j = 1, \ldots, \mathtt{JM} - 1$。

8.1 并行算法设计

设计求解方程 (8.1) 的 MPI 并行算法必须考虑两个关键问题：

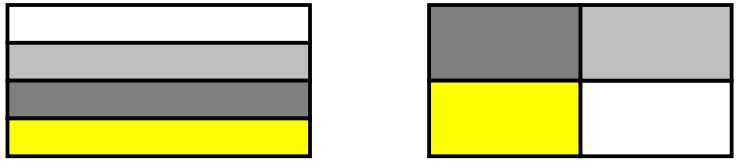
第一，选择恰当的区域分解策略，将区域 Ω 分解成多个子区域，分配给不同的进程，并保证进程间的负载平衡和最小的消息传递通信开销。通常有两种方式：

(1) 沿 x 方向或 y 方向的一维条分解策略，图 8.1(a) 显示了沿 y 方向划分的情况；

(2) 沿两个方向的二维块分解策略，如图 8.1(b) 所示。显然，如果某个方向的进程个数等于 1，则二维块分解策略就退化为一维条分解策略。无论哪种方式，都应该尽量保证每个子区域包含的网格结点个数相等，因为这样才能保证进程间的负载平衡。

第二，选择合适的通信数据结构。由式 (8.6) 可知，在任意网格结点上，执行 Jacobi 点迭代需要知道该结点上、下、左、右四个相邻结点上的近似解。因此，在每次 Jacobi 迭代之前，每个进程必须与其相邻的进程交换边界结点上的近似解。

为了描述通信数据结构，不妨设方程近似解定义在网格单元的中心。图 8.2 给出了一个 3×3 的二维块区域分解，其中，各个子区域被分配给不同的进程，各个进程负责求解该子区域的近似解。具



(a) 一维条分解 (b) 二维块分解

图 8.1 两种区域分解策略

体地，进程 5 将向其相邻的四个进程 (进程 2、进程 4、进程 6 和进程 8) 输出 “●” 标示的网格单元的近似解；同时，从这四个进程接收用 “○” 标示的网格单元上的近似解。因此，如何管理这些沿区域分解边界交换的网格单元上的量，将会直接影响到 MPI 程序的并行性能。图 8.3 给出了一个比较有效的办法，就是沿各个子区域的边界，向外增加一个宽度为 1 的辅助网格单元，用于存储相邻子区域在这些网格单元上的近似解。

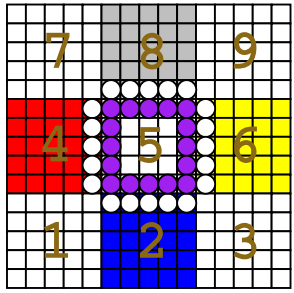


图 8.2 3 × 3 的二维块区域分解

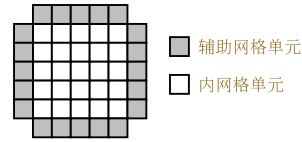


图 8.3 辅助网格单元示意图

类似地，同样的数据结构也适应于近似解定义在网格结点上的情形，这里不再讨论。

8.2 MPI 并行程序设计

下面，基于以上二维块区域分解策略和通信数据结构，给出求解差分方程 (8.6) 的 MPI 并行程序的 Fortran 代码。其中，近似解定义在网格结点上。为简单起见，这里假设进程数 $p = \texttt{NPX} \times \texttt{NPY}$，NPX 和 NPY 分别是沿 x 方向和 y 方向的进程个数，网格单元个数 IM 和 JM 能分别被 NPX 和 NPY 整除，子区域的网格规模为 IML × JML，其中 IML = IM/NPX，JML = JM/NPY，进程按自然序排列 (先沿 x 方向，后沿 y 方向)。

