Основы параллельного программирования с использованием MPI

Лекция 4

Немнюгин Сергей Андреевич

Санкт-Петербургский государственный университет

кафедра вычислительной физики

snemnyugin@mail.ru

Интернет-Университет
Суперкомпьютерных Технологий
High-Performance Computing University

Лекция 4

Аннотация

В лекции рассматривается простая вычислительная задача и реализация параллельного алгоритма её решения. Далее мы познакомимся со средствами организации неблокирующих двухточечных обменов. Рассматриваются операции неблокирующих отправки и приёма сообщений, процедуры-пробники, отложенные обмены. Даются примеры использования как блокирующих, так и неблокирующих двухточечных операций.

План лекции

- Пример использования блокирующих двухточечных обменов
- Общая характеристика неблокирующих обменов.
- Неблокирующие передача и приём.
- Проверка выполнения неблокирующих обменов.
- Пробники.
- Отложенные обмены.

Одномерное уравнение Лапласа. Метод Якоби

Уравнение Лапласа

В качестве примера использования операций двухточечного обмена рассмотрим численное решение задачи Дирихле для уравнения Лапласа в одномерном и двумерном случаях:

$$\Delta u = 0$$

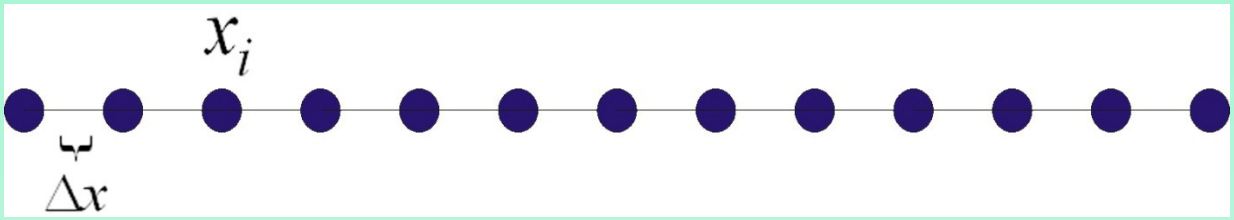
$$u\,|_{\Gamma} = \varphi$$

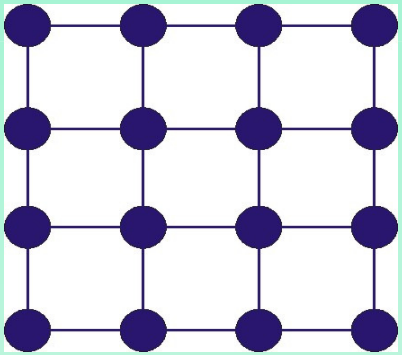
где φ - значение решения на границе области,

Δ - оператор Лапласа:

$\Delta = \frac{\partial^2}{\partial x^2}$ - одномерный; $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ - двумерный.

Численное решение основано на введении 1- или 2-мерной сетки в области, ограниченной границей Г





Метод Якоби является итерационным методом. Сначала задаются произвольные значения функции u во внутренних узлах сетки, затем выполняются итерации (1-мерный случай):

