ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ MPI ДЛЯ РАС-ПАРАЛЛЕЛИВАНИЯ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ НА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ С НЕВЫСОКОЙ СКОРОСТЬЮ ОБМЕНОВ МЕЖДУ ПРОЦЕССОРАМИ

М.А. Толстых

Институт вычислительной математики РАН

Создание программных комплексов задач математической физики на параллельных вычислительных системах с распределенной памятью и ограниченной пропускной способностью связующей сети имеет ряд особенностей, которые рассматриваются в данной статье. Предполагается, что читатель знаком в общих чертах со стандартом передачи данных MPI [1, 2, 3]. Предполагается также, что алгоритм задачи допускает распараллеливание.

Рассмотрим аппаратные особенности типичных вычислительных систем данного класса. Как правило, число процессоров не превышает 40, время задержки обмена (т.е. время между поступлением команды пользователя на пересылку данных и реальным началом этой пересылки) 6-15 мкс, реальная (ощущаемая пользователем) пропускная способность сети, связывающей процессоры, 10-200 Мбайт/с. Данные о некоторых существующих сетях приведены в табл. 1. Более подробную и регулярно обновляемую информацию можно получить на сайте http://parallel.ru. Следует учитывать, что приведенные данные показывают максимальные значения.

Современные процессоры успевают выполнить порядка 10 операций с плавающей точкой за время, необходимое для пересылки одного результата вычислений (на параллельных системах, где узлы соединены сетью Fast Ethernet, – порядка 100 операций). Это правило верно и для очень дорогих вычислительных систем – например, на системе Fujitsu VPP5000 пиковая производительность процессора составляет 9,6 Гфлопс (на реальных задачах, как правило, в два раза меньше), а пиковая пропускная способность сети – 2 по 1,6 Гбайт/с (при одновременном двустороннем обмене). Этот фактор нужно обязательно учитывать при создании параллельных программных комплексов.

Таблица 1

Характеристики типичных сетей, используемых в параллельных вычислительных системах

Тип сети	Время задержки, мкс	Пиковая пропускная способность поверх MPI, Мбайт/с	Где установлена
Fast Ethernet	85 (LAM MPI)	10,5	Кластер ТГУ
Gigabit Ethernet		60	
SCI	7	88	Кластер НИВЦ МГУ
Myrinet	18	140	МВС 1000М, МСЦ
Myrinet 2000	9	210	МВС 1000М, МСЦ с 2002 г.

В классическом учебнике по использованию интерфейса передачи сообщений [1] для распараллеливания многомерных задач математической физики рекомендуется применять двумерное разбиение расчетной области. Однако преимущество двумерного разбиения сильно зависит от задачи, ее размерности и типа используемых обменов. Проиллюстрируем сказанное на примере. Рассмотрим модифицированный тест из упомянутого учебника. Его исходные тексты находятся в файле

ftp://ftp.mcs.anl.gov/pub/mpi/using/UsingMPI.tar.gz.

В этом тесте решается уравнение Пуассона в квадрате итерационным методом Якоби. Аналогичный подход к распараллеливанию применяется и для многих других задач математической физики (например, решения нестационарного уравнения или системы уравнений в частных производных с использованием явной схемы по времени).

Число арифметических операций, приходящееся на один узел расчетной области в исходном тесте, составляет всего 6, поэтому для приближения к реальным задачам тест был усложнен путем введения дополнительных операций. Математически модифицированный тест эквивалентен исходной задаче, так как правая часть уравнения задается равной нулю, но число операций на один узел сетки составляет теперь 23, кроме того, вычисляются 6 раз значения тригонометрических функций. Тексты подпрограмм этого теста при одномерном и двумерном разбиениях

расчетной области приведены на рис. 1 и 2 соответственно. Для обменов используются блокирующие процедуры двустороннего обмена MPI (*MPI_SENDRECV*). Это означает, что выход из подпрограммы обмена осуществляется только после ее завершения.

```
c**************************************************************
c   oned.f – a solution to the Poisson problem using Jacobi
c   iteration on a 1-d decomposition
c
c  The size of the domain is set by processor 0 and broadcast
c  to all other processors.  The Jacobi iteration is run until
c  the change in successive elements is small or a maximum
c  number of iterations is reached.
c**************************************************************
      program main
!
      include "mpif.h"
      integer maxn
      parameter (maxn = 202)
      double precision a(maxn,maxn),b(maxn,maxn),
     $                 f(maxn,maxn)
      integer nx, ny
      integer myid, numprocs, ierr
      integer comm1d, nbrbottom, nbrtop, s, e, it
      double precision diff, diffnorm, dwork
      double precision t1, t2
      external diff

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
c
      if (myid .eq. 0) then
c
c         The size of the problem
c
          nx = 200
      endif
   call MPI_BCAST(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