代码 8.1： 点 Jacobi 迭代 MPI 并行程序：二维块分解策略。

文件名: code/poisson/poisson0.f

```
! Poisson 方程求解: 使用阻塞通信 (可能死锁)。作者: 莫则尧
      INCLUDE  'mpif.h'
      PARAMETER(DW=2.0, DH=3.0)  ! 问题求解区域沿 X、Y 方向的大小
      PARAMETER(IM=30, JM=60)    ! 沿 X、Y 方向的全局网格规模
      PARAMETER(NPX=2,  NPY=1)   ! 沿 X、Y 方向的进程个数
      PARAMETER(IML=IM/NPX, JML=JM/NPY)
        ! 各进程沿 X、Y 方向的局部网格规模
      REAL  U(0:IML+1, 0:JML+1)  ! 定义在网格结点的近似解
      REAL  US(0:IML+1, 0:JML+1) ! 定义在网格结点的精确解
      REAL  U0(IML, JML)         ! Jacobi 迭代辅助变量
      REAL  F(IML, JML)          ! 函数 f(x,y) 在网格结点上的值
      INTEGER  NPROC             ! mpirun 启动的进程个数, 必须等于 NPX*NPY
      INTEGER  MYRANK,MYLEFT,MYRIGHT,MYUPPER,MYLOWER
                                 ! 各进程自身的进程号, 4 个相邻进程的进程号
      INTEGER  MEPX,MEPY         ! 各进程自身的进程号沿 X、Y 方向的坐标
      REAL  XST,YST              ! 各进程拥有的子区域沿 X、Y 方向的起始坐标
      REAL  HX, HY               ! 沿 X、Y 方向的网格离散步长
      REAL  HX2,HY2,HXY2,RHXY
      INTEGER  IST,IEND,JST,JEND
                        ! 各进程沿 X、Y 方向的内部网格结点的起始和终止坐标
      INTEGER  HTYPE, VTYPE
```

```
                    ! MPI 用户自定义数据类型, 表示各进程沿 X、Y 方向
                    ! 与相邻进程交换的数据单元
      INTEGER  STATUS(MPI_STATUS_SIZE)  !
      DOUBLE PRECISION T0, T1
! In-line functions
      solution(x,y)=x*x+y*y    ! 解析解
      rhs(x,y)=-4.0              ! Poisson 方程源项 (右端项)
! 程序可执行语句开始
      CALL MPI_Init(IERR)
      CALL MPI_Comm_size(MPI_COMM_WORLD,NPROC,IERR)
      IF (NPROC.NE.NPX*NPY.OR.MOD(IM,NPX).NE.0.OR.MOD(JM,NPY).NE.0) THEN
         PRINT *, '+++ mpirun -np xxx error OR grid scale error, ',
     &           'exit out +++'
         CALL MPI_Finalize(IERR)
         STOP
      ENDIF
! 按自然序 (先沿 X 方向, 后沿 Y 方向) 确定各进程自身及其 4 个相邻进程的进程号
      CALL MPI_Comm_rank(MPI_COMM_WORLD,MYRANK,IERR)
      MYLEFT  = MYRANK - 1
      IF (MOD(MYRANK,NPX).EQ.0)   MYLEFT=MPI_PROC_NULL
      MYRIGHT = MYRANK + 1
      IF (MOD(MYRIGHT,NPX).EQ.0)  MYRIGHT=MPI_PROC_NULL
      MYUPPER = MYRANK + NPX
      IF (MYUPPER.GE.NPROC)       MYUPPER=MPI_PROC_NULL
      MYLOWER = MYRANK - NPX
      IF (MYLOWER.LT.0)           MYLOWER=MPI_PROC_NULL
      MEPY=MYRANK/NPX
      MEPX=MYRANK-MEPY*NPX
!                     对应二维 NPYxNPX Cartesian 行主序坐标为 (MEPY,MEPX).
! 基本变量赋值, 确定各进程负责的子区域
      HX =DW/IM
      HX2=HX*HX
      HY =DH/JM
      HY2=HY*HY
      HXY2=HX2*HY2
      RHXY=0.5/(HX2+HY2)
```

```
      DX=HX2*RHXY
      DY=HY2*RHXY
      DD=RHXY*HXY2
      XST=MEPX*DW/NPX
      YST=MEPY*DH/NPY
      IST=1
      IEND=IML
      IF (MEPX.EQ.NPX-1) IEND=IEND-1   ! 最右边的区域 X 方向少一个点
      JST=1
      JEND=JML
      IF (MEPY.EQ.NPY-1) JEND=JEND-1   ! 最上边的区域 Y 方向少一个点
! 数据类型定义
      CALL MPI_Type_contiguous(IEND-IST+1, MPI_REAL, HTYPE, IERR)
      CALL MPI_Type_commit(HTYPE, IERR)
                ! 沿 X 方向的连续 IEND-IST+1 个 MPI_REAL 数据单元,
                ! 可用于表示该进程与其上、下进程交换的数据单元
      CALL MPI_Type_vector(JEND-JST+1, 1, IML+2, MPI_REAL, VTYPE, IERR)