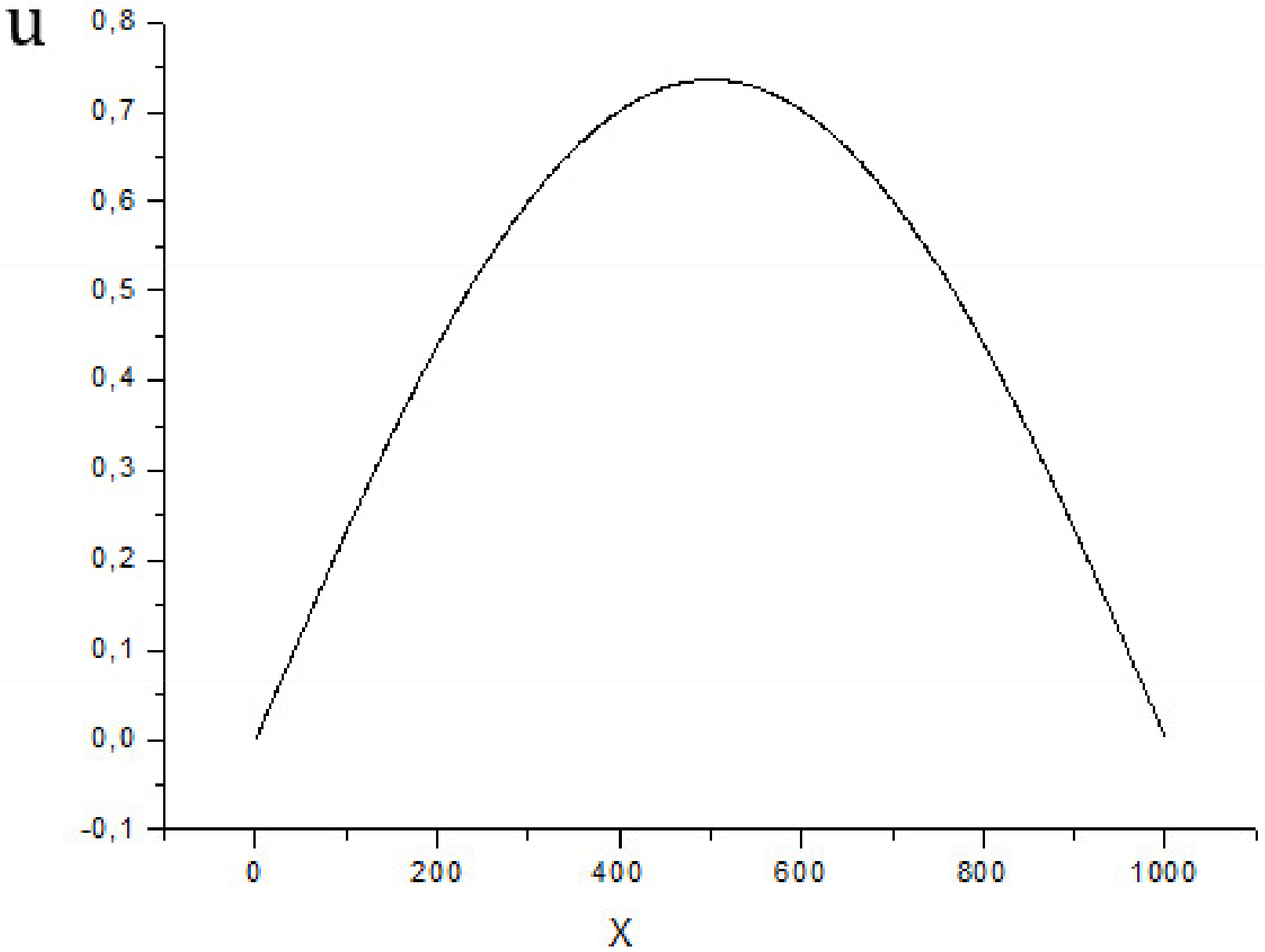
$$u_i^{(k+1)} = \frac{1}{2}\left(u_{i-1}^{(k)} + u_{i+1}^{(k)} - \Delta x^2 u_i^{(k)}\right)$$

где $u_i^{(k)} = u^{(k)}(x_i)$ - значение u в i-м узле, полученное на k-й итерации.