```
      ny   = nx
c
c Get a new communicator for a decomposition of the domain
c
      call MPI_CART_CREATE( MPI_COMM_WORLD, 1, numprocs,
     $           .false.,.true., comm1d, ierr )
c
c Get my position in this communicator, and my neighbors
c
      call MPI_COMM_RANK( comm1d, myid, ierr )
      call MPI_Cart_shift(comm1d,0,1, nbrbottom, nbrtop, ierr)
c
c Compute the actual decomposition
c
      call MPE_DECOMP1D( ny, numprocs, myid, s, e )
!
! Initialize the right-hand-side (f) and the initial solution
! guess (a)
      call onedinit( a, b, f, nx, s, e )
!
! Actually do the computation.  Note the use of a collective
! operation to check for convergence, and a do-loop to bound
! the number of iterations.
!
      call MPI_BARRIER( MPI_COMM_WORLD, ierr )
      t1 = MPI_WTIME()
      do 10 it=1, 100
        call exchng1(a, nx, s, e, comm1d, nbrbottom, nbrtop )
          call sweep1d( a, f, nx, s, e, b )
          call exchng1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
          call sweep1d( b, f, nx, s, e, a )
          dwork = diff( a, b, nx, s, e )
          call MPI_Allreduce( dwork, diffnorm, 1,
     $           MPI_DOUBLE_PRECISION,MPI_SUM, comm1d, ierr )
          if (diffnorm .lt. 1.0e-5) goto 20
!           if (myid .eq. 0) print *, 2*it, ' Difference is ',
!     &     diffnorm
10     continue
      if (myid .eq. 0) print *, 'Failed to converge'
```

```
20    continue
      t2 = MPI_WTIME()
      if (myid .eq. 0) then
         print *,'Converged after ',2*it,' Iterations in '
     $      , t2 - t1,   ' secs '
      endif
!
      call MPI_FINALIZE(ierr)
      end
!
! Perform a Jacobi sweep for a 1-d decomposition.
! Sweep from a into b
! Modified by M.Tolstykh – increase operation count
!
      subroutine sweep1d( a, f, nx, s, e, b )
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1), f(0:nx+1,s-1:e+1),
     +                 b(0:nx+1,s-1:e+1)
!
      integer i, j
      double precision h,c,x,y,pi2
!
      h = 1.0d0 / dble(nx+1)
      pi2 = asin(1.d0)
      do j = s, e
        do i = 1, nx
         x = dble(i-1)*h*pi2
         y = dble(j-1)*h*pi2
         c = cos(x)**3*cos(y)**2
         b(i,j) = 0.25*c*(a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j))
     +  - h *h* f(i,j)*cos(x**3)*sin(x**2)*cos(y**3)*sin(y**2)
        enddo
      enddo
      return
      end
!
       subroutine exchng1(a, nx, s,e,comm1d, nbrbottom, nbrtop)
!
!   exchanges for 1d decomposition
```

```
!
      include 'mpif.h'
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
!
      call MPI_SENDRECV(
     &         a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
     &     a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
     &           comm1d, status, ierr )
      call MPI_SENDRECV(
     &        a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
     &        a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
     &           comm1d, status, ierr )
      return
      end
c
c  The rest of the 1-d program
c
      double precision function diff( a, b, nx, s, e )
      integer nx, s, e
      double precision a(0:nx+1, s-1:e+1), b(0:nx+1, s-1:e+1)
c
      double precision sum
      integer i, j
c
      sum = 0.0d0
      do 10 j=s,e
        do 10 i=1,nx
          sum = sum + (a(i,j) - b(i,j)) ** 2
 10     continue
c
      diff = sum
      return
      end
      subroutine onedinit( a, b, f, nx, s, e )
      integer nx, s, e
      double precision a(0:nx+1, s-1:e+1), b(0:nx+1, s-1:e+1),
```