      CALL MPI_Type_commit(VTYPE, IERR)
                ! 沿 Y 方向的连续 JEND-JST+1 个 MPI_REAL 数据单元,
                ! 可用于表示该进程与其左、右进程交换的数据单元
! 初始化
      DO J=JST-1, JEND+1
      DO I=IST-1, IEND+1
         xx=(I+MEPX*IML)*HX           ! xx=XST+I*HX
         yy=(J+MEPY*JML)*HY           ! yy=YST+J*HY
         IF (I.GE.IST.AND.I.LE.IEND .AND. J.GE.JST.AND.J.LE.JEND) THEN
            U(I,J)  = 0.0             ! 近似解赋初值
            US(I,J) = solution(xx,yy) ! 解析解
            F(I,J)  = DD*rhs(xx,yy)   ! 右端项
         ELSE IF ((I.EQ.IST-1  .AND. MEPX.EQ.0) .OR.
     &            (J.EQ.JST-1  .AND. MEPY.EQ.0) .OR.
     &            (I.EQ.IEND+1 .AND. MEPX.EQ.NPX-1) .OR.
     &            (J.EQ.JEND+1 .AND. MEPY.EQ.NPY-1)) THEN
            U(I,J) = solution(xx,yy)  ! 边界值
         ENDIF
```

```
      ENDDO
      ENDDO
! Jacobi 迭代求解
      NITER=0
      T0 = MPI_Wtime()
100   CONTINUE
      NITER=NITER+1
! 交换定义在辅助网格结点上的近似解
      CALL MPI_Send(U(1,1),      1, VTYPE, MYLEFT,  NITER+100, !
     &              MPI_COMM_WORLD,IERR)                    ! 发送左边界
      CALL MPI_Send(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送右边界
      CALL MPI_Send(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送下边界
      CALL MPI_Send(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送上边界
      CALL MPI_Recv(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
     &              MPI_COMM_WORLD,STATUS,IERR)             ! 接收右边界
      CALL MPI_Recv(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
     &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收左边界
      CALL MPI_Recv(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
     &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收上边界
      CALL MPI_Recv(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
     &              MPI_COMM_WORLD, STATUS, IERR)           ! 接收下边界
      DO J=JST,JEND     !
      DO I=IST,IEND
         U0(I,J)=F(I,J)+DX*(U(I,J-1)+U(I,J+1))+DY*(U(I-1,J)+U(I+1,J))
      ENDDO
      ENDDO             !
! 计算与精确解间的误差
      ERR=0.0
      DO J=JST,JEND
      DO I=IST,IEND
         U(I,J)=U0(I,J)
         ERR=MAX(ERR, ABS(U(I,J)-US(I,J))) ! 用 L^∞ 模以使误差与NP无关
      ENDDO
```

```
129       ENDDO
130       ERR0=ERR
131       CALL MPI_Allreduce(ERR0,ERR,1,MPI_REAL,MPI_MAX,
132      &                   MPI_COMM_WORLD,IERR)
133       IF (MYRANK.EQ.0 .AND. MOD(NITER,100).EQ.0) THEN
134          PRINT *, 'NITER = ', NITER, '    ERR = ', ERR
135       ENDIF
136       IF (ERR.GT.1.E-3)  THEN     ! 收敛性判断
137          GOTO 100                 ! 没有收敛, 进入下次迭代
138       ENDIF
139       T1 = MPI_Wtime()
140       IF (MYRANK.EQ.0) THEN
141          PRINT *, ' !!! Successfully converged after ',
142      &           NITER, ' iterations'
143          PRINT *, ' !!! error = ', ERR, '   wtime = ', T1 - T0
144       ENDIF
145   ! 输出近似解 (略)
146       CALL MPI_Finalize(IERR)
147       END
```