Последовательная программа на языке Fortran 90

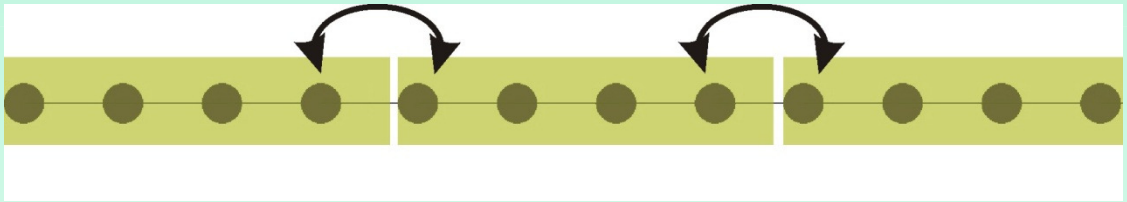
```
program jacobi_serial
implicit none

real, dimension(0:10001) :: x, newx
real :: dx2
integer :: n, noiters, i, k

 open(unit = 12, file = "laplace1d.in")
! Количество узлов
 read(12,  *)  n
! Количество итераций
 read(12,  *)  noiters
 close(12)

dx2 = (1. / n)**2

do i = 1, n
 x(i) = 1.0
enddo
```

```
x(0) = 0.0
x(n + 1) = 0.0

do k = 1, noiters
 do i = 1, n
  newx(i) = 0.5 * (x(i - 1) + x(i + 1) - dx2 * x(i))
 enddo
 do i =1, n
  x(i) = newx(i)
 enddo

enddo

 open(unit = 11, file = "laplace1d_serial.dat", status = "NEW")
  write(11, "(2x, e8.3)") (x(i),   i = 1, n)
 close(11)

end
```


u
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
-0,1
0
200
400
600
800
1000
X

Параллельный алгоритм

Параллельный алгоритм основан на декомпозиции по данным – разбиении одномерной сетки на одинаковые части. Каждая часть обрабатывается на отдельном процессоре. Обмен заключается в пересылке значений функции в граничных узлах. Он может быть организован с помощью операций двухточечного обмена:



Параллельная программа на языке Fortran 90

```
program jacobi_parallel
implicit none
include "mpif.h"
real, dimension(0:10001) :: x,newx
real :: dx2
integer :: n, noiters, i, k
integer :: p, me, ln, tag, ierr
integer, dimension(MPI_STATUS_SIZE) :: status
! Ввод исходных данных
 open(unit = 12, file = "laplace1d.in")
! Количество узлов
 read(12,  *)  n
! Количество итераций
 read(12,  *)  noiters
 close(12)

dx2 = (1. / n)**2
```

```
call MPI_Init(ierr)

call MPI_Comm_rank(MPI_COMM_WORLD, me, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

tag = 0
ln = n / p

do i = 1, ln
 x(i) = 1.0
enddo

if(me = = 0) then
x(0) = 0.0; lm = 0
else
lm = ln * me
endif

if(me = = p - 1) then
x(ln + 1) = 0.0
endif
```

```
do k = 1, noiters
! Пересылки граничных значений
 if(me - 1 >= 0) call MPI_Send(newx(1), 1, MPI_REAL, me - 1, tag,
MPI_COMM_WORLD, ierr)
 if(me + 1 < p)  call MPI_Recv(x(ln + 1), 1, MPI_REAL, me + 1, tag,
MPI_COMM_WORLD, status, ierr)
 tag = tag + 1
 if(me + 1 < p)  call MPI_Send(newx(ln), 1, MPI_REAL, me + 1, tag,
MPI_COMM_WORLD, ierr)
 if(me - 1 >= 0) call MPI_Recv(x(0), 1, MPI_REAL, me - 1, tag,
MPI_COMM_WORLD, status, ierr)
 tag = tag + 1
! Итерации Якоби
 do i = 1, ln
  newx(i) = 0.5 * (x(i - 1) + x(i + 1) - dx2 * x(i))
 enddo

 do i = 1, ln
  x(i) = newx(i)
 enddo
enddo
```

```
! Собираем решение
if(me = = 0) then
  do i = 1, ln
   z(i) = x(i)
  enddo
  do k = 1, p - 1
   lm = ln * k
   call MPI_Recv(z(lm), ln, MPI_REAL, k, k, MPI_COMM_WORLD, status, ierr)
  enddo
 else
   call MPI_Send(x(1), ln, MPI_REAL, 0, me, MPI_COMM_WORLD, ierr)
 endif

call MPI_Finalize(ierr)
! Запись результата в файл
if(me = = 0) then
 open(unit = 11, file = "laplace1d_parallel.dat", status = "NEW")
  write(11, "(2x, e8.3)") (z(i),   i = 1, n)
 close(11)
endif
end
```