```
     &                f(0:nx+1, s-1:e+1)
!
      integer i, j
!
      do 10 j=s-1,e+1
         do 10 i=0,nx+1
            a(i,j) = 0.0d0
            b(i,j) = 0.0d0
            f(i,j) = 0.0d0
 10       continue
c
c     Handle boundary conditions
c
      do 20 j=s,e
         a(0,j) = 1.0d0
         b(0,j) = 1.0d0
         a(nx+1,j) = 0.0d0
         b(nx+1,j) = 0.0d0
 20    continue
      if (s .eq. 1) then
         do 30 i=1,nx
            a(i,0) = 1.0d0
            b(i,0) = 1.0d0
 30       continue
      endif
      return
      end
```

Рис. 1. Распечатка программы решения уравнения Пуассона с одномерным разбиением области

```
c**********************************************************
c   twod.f – a solution to the Poisson problem by using Jacobi
c   iteration on a 2-d decomposition
c
c   Node 0 prints the result.
c
c**********************************************************
```

```
      program main
      include "mpif.h"
      integer maxn
      parameter (maxn = 202)
      double precision a(maxn,maxn),b(maxn,maxn),f(maxn,maxn)
      integer nx, ny
      integer myid, numprocs, it, rc, comm2d, ierr, stride
      integer nbrleft, nbrright, nbrtop, nbrbottom
      integer sx, ex, sy, ey
      integer dims(2)
      logical periods(2)
      double precision diff2d, diffnorm, dwork
      double precision t1, t2
      external diff2d
      data periods/2*.false./

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      if (myid .eq. 0) then
c
c        The size of the problem
c
        nx = 200
      endif
      call MPI_BCAST(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,
     $                    ierr)
      ny   = nx
c
c Get a new communicator for a decomposition of the domain.
C Let MPI find a "good" decomposition
c
      dims(1) = 0
      dims(2) = 0
      call MPI_DIMS_CREATE( numprocs, 2, dims, ierr )
      call MPI_CART_CREATE( MPI_COMM_WORLD, 2, dims,
     $                              periods, .true.,  comm2d, ierr )
c
c Get my position in this communicator
```

```
c
      call MPI_COMM_RANK( comm2d, myid, ierr )
c
c My neighbors are now +/- 1 with my rank. Handle the case
c of the boundaries by using MPI_PROCNULL.
      call fnd2dnbrs(comm2d,nbrleft,nbrright,nbrtop,nbrbottom)
c
c Compute the decomposition
c
      call fnd2ddecomp( comm2d, nx, sx, ex, sy, ey )
c
c Create a new, "strided" datatype for the exchange
c  in the "non-contiguous" direction
c
      call mpi_Type_vector( ey-sy+1, 1, ex-sx+3,
     $                 MPI_DOUBLE_PRECISION, stride, ierr)
      call mpi_Type_commit( stride, ierr )
c
c Initialize the right-hand-side (f) and the initial
c solutionguess (a)
c
      call twodinit( a, b, f, nx, sx, ex, sy, ey )
c
c Actually do the computation. Note the use of a collective
c operation to check for convergence, and a do-loop to bound
c   the number of iterations.
c
      call MPI_BARRIER( MPI_COMM_WORLD, ierr )
      t1 = MPI_WTIME()
      do 10 it=1, 100
       call exchng2( b, sx, ex, sy, ey, comm2d, stride,
     $               nbrleft, nbrright, nbrtop, nbrbottom)
       call sweep2d( b, f, nx, sx, ex, sy, ey, a )
       call exchng2( a, sx, ex, sy, ey, comm2d, stride,
     $               nbrleft, nbrright, nbrtop, nbrbottom)
       call sweep2d( a, f, nx, sx, ex, sy, ey, b )
       dwork = diff2d( a, b, nx, sx, ex, sy, ey )
       call MPI_Allreduce( dwork, diffnorm, 1,
     $    MPI_DOUBLE_PRECISION, MPI_SUM, comm2d, ierr)
```