上例中固定了该 MPI 程序产生的进程个数 NP = NPY*NPX，这样做主要是为了方便将各进程的数组大小声明成仅为相应串行程序的 1/(NPY*NPX)，从而使得原来串行程序在单机上由于内存不够而无法计算的问题通过多机并行计算成为可能[1]。当然，这样做也带来一些不便，例如它要求 MPI 运行命令 "mpirun -np xxx" 中的参数 xxx 等于 NPY*NPX，并且参数 NPY 和 NPX 被改变后，必须重新编译该程序。

[1]Fortran 77 中没有动态内存分配功能，习题 1 中提供了一些克服这一限制的方法。

8.3 并行效率分析

本节对 Jacobi 迭代的计算量、通信量进行统计，分析算法的并行效率。

首先统计程序的浮点计算时间。代码 8.1 的主要计算量是 117–121 行的 Jacobi 迭代循环，每步循环需要 6 次浮点计算 (4 次加法，2 次乘法，其中可能的优化可以参看习题 4)。此外，在计算近似解误差的循环中，平均每个网格点需要 3 次浮点计算 (1 次浮点减法，1 次取绝对值，1 次取 max)。因此，每个进程中每步迭代的总浮点计算量近似等于：

$$9 \times \texttt{IML} \times \texttt{JML}$$

假设每个处理机上运行一个进程，而处理机完成一次浮点运算的时间为 T_0，则每个进程的总计算时间为：

$$T_{\text{cpu}} \approx 9 \times T_0 \times \texttt{IML} \times \texttt{JML}$$

下面统计程序的通信时间。每步迭代中的通信分两部分，第一部分通信是相邻子区域间交换辅助网格点上的近似解。为简单起见，假设一次通信的时间包括通信开销 (延迟) 和数据传输时间 (带宽)，并且发送数据和接收数据的时间一样，其中通信开销只依赖于通信次数，数据传输时间只依赖于收发的数据量，则各进程中每步迭代总通信时间的最大者为 (假设总进程数 p 大于 2)：

$$T_{\texttt{comm}} + T_{\texttt{lat}} \approx \begin{cases} 4 \times T_1 \times \texttt{JM} + 4 \times T_2, & \texttt{NPY} = 1 \\ 4 \times T_1 \times \texttt{IM} + 4 \times T_2, & \texttt{NPX} = 1 \\ 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) + 8 \times T_2, & \texttt{NPX} > 1, \texttt{NPY} > 1 \end{cases} \tag{8.7}$$

其中 T_1 为传送一个浮点数需要的平均时间 (通信带宽的倒数)，T_2 为一次通信的开销 (通信延迟)。上式表明，如果采用一维条划分的

区域划分方式，每个进程的通信量是一个与进程数无关的常数；如果采用二维块划分的区域划分方式，则通信次数是一维条划分时的 2 倍，而通信量依赖于进程数，并且随进程数的增加而减少。不难看出，如果固定总的处理器数目和问题规模，则使得通信量最小的最优进程网格划分是使得 IML 和 JML 的值尽可能接近的划分，这就是通常所说的划分处理器网格时应该尽量使得子区域接近“正方形”。为简化讨论，这里仅考虑式 (8.7) 最后一种情况。

第二部分通信是计算全局误差时的归约操作。这类操作通常采用树型算法，所需要的时间为：

$$T_{\texttt{reduce}} = C \times \log p$$

其中 C 为常数，p 为总进程数。

由于 Jacobi 迭代中没有因数据相关而引起的空闲等待，因而并行程序的运行时间为：

$$\begin{aligned} T_{\text{并行}} &= T_{\text{cpu}} + T_{\text{comm}} + T_{\text{lat}} + T_{\text{reduce}} \\ &\approx 9 \times T_0 \times \texttt{IML} \times \texttt{JML} + 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) \\ &\quad + 8 \times T_2 + C \times \log p \end{aligned}$$

显然，串行程序的运行时间为：

$$T_{\text{串行}} = 9 \times T_0 \times (\texttt{IM} - 1) \times (\texttt{JM} - 1) \approx 9 \times T_0 \times \texttt{IM} \times \texttt{JM}$$

因此，并行加速比为：

$$\begin{aligned} S &= T_{\text{串行}} / T_{\text{并行}} \\ &\approx \frac{9 \times T_0 \times \texttt{IM} \times \texttt{JM}}{9 \times T_0 \times \texttt{IML} \times \texttt{JML} + 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) + 8 \times T_2 + C \times \log p} \end{aligned}$$