Двумерное уравнение Лапласа. Последовательная программа на языке Fortran 90

laplace2d_serial.f90

```
program laplace
 implicit none
 integer :: nx, ny, i, j, iter
 real(8) :: v0, v1, change
 integer, parameter :: ndim = 100
 real, dimension(ndim, ndim) :: v
 open(unit = 12, file = "laplace.in")
 ! Количество узлов вдоль x
 read(12,  *)  nx
 ! Количество узлов вдоль y
 read(12,  *)  ny
 ! Граничное значение OX
 read(12,  *)  v0
 ! Граничное значение OY
 read(12,  *)  v1
 ! Относительная погрешность
 read(12, *) change
 close(12)
```

```
 change = change / 100
! Граничные значения
 boundary_potential_x : do i = 1, nx
  v(i, 1) = v0 ; v(i, ny) = v0
 end do boundary_potential_x

 boundary_potential_y : do j = 1, ny
  v(1, j) = v1 ; v(nx, j) = v1
 end do boundary_potential_y
! Начальное приближение для внутренних узлов
 initial_values : do i = 2, nx -  1
  do j = 2, ny - 1
   v(i, j) = 0.9d0 * v0
  end do
 end do initial_values

call relax(v, nx, ny, change, iter)
end
```

```
subroutine relax(v, nx, ny, change, iter)
 implicit none
 integer :: nx, ny, i, j, iter, idum
 real(8) :: v0, change, diff, dmax
 integer, parameter :: ndim = 100
 real, dimension(ndim, ndim) :: v, vaverage
 iter =  0
 iterations : do idum = 1, 100000
  dmax = 0
  iter = iter + 1
 do i = 2, nx -  1
    average_potential : do j = 2, ny -  1
    vaverage(i, j) = v(i + 1, j) + v(i - 1, j)
    vaverage(i, j) = vaverage(i, j) + v(i, j + 1) + v(i, j - 1)
    vaverage(i, j) = 0.25d0 * vaverage(i, j)
    diff = abs((v(i, j) - vaverage(i, j)) / vaverage(i, j))
     if (diff > dmax) dmax = diff
    end do average_potential
   end do
```

```
    x_loop : do i = 2, nx -  1
    y_loop : do j = 2, ny -  1
     v(i, j) = vaverage(i, j)
    end do y_loop
   end do x_loop
   if (dmax < change) then
   call output(v, nx, ny, iter)
   return
   end if
end do iterations
return
end
```

```
subroutine output(v, nx, ny, iter)
 implicit none
 integer :: nx, ny, i, j, iter
 integer, parameter :: ndim = 100
 real, dimension(ndim, ndim) :: v, vaverage
 write(6, *)  'Number of iterations = ',   iter
 open(unit = 11, file = "laplace.dat", status = "NEW")
 do j = ny, 1, -1
  write(11, "(10(d8.3, 2x))") (v(i, j),   i = 1, nx)
 end do
 close(11)
 return
end
```