```
      if (diffnorm .lt. 1.0e-5) goto 20
!     if (myid.eq.0) print *,2*it,' Difference is ',diffnorm
10    continue
      if (myid .eq. 0) print *, 'Failed to converge'
20    continue
      t2 = MPI_WTIME()
      if (myid .eq. 0) then
        print *, 'Converged after ', 2*it, ' Iterations in ',
     $         t2 - t1,    ' secs '
      endif
c
      call MPI_Type_free( stride, ierr )
      call MPI_Comm_free( comm2d, ierr )
      call MPI_FINALIZE(rc)
      end
!  Perform a Jacobi sweep for a 2-d decomposition
!  Modified by M.Tolstykh – increase operation count
!
      subroutine sweep2d( a, f, n, sx, ex, sy, ey, b )
      integer n, sx, ex, sy, ey
      double precision a(sx-1:ex+1, sy-1:ey+1),
     +     f(sx-1:ex+1, sy-1:ey+1),b(sx-1:ex+1, sy-1:ey+1)
!
      integer i, j
      double precision h, c, x, y, pi2
!
      h = 1.0d0 / dble(n+1)
      pi2 = asin(1.d0)
      do j= sy, ey
        do i= sx, ex
          x = dble(i-1)*h*pi2
          y = dble(j-1)*h*pi2
          c = cos(x)**3*cos(y)**2
          b(i,j) =0.25*c*(a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j))
     +   - h*h*f(i,j)*cos(x**3)*sin(x**2)*cos(y**3)*sin(y**2)
        enddo
      enddo
      return
      end
```

```
      subroutine exchng2( a, sx, ex, sy, ey,
     &                    comm2d, stridetype,
     &                    nbrleft, nbrright, nbrtop, nbrbottom )
       include 'mpif.h'
       integer sx, ex, sy, ey, stridetype
       double precision a(sx-1:ex+1, sy-1:ey+1)
       integer nbrleft, nbrright, nbrtop, nbrbottom, comm2d
       integer status(MPI_STATUS_SIZE), ierr, nx
c
       nx = ex - sx + 1
c  These are just like the 1-d versions, except for less data
       call MPI_SENDRECV( a(sx,ey), nx, MPI_DOUBLE_PRECISION,
     &                    nbrtop, 0,
     &                    a(sx,sy-1), nx, MPI_DOUBLE_PRECISION,
     &                    nbrbottom, 0, comm2d, status, ierr )
       call MPI_SENDRECV( a(sx,sy), nx, MPI_DOUBLE_PRECISION,
     &                    nbrbottom, 1,
     &                    a(sx,ey+1), nx, MPI_DOUBLE_PRECISION,
     &                    nbrtop, 1, comm2d, status, ierr )
c
c This uses the vector datatype stridetype
         call MPI_SENDRECV(a(ex,sy), 1, stridetype, nbrright, 0,
     &                    a(sx-1,sy), 1, stridetype, nbrleft, 0,
     &                    comm2d, status, ierr )
         call MPI_SENDRECV( a(sx,sy),  1, stridetype, nbrleft,
     &                    1, a(ex+1,sy), 1, stridetype, nbrright, 1,
     &                    comm2d, status, ierr )
         return
       end
c
c  The rest of the 2-d program
c
       double precision function diff2d(a,b, nx,sx, ex, sy, ey)
       integer nx, sx, ex, sy, ey
       double precision a(sx-1:ex+1, sy-1:ey+1),
     &                  b(sx-1:ex+1, sy-1:ey+1)
c
       double precision sum
       integer i, j
```

```
c
      sum = 0.0d0
      do 10 j=sy,ey
        do 10 i=sx,ex
          sum = sum + (a(i,j) - b(i,j)) ** 2
 10     continue
c
      diff2d = sum
      return
      end
      subroutine twodinit( a, b, f, nx, sx, ex, sy, ey )
      integer nx, sx, ex, sy, ey, i, j
      double precision a(sx-1:ex+1, sy-1:ey+1),
     &       b(sx-1:ex+1, sy-1:ey+1),f(sx-1:ex+1, sy-1:ey+1)
c
      do 10 j=sy-1,ey+1
        do 10 i=sx-1,ex+1
          a(i,j) = 0.0d0
          b(i,j) = 0.0d0
          f(i,j) = 0.0d0
 10     continue
c
c     Handle boundary conditions
c
      if (sx .eq. 1) then
        do 20 j=sy,ey
          a(0,j) = 1.0d0
          b(0,j) = 1.0d0
 20     continue
      endif
      if (ex .eq. nx) then
        do 21 j=sy,ey
          a(nx+1,j) = 0.0d0
          b(nx+1,j) = 0.0d0
 21     continue
      endif
      if (sy .eq. 1) then
        do 30 i=sx,ex
          a(i,0) = 1.0d0
```