并行效率为 (注意 IML = IM/NPX，JML = JM/NPY)：

$$
\begin{aligned}
E &= S/p \\
&= \frac{9 \times T_0 \times \mathtt{IM} \times \mathtt{JM}}{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML} + 4 \times T_1 \times (\mathtt{IML} + \mathtt{JML}) + 8 \times T_2 + C \times \log p} \\
&\quad \times \frac{1}{\mathtt{NPX} \times \mathtt{NPY}} \\
&= \frac{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML}}{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML} + 4 \times T_1 \times (\mathtt{IML} + \mathtt{JML}) + 8 \times T_2 + C \times \log p}
\end{aligned}
$$

上式中 T_0，T_1，T_2 和 C 均为常数。令：

$$
\delta = 4 \times T_1 \times \frac{\mathtt{IML} + \mathtt{JML}}{\mathtt{IML} \times \mathtt{JML}} + \frac{8 \times T_2 + C \times \log p}{\mathtt{IML} \times \mathtt{JML}} \tag{8.8}
$$

则：

$$
E = \frac{1}{1 + \delta/T_0} \tag{8.9}
$$

由于 IML + JML 相当于子区域边界上的网格点数的一半，IML × JML 相当于子区域内部的网格点数，因而通常称 (IML + JML)/(IML × JML) 为子区域的“面体比”[2]。显然面体比随子区域的增大而减小。从式 (8.8) 和式 (8.9) 易知，子区域的面体比越小，并行效率越高。因此，当进程数目固定时，并行效率将随子区域问题规模的增加而增加。

8.4 MPI 并行程序的改进

在代码 8.1 中，交换定义在辅助网格结点近似解的 8 条消息传递语句 (101–116 行) 是该 MPI 程序的关键。但是，由 MPI 标准可知，它们是不安全的，因为在某些并行机上，当消息较长时，可能由于 MPI 系统缓存区大小的限制，而导致执行该 MPI 程序的进程死

[2]名词“面体比”来源于三维问题，出于习惯对二维问题依然沿用这一叫法。

锁 (参看 196 页代码 3.1)。这里，可以将它们替换成如下的非阻塞通信函数。

代码 8.2: 改进一：非阻塞通信 (源程序见文件 poisson1.f)。

```
! 用下一行替换 poisson0.f 第 24 行
      INTEGER  REQ(8), STATUS(MPI_STATUS_SIZE,8)
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson0.f 101-116 行
      CALL MPI_Isend(U(1,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD,REQ(1),IERR)           ! 发送左边界
      CALL MPI_Isend(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD,REQ(2),IERR)           ! 发送右边界
      CALL MPI_Isend(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD,REQ(3),IERR)           ! 发送下边界
      CALL MPI_Isend(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD,REQ(4),IERR)           ! 发送上边界
      CALL MPI_Irecv(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD, REQ(5),IERR)          ! 接收右边界
      CALL MPI_Irecv(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD, REQ(6),IERR)          ! 接收左边界
      CALL MPI_Irecv(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD, REQ(7),IERR)          ! 接收上边界
      CALL MPI_Irecv(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD, REQ(8),IERR)          ! 接收下边界
      CALL MPI_Waitall(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
      ... ... ... ... ... ... (略)
```

在代码 8.1 中，将 117–121 行的循环分裂成两个部分，其中一个部分需要辅助网格点上的近似解，而另一个部分不需要辅助网格点上的近似解。这样，为了改进该 MPI 程序的并行性能，可以将后一个部分的计算与代码 8.2 的非阻塞消息传递重叠起来，从而达到屏蔽网络延迟的目的。具体改进如下。

代码 8.3: 改进二：重叠通信与计算 (源程序见文件 poisson2.f)。

```
    ... ... ... ... ... ... (略)
! 用下述内容替换 poisson1.f 118-122 行
     DO J=JST+1,JEND-1
     DO I=IST+1,IEND-1
        U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
    &                          +HY2*(U(I-1,J)+U(I+1,J)))
     ENDDO
     ENDDO
     CALL MPI_Waitall(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
     DO J=JST, JEND, JEND-JST
     DO I=IST, IEND
        U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
    &                          +HY2*(U(I-1,J)+U(I+1,J)))
     ENDDO
     ENDDO
     DO J=JST, JEND
     DO I=IST, IEND, IEND-IST
        U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
    &                          +HY2*(U(I-1,J)+U(I+1,J)))
     ENDDO
     ENDDO
    ... ... ... ... ... ... (略)
```