Задание для самостоятельной работы

Задание 1

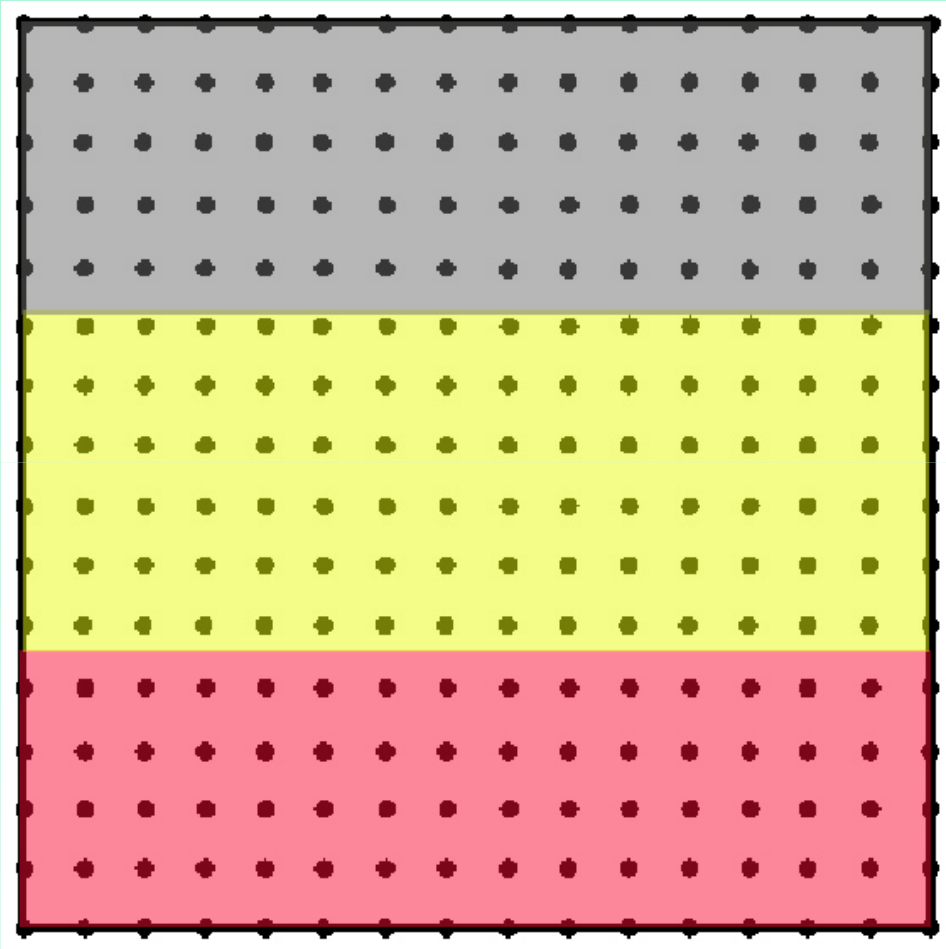
Имеется последовательная программа на языке Fortran 90 решения двумерного уравнения Лапласа методом Якоби.