```
          b(i,0) = 1.0d0
 30       continue
      endif
      return
      end
c
c This routine show how to determine the neighbors in a 2-d
c  decomposition of the domain. This assumes that
c   MPI_Cart_create has already been called
c
      subroutine fnd2dnbrs( comm2d,
     $                 nbrleft, nbrright, nbrtop, nbrbottom )
      integer comm2d, nbrleft, nbrright, nbrtop, nbrbottom
c
      integer ierr
c
      call MPI_Cart_shift(comm2d, 0,1,nbrleft,nbrright,ierr)
      call MPI_Cart_shift(comm2d, 1,1,nbrbottom,nbrtop,ierr)
c
      return
      end
c
      subroutine fnd2ddecomp( comm2d, n, sx, ex, sy, ey )
      integer comm2d
      integer n, sx, ex, sy, ey
      integer dims(2), coords(2), ierr
      logical periods(2)
c
      call MPI_Cart_get(comm2d,2, dims, periods, coords,ierr)
      call MPE_DECOMP1D( n, dims(1), coords(1), sx, ex )
      call MPE_DECOMP1D( n, dims(2), coords(2), sy, ey )
c
      return
      end
```

Рис. 2. Распечатка программы решения уравнения Пуассона с двумерным разбиением области

Оба теста используют подпрограмму разбиения расчетной области по данной координате из библиотеки расширений MPI *MPE_DECOMP1D* (рис. 3).