在代码 8.1–8.3 中，各进程按自然序 (先 x 后 y) 确定与它相邻的 4 个进程的进程号 (MYLEFT，MYRIGHT，MYLOWER，MYUPPER)，以及它自己所处的行主序位置 (MEPY，MEPX)。实际上，这些进程按区域分解策略可以很自然地映射到 NPY*NPX 的二维 Cartesian 拓扑结构 (参看 3.2.8)，而 (MEPY，MEPX) 就是各进程在该拓扑结构中的坐标。因此，可以从通信器 MPI_COMM_WORLD 出发，建立二维 Cartesian 拓扑结构，从而方便地确定各进程的相邻关系，并使得之后的所有 MPI 消息传递均基于该拓扑结构进行。

代码 8.4: 改进三：二维 Cartesian 拓扑结构 (源程序见文件 poisson3.f)。

```
      ... ... ... ... ... ... (略)
! 在 poisson[012].f 程序头(变量声明部分)加入下面二行
      INTEGER COMM, DIMS(2),COORD(2)
      LOGICAL PERIOD(2),REORDER
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson[012].f 38-49 行
      DIMS(1)=NPY            ! 拓扑结构中Y方向的进程个数
      DIMS(2)=NPX            ! 拓扑结构中X方向的进程个数
      PERIOD(1)=.FALSE.      ! 沿Y方向, 拓扑结构非周期连接
      PERIOD(2)=.FALSE.      ! 沿X方向, 拓扑结构非周期连接
      REORDER=.TRUE.         ! 在新通信器中, 允许进程重新排序
      CALL MPI_Cart_create(MPI_COMM_WORLD, 2, DIMS, PERIOD, REORDER,
     &                     COMM, IERR)
      CALL MPI_Comm_rank(COMM,MYRANK,IERR)
      CALL MPI_Cart_coords(COMM,MYRANK,2,COORD,IERR)
      MEPY=COORD(1)
      MEPX=COORD(2)
      CALL MPI_Cart_shift(COMM, 0, 1, MYLOWER, MYUPPER, IERR)   ! Y方向
      CALL MPI_Cart_shift(COMM, 1, 1, MYLEFT,  MYRIGHT, IERR)   ! X方向
      ... ... ... ... ... ... (略)
```

代码 8.1–8.4 中忽略了近似解的输出。这里采用 3.2.9 中介绍的 MPI 并行 I/O 函数实现近似解的并行输出，要求输出的近似解按自然序排列，且包含物理边界结点。