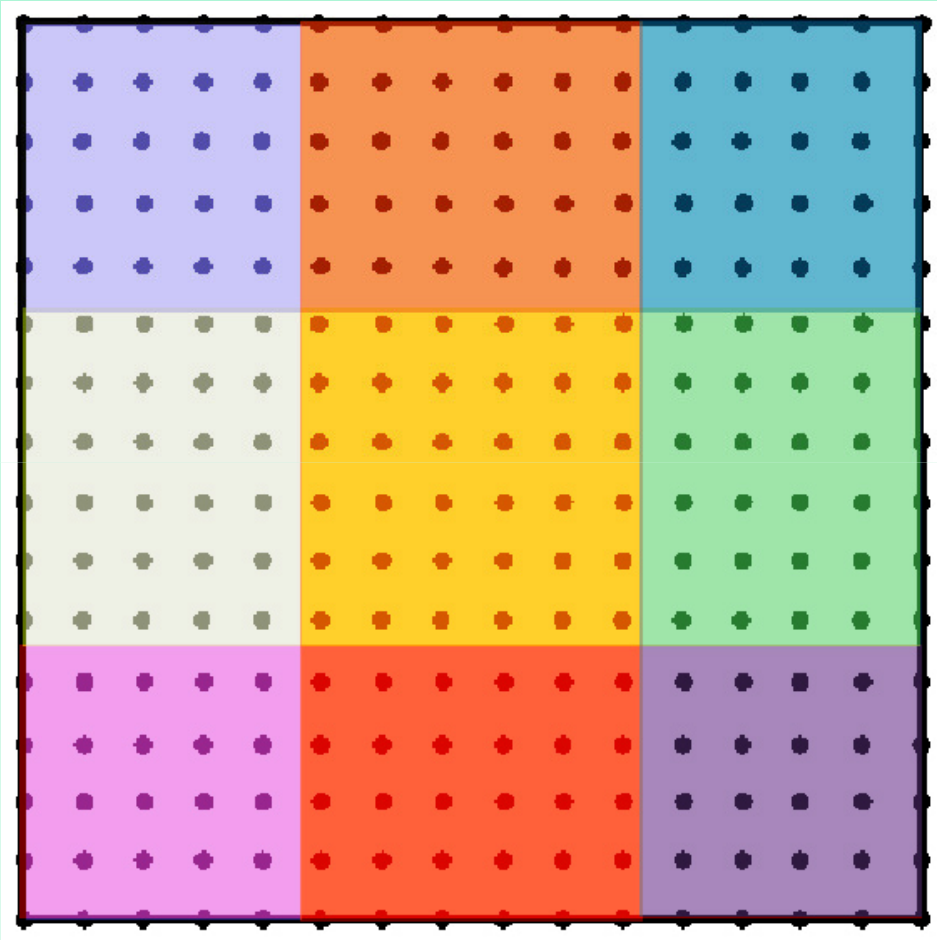
Ниже приводится результат исследования с помощью анализатора Intel ® VTune Performance Analyzer.

```
0x2dab  57          653   291      vaverage(i, j) = v(i + 1, j) + v(i - 1, j)
0x2e09  58          894   276      vaverage(i, j) = vaverage(i, j) + v(i, j + 1) + v(i, j - 1)
0x2e83  59          700   454      vaverage(i, j) = 0.25d0 * vaverage(i, j)
        60            0     0      !
        61            0     0      ! Relative change of potential
        62            0     0      !
0x2ec9  63         4781  9141     diff = abs((v(i, j) - vaverage(i, j)) / vaverage(i, j))
0x2f24  64          789   360      if (diff > dmax) dmax = diff
0x2f38  65          294   105     end do average_potential
0x2f4a  66           19     8      end do
        67            0     0    !
        68            0     0    ! Update potential of each cell
        69            0     0    !
0x2f5c  70            0     0      x_loop : do i = 2, nx -  1
0x2f77  71            4    10      y_loop : do j = 2, ny -  1
0x2f92  72          885  1311      v(i, j) = vaverage(i, j)
0x2fca  73          369   527     end do y_loop
0x2fd8  74           17    37     end do x_loop
0x2fe6  75            0     0      if (dmax < change) then
0x2ff5  76            0     0      call output(v, nx, ny, iter)
```

Задание для самостоятельной работы

Написать параллельный вариант этой программы.
Применить декомпозицию по данным.
Обмен значениями функции в граничных узлах подобластей.





Неблокирующие двухточечные обмены

Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи.

Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (то есть одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы.

Для завершения неблокирующего обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи.

ВНИМАНИЕ!
При неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, но запись в буфер или считывание из него после этого производить **нельзя** - сообщение может быть еще не отправлено или не получено и работа с буфером может «испортить» его содержимое.

Неблокирующий обмен выполняется в два этапа:

1. **инициализация** обмена;
2. **проверка завершения** обмена.

Разделение этих шагов делает необходимым *маркировку* каждой операции обмена, которая позволяет целенаправленно выполнять проверки завершения соответствующих операций.

Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена*

Инициализация неблокирующей стандартной передачи выполняется подпрограммами `MPI_I[S, B, R]send`. Стандартная неблокирующая передача выполняется подпрограммой:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Isend(buf, count, datatype, dest, tag, comm, request,
ierr)
```

Входные параметры этой подпрограммы аналогичны аргументам подпрограммы `MPI_Send`.
Выходной параметр `request` - идентификатор операции.

Инициализация неблокирующего приема выполняется при вызове подпрограммы:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Irecv(buf, count, datatype, source, tag, comm, request,
ierr)
```

Назначение аргументов здесь такое же, как и в ранее рассмотренных подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (`source`).

Вызовы подпрограмм неблокирующего обмена формируют *запрос* на выполнение операции обмена и связывают его с идентификатором операции `request`. Запрос идентифицирует свойства операции обмена:

- режим;
- характеристики буфера обмена;
- контекст;
- тег и ранг.

Запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения.

Проверка выполнения обмена

Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова *подпрограмм ожидания*, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение «истина», если операция выполнена

В том случае, когда одновременно несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам.

Есть три типа таких проверок:

1. проверка завершения всех обменов;
2. проверка завершения любого обмена из нескольких;
3. проверка завершения заданного обмена из нескольких.

Каждая из этих проверок имеет две разновидности:

1. «ожидание»;
2. «проверка».

Блокирующие операции проверки

Подпрограмма `MPI_Wait` блокирует работу процесса до завершения приема или передачи сообщения:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_Wait(request, status, ierr)
```

Входной параметр `request` — идентификатор операции обмена, выходной — статус (`status`).

Успешное выполнение подпрограммы `MPI_Wait` после вызова `MPI_Ibsend` подразумевает, что буфер передачи можно использовать вновь, то есть пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы `MPI_Buffer_attach`. В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму `MPI_Cancel`, которая освобождает память, выделенную подсистеме коммуникаций.

Проверка завершения всех обменов

Проверка завершения всех обменов выполняется подпрограммой:

```
int MPI_Waitall(int count, MPI_Request requests[],
MPI_Status statuses[])

MPI_Waitall(count, requests, statuses, ierr)
```

При вызове этой подпрограммы выполнение процесса блокируется до тех пор, пока **все** операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` - количество запросов на обмен (размер массивов `requests` и `statuses`).

В результате выполнения подпрограммы `MPI_Waitall` запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение `MPI_REQUEST_NULL`.

В случае неуспешного выполнения одной или более операций обмена подпрограмма `MPI_Waitall` возвращает код ошибки `MPI_ERR_IN_STATUS` и присваивает полю ошибки статуса значение кода ошибки соответствующей операции.

Если операция выполнена успешно, полю присваивается значение `MPI_SUCCESS`, а если не выполнена, но и не было ошибки - значение `MPI_ERR_PENDING`. Это соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Проверка завершения любого числа обменов

Проверка завершения любого числа обменов выполняется подпрограммой:

```
int MPI_Waitany(int count, MPI_Request requests[], int
*index, MPI_Status *status)

MPI_Waitany(count, requests, index, status, ierr)
```

Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.
Входные параметры:

- `requests` - запрос;
- `count` - количество элементов в массиве `requests`.

Выходные параметры:

- `index` - индекс запроса (в языке C это целое число от 0 до `count` – 1, а в языке Fortran от 1 до `count`) в массиве `requests`;
- `status` - статус.

Если в списке вообще нет активных запросов или он пуст, вызовы завершаются сразу со значением индекса `MPI_UNDEFINED` и пустым статусом.

Неблокирующие процедуры проверки

Подпрограмма `MPI_Test` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status
*status)

MPI_Test(request, flag, status, ierr)
```

Входной параметр: идентификатор операции обмена `request`.
Выходные параметры:

- `flag` — «истина», если операция, заданная идентификатором `request`, выполнена;
- `status` — статус выполненной операции.

Неблокирующая проверка завершения всех обменов

Подпрограмма `MPI_Testall` выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

```
int MPI_Testall(int count, MPI_Request requests[], int
*flag, MPI_Status statuses[])

MPI_Testall(count, requests, flag, statuses, ierr)
```

При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
Параметр `count` - количество запросов.
Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.