```
C ******************************************************************
c  This routine is for producing a decomposition of
c a 1-d array when given a number of processors. It
c may be used in "direct" product  decomposition.
c The values returned assume a "global" domain in [1:n]
c
******************************************************************
      subroutine MPE_DECOMP1D(n,numprocs, myid, s, e)
      integer n, numprocs, myid, s, e
      integer nlocal
      integer deficit
c
      nlocal  = n / numprocs
      s       = myid * nlocal + 1
      deficit = mod(n,numprocs)
      s       = s + min(myid,deficit)
      if (myid .lt. deficit) then
         nlocal = nlocal + 1
      endif
      e = s + nlocal - 1
      if (e .gt. n .or. myid .eq. numprocs-1) e = n
      return
      end
```

Рис. 3. Распечатка программы разбиения расчетной области

Результаты запуска этого теста на кластере НИВЦ МГУ при количестве процессоров от 1 до 8 приведены в табл. 2 (вторая и третья прографка). Параллельное ускорение определяется как отношение продолжительности вычислений на одном процессоре ко времени вычислений на данном числе процессоров.

Таблица 2

Параллельное ускорение для различных версий программы

Кол-во процессоров	Одномерное разбиение	Двумерное разбиение	Одномерное разбиение, асинхронные обмены
2	1,99	1,84	1,92
3	2,67	2,55	2,74
4	3,32	3,2	3,42
6	4,48	4,15	4,84
8	5,18	4,79	5,62

Мы видим, что при данной размерности задачи (200) и небольшом количестве процессоров параллельное ускорение больше для одномерного разбиения расчетной области даже при числе процессоров 2 и 3, когда эти разбиения совпадают. Это вызвано большими накладными расходами на организацию обменов. Накладные расходы включают в себя как аппаратную часть (время задержки начала обмена), так и программную (время работы подпрограмм MPI). Сравним одномерное и двумерное разбиения при числе процессоров, равном 8. В случае одномерного разбиения на каждой итерации рассматриваемого алгоритма всем процессорам необходимо переслать $2\times7\times nx$ чисел, при этом 2×7 раз вызывается подпрограмма обменов. Для двумерного разбиения, когда число процессоров по координате x составляет 4, а по y – 2, общий объем пересылаемой информации $3\times2\times ny+2\times nx$ чисел ($=4\times2\times nx$, когда $nx=ny$), т.е. в 1,75 раз меньше, чем для одномерного случая. Зато при двумерном разбиении необходимо 2×10 обращений к подпрограмме обменов.

Теперь рассмотрим обмены, совершаемые каждым процессором. В одномерном случае на каждой итерации каждый процессор пересылает и получает по $2\times nx$ чисел (объем пересылок $4\times nx$) независимо от числа процессоров, подпрограммы двунаправленных обменов (*MPI_SENDRECV*) вызываются 2 раза. Для двумерного случая объем пересылок для каждого процессора уменьшается при росте числа процессоров и составляет $2\times2\times(nx+ny)/P$, где P – число процессоров ($=8nx/P$, если $nx=ny$), а подпрограммы обменов вызываются 4 раза. Казалось бы, что двумерное разбиение должно быть выгоднее уже при небольшом числе процессоров. Однако надо помнить, что на вычислительных системах рассматриваемого класса одно- или двухпроцессорные вычислительные узлы имеют всего одно соединение с остальными процессорами (через коммутатор) и скорость обмена по нему ограничена. Таким образом, на однопроцессорном вычислительном узле все 4

двухсторонних обмена при двумерном разбиении будут выполняться через одну сетевую карту. Ситуация еще хуже для двухпроцессорного вычислительного узла, где через одну сетевую карту будет выполняться сразу 6 обменов. Очевидно, что реальная скорость заданного обмена при одномерном разбиении будет выше в 2 раза для однопроцессорного узла и в 3 раза для двухпроцессорного. Также влияние накладных расходов на обмены будет всегда больше для двумерного разбиения.

Отметим, что в дорогих вычислительных системах вычислительные узлы имеют несколько соединений. Например, на Cray T3E каждый двухпроцессорный узел имеет 4 связи, топология соединительной сети представляет собой двумерный тор.

Теоретически при заданном числе процессоров по мере увеличения размерности задачи *nx* накладные расходы на каждый обмен останутся неизменными, а время, затрачиваемое на пересылку данных, будет расти. При увеличении числа процессоров и заданной размерности время на пересылку будет сокращаться при двумерном разбиении. Тогда, начиная с некоторой размерности и некоторого числа процессоров, для данной задачи более эффективным должно стать двумерное разбиение области. Однако может оказаться, что эта размерность и число процессоров превосходят разумные для данной задачи и вычислительной системы значения.

Для повышения эффективности распараллеливания необходимо использовать асинхронные обмены, т.е. совершать пересылки данных в приграничных частях на фоне расчетов во внутренних частях подобластей. Для этого используются подпрограммы *MPI_ISEND* и *MPI_IRECV*, также изменяется порядок вычислений в расчетной подобласти: сначала рассчитываются приграничные узлы, затем запускаются обмены вычисленных значений с соседними процессорами. Во время выполнения этих обменов совершаются расчеты в остальной (внутренней) части расчетной области. Проиллюстрируем это на нашем примере. Подпрограмма, объединяющая вычисления и асинхронные обмены для одномерного разбиения, приведена на рис. 4.

```
! Perform a Jacobi sweep for a 1-d decomposition,
! exchange boundary values.
! Sweep from a into b
! by M.Tolstykh from s/r sweep and exch1d
!
      subroutine swexch1(a,f,nx,s,e,b,comm1d,nbrbottom,nbrtop)
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1), f(0:nx+1,s-1:e+1),
```

```
     +                    b(0:nx+1,s-1:e+1)
!
      include ”mpif.h”
      integer i, j,comm1d,nbrbottom,nbrtop
      integer status_arr(MPI_STATUS_SIZE,4),ier,req(4)
      double precision h,c,x,y,pi2
! initialize receive
      call MPI_IRECV(b(1,s-1),nx,MPI_DOUBLE_PRECISION,
     &                    nbrbottom, 0, comm1d,req(1),ier)
      call MPI_IRECV(b(1,e+1),nx,MPI_DOUBLE_PRECISION,
     &                    nbrtop, 1, comm1d,req(2),ier)