代码 8.5: 改进四：并行 I/O (源程序见文件 poisson4)。该程序使用了独立文件指针聚合型输出函数 `MPI_File_write_all` (参看表 3.3)。

```
      ... ... ... ... ... ... (略)
! 在程序头(变量声明部分)加入下面内容
      INTEGER FH, FILETYPE, MEMTYPE, GSIZE(2), LSIZE(2), START(2)
! 注意: 从下面三种形式的变量声明中根据所使用的MPI系统选择一个正确的
```

```
! (可以参考文件 mpiof.h 或 mpif.h 中 MPI_OFFSET_KIND 的定义)
!     INTEGER(kind=MPI_OFFSET_KIND) OFFSET    ! 适用于 Fortran 90/95
!     INTEGER*8 OFFSET                        ! 适用于 64 位系统
      INTEGER*4 OFFSET                        ! 适用于某些 32 位系统
      ... ... ... ... ... ... (略)
! 在标有“输出近似解(略)”处 (倒数第三行) 加入下述内容
      GSIZE(1)=IM+1
      GSIZE(2)=JM+1
      LSIZE(1)=IEND-IST+1
      IF (MEPX.EQ.0) LSIZE(1)=LSIZE(1)+1
      IF (MEPX.EQ.NPX-1) LSIZE(1)=LSIZE(1)+1
      LSIZE(2)=JEND-JST+1
      IF (MEPY.EQ.0) LSIZE(2)=LSIZE(2)+1
      IF (MEPY.EQ.NPY-1) LSIZE(2)=LSIZE(2)+1
      START(1)=IML*MEPX
      IF (MEPX.NE.0) START(1)=START(1)+1
      START(2)=JML*MEPY
      IF (MEPY.NE.0) START(2)=START(2)+1
!  定义局部子数组数据类型
      CALL MPI_Type_create_subarray(2, GSIZE, LSIZE, START,
     &          MPI_ORDER_FORTRAN, MPI_REAL, FILETYPE, IERR)
      CALL MPI_Type_commit(FILETYPE, IERR)
! 打开二进制文件
      CALL MPI_File_open(COMM,'result.dat',
     &          MPI_MODE_CREATE + MPI_MODE_WRONLY,
     &          MPI_INFO_NULL, FH, IERR)
      OFFSET=0          ! 注意使用正确的变量类型 (INTEGER*4 或 INTEGER*8)
                        ! (参考文件 mpif.h 中 MPI_OFFSET_KIND 的定义)
      CALL MPI_File_set_view(FH, OFFSET, MPI_REAL, FILETYPE,
     &          'native', MPI_INFO_NULL, IERR)
! 定义数据类型, 描述子数组在内存中的分布
      GSIZE(1)=IML+2
      GSIZE(2)=JML+2
      START(1)=1
      IF (MEPX.EQ.0) START(1)=0
      START(2)=1
```

```
      IF (MEPY.EQ.0) START(2)=0
      CALL MPI_Type_create_subarray(2, GSIZE, LSIZE, START,
     &          MPI_ORDER_FORTRAN, MPI_REAL, MEMTYPE, IERR)
      CALL MPI_Type_commit(MEMTYPE, IERR)
! 输出近似解(含物理边界结点)
      CALL MPI_File_write_all(FH, U, 1, MEMTYPE, STATUS, IERR)
      CALL MPI_File_close(FH, IERR)
      ... ... ... ... ... ... (略)
```

至此，代码 8.2–8.5 分别从非阻塞通信、重叠通信与计算、拓扑结构和并行 I/O 四个方面，利用相应的 MPI 函数，依次改进了代码 8.1 中 MPI 程序的功能和并行性能。通过该应用示例，读者可以较好地将所介绍的 MPI 函数联系在一起，解决实际问题。

习　题

1. 试用下面两种方式修改代码 8.1，使得改变进程数目或问题规模后不必重新编译：

(1) 根据结点机的物理内存容量固定各个数组大小；

(2) 利用一个 C 语言编写的函数动态分配内存。

2. 代码 8.1 中使用了 L^∞ 模计算近似解的误差，从并行计算和舍入误差的角度讲使用 L^∞ 模和 L^2 模有什么区别?

3. 修改代码 8.1，使其能够处理 NPX 不是 IM 的倍数或 NPY 不是 JM 的倍数的情况。

4. 代码 8.1 第 117–121 行的 Jaboci 迭代循环中，假设 $h_x = h_y$ (即 DX = DY)，试改写代码，将平均每个网格点的浮点工作量降为 1 个乘法、3 个加法。

5. 使用不同处理机数目、区域划分及问题规模运行代码 8.1，计算相应的并行效率和加速比，并分析所得到的性能结果。

6. 通过将代码 8.1 中的 `MPI_Send` 和 `MPI_Recv` 进行适当配对，然后用 `MPI_Sendrecv` 代替是否可以避免通信死锁？用这样的方式避免通信死锁与代码 8.2 相比有何优劣？

7. 在代码 8.1–8.5 中用了一排辅助网格单元。通过增加虚拟网格点的宽度可以提高通信粒度。例如，如果使用两排辅助网格单元，则可以每两次迭代交换一次子区域边界附近的近似解，代价是子区域间增加了少量的重复计算。试修改代码 8.4 或代码 8.5 中的程序，在程序中增加一个参数 `BW`，它代表辅助网格单元的宽度 (`BW` $\geqslant 1$)，比较不同 `BW` 的值对程序性能的影响。

8. 修改代码 8.5，将 Jacobi 迭代改为红黑顺序的 Gauss–Seidel 迭代。

9. 统计 Jacobi 迭代的浮点运算次数，修改代码 8.1–8.4，在程序结束时打印出实际达到的 Mflops 值，并根据处理机的峰值性能计算程序的实际效率。

10. 在本章程序的基础上，设计三维 Poisson 方程 Jacobi 迭代求解的 MPI 并行程序。