Неблокирующая проверка любого числа обменов

Подпрограмма `MPI_Testany` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Testany(int count, MPI_Request requests[], int
*index, int *flag, MPI_Status *status)

MPI_Testany(count, requests, index, flag, status, ierr)
```

Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы `MPI_Waitany`. Дополнительный аргумент `flag`, принимает значение «истина», если одна из операций завершена.

Блокирующая подпрограмма `MPI_Waitany` и неблокирующая `MPI_Testany` взаимозаменяемы, как и другие аналогичные пары.

Другие операции проверки

Подпрограммы `MPI_Waitsome` и `MPI_Testsome` действуют аналогично подпрограммам `MPI_Waitany` и `MPI_Testany`, кроме случая, когда завершается более одного обмена. В подпрограммах `MPI_Waitany` и `MPI_Testany` обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для `MPI_Waitsome` и `MPI_Testsome` статус возвращается для всех завершенных обменов. Эти подпрограммы можно использовать для определения, сколько обменов завершено.

Интерфейс этих подпрограмм:

```
int MPI_Waitsome(int incount, MPI_Request requests[], int
*outcount, int indices[], MPI_Status statuses[])

MPI_Waitsome(incount, requests, outcount, indices,
statuses, ierr)
```

Здесь `incount` - количество запросов. В `outcount` возвращается количество выполненных запросов из массива `requests`, а в первых `outcount` элементах массива `indices` возвращаются индексы этих операций. В первых `outcount` элементах массива `statuses` возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру `outcount` присваивается значение `MPI_UNDEFINED`.

Неблокирующая проверка выполнения обменов

```
int MPI_Testsome(int incount, MPI_Request requests[], int
*outcount, int indices[], MPI_Status statuses[])

MPI_Testsome(incount, requests, outcount, indices,
statuses, ierr)
```

Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Примеры использования неблокирующих двухточечных обменов

Пример 1

```
program main_mpi
include 'mpif.h'
integer rank, tag, cnt, ierr, status(MPI_STATUS_SIZE)
integer request
real sndbuf(5) /1., 2., 3., 4., 5./
real rcvbuf(5)
cnt = 5
tag = 0
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
…
```

```
if(rank.eq.0) then
call MPI_Isend(sndbuf(1), cnt, MPI_REAL, 1, tag,
                  MPI_COMM_WORLD, request, ierr)
print *, "process ", rank, " send before Wait", sndbuf
call MPI_Wait(request, status, ierr)
print *, "process ", rank, " send after Wait", sndbuf
else
call MPI_Irecv(rcvbuf(1), cnt, MPI_REAL, 0, tag,
                  MPI_COMM_WORLD, request, ierr)
print *, "process ", rank, " received before Wait", rcvbuf
call MPI_Wait(request, status, ierr)
print *, "process ", rank, " received after Wait", rcvbuf
end if
call MPI_Finalize(ierr)
stop
end
```

Результат выполнения:

```
[nemnugin@pd00 ~]$ mpiexec -n 2 ./a.out
 PROCESS  1 RECEIVED BEFORE WAIT  2.83748136E-08  0.  1.01862059E+31
  7.00649232E-44  0.
 PROCESS  0 SEND BEFORE WAIT  1.  2.  3.  4.  5.
 PROCESS  0 SEND AFTER WAIT  1.  2.  3.  4.  5.
 PROCESS  1 RECEIVED AFTER WAIT  1.  2.  3.  4.  5.
[nemnugin@pd00 ~]$
```

Пример 2

```
program main_mpi
include 'mpif.h'
integer rank, tag1, tag2, cnt, ierr, status(MPI_STATUS_SIZE)
integer request
real sndbuf1, sndbuf2, rcvbuf1, rcvbuf2
cnt = 1
tag = 0
sndbuf1 = 3.14159
sndbuf2 = 2.71828
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
…
```

```
if (rank.eq.0) then
call MPI_Send(sndbuf1, cnt, MPI_REAL, 1, tag1, MPI_COMM_WORLD,
                    ierr)
print *, "process ", rank, " send ", sndbuf1
call MPI_Send(sndbuf2, cnt, MPI_REAL, 1, tag2, MPI_COMM_WORLD,
                    ierr)
print *, "process ", rank, " send ", sndbuf2
else
call MPI_Irecv(rcvbuf1, cnt, MPI_REAL, 0, tag1, MPI_COMM_WORLD,
                    request, ierr)
call MPI_Recv(rcvbuf2, cnt, MPI_REAL, 0, tag2, MPI_COMM_WORLD