      h = 1.0d0 / dble(nx+1)
      pi2 = asin(1.d0)
! calculate boundary values
      do i = 1, nx
        x = dble(i-1)*h*pi2
        y = dble(s-1)*h*pi2
        c = cos(x)**3*cos(y)**2
        b(i,s) =0.25*c*(a(i-1,s)+a(i,s+1)+a(i,s-1)+a(i+1,s))-
     +    h*h* f(i,s)*cos(x**3)*sin(x**2)*cos(y**3)*sin(y**2)
        y = dble(e-1)*h*pi2
        c = cos(x)**3*cos(y)**2
        b(i,e) =0.25*c*(a(i-1,e)+a(i,e+1)+a(i,e-1)+a(i+1,e))-
     +     h*h* f(i,e)*cos(x**3)*sin(x**2)*cos(y**3)*sin(y**2)
      enddo
! initialize send of boundary values
        call MPI_ISEND(b(1,e),nx,MPI_DOUBLE_PRECISION,
     &                     nbrtop,0,comm1d,req(3),ier)
        call MPI_ISEND(b(1,s),nx,MPI_DOUBLE_PRECISION,
     &                    nbrbottom, 1, comm1d,req(4),ier)
! then calculate the rest of the domain
      do j = s+1, e-1
        do i = 1, nx
          x = dble(i-1)*h*pi2
          y = dble(j-1)*h*pi2
          c = cos(x)**3*cos(y)**2
          b(i,j)=0.25*c*(a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j))
     +    -h*h* f(i,j)*cos(x**3)*sin(x**2)*cos(y**3)*sin(y**2)
```

```
      enddo
    enddo
! waiting for completion of exchanges
    call MPI_WAITALL( 4, req, status_arr, ier)
    return
    end
```

Рис. 4. Текст подпрограммы вычислений и асинхронных обменов swexch1

В головной программе первые 4 строки цикла с меткой 10 заменяются двумя вызовами этой подпрограммы:

```
call swexch1(a,f,nx,s,e,b,comm1d,nbrbottom,nbrtop)
call swexch1(b,f,nx,s,e,a,comm1d,nbrbottom,nbrtop)
```

Результаты расчетов по этой программе приведены в табл. 2, четвертая прографка. Мы видим, что эффективность распараллеливания по сравнению с исходной версией программы возросла. Можно использовать асинхронные обмены и для двумерного разбиения расчетной области, но в нашем примере для данной размерности задачи это вряд ли оправдано по причине небольшого числа арифметических операций (во время выполнения которых должна осуществляться пересылка данных).

В случае же решения трехмерных задач или двумерных задач с большим объемом вычислений в промежутках времени между обменами и/или с большой размерностью массивов использование двумерного разбиения области в сочетании с асинхронными обменами, как правило, приводит к большей эффективности распараллеливания. Это утверждение справедливо лишь для данной структуры обменов, когда обмены совершаются только с процессорами, ведущими вычисления в подобластях, которые находятся слева, справа, сверху и снизу от подобласти данного процессора. Если же, например, шаблон разностной схемы требует пересылки данных между процессорами, которые ведут расчеты в подобластях, находящихся по диагонали от данной, то число обменов, совершаемых каждым процессором при двумерном разбиении, вырастет с 4 до 8. Это скорее всего сделает реализацию двумерного разбиения на вычислительных системах рассматриваемого класса невыгодным из-за больших накладных расходов на обмены. Для одномерного разбиения число обменов в этом случае останется неизменным.

Из результатов этого теста можно также сделать вывод, что в случае решения системы уравнений на параллельном компьютере с невысокими

характеристиками связующей сети для уменьшения числа обменов (и, следовательно, затрат на их организацию) полезно все расчетные переменные помещать в один массив. При этом индексы данного массива, соответствующие координатам, по которым производится разбиение области, следует делать последними.

Выводы

Прежде чем приступить к программированию параллельного алгоритма, нужно выбрать оптимальную стратегию распараллеливания, основываясь на технических характеристиках процессоров и связующей сети используемой вычислительной системы, а также на сведениях о свойствах алгоритма, размерности и вычислительной сложности решаемой задачи. При этом можно заранее оценить эффективность параллельной реализации модели. Для достижения высокой эффективности распараллеливания следует использовать асинхронные обмены данными между процессорами. Это может потребовать изменения расчетных подпрограмм исходной последовательной версии программы.

Список литературы

1. *Gropp W., Lusk E., Skjellum A*. Using MPI-2: Portable Parallel Programming with the Message-Passing Interface. – 1999. – MIT Press. – 382 p.
2. http://parallel.ru/vvv/mpi.html
3. http://www-unix.mcs.anl.gov/mpi/index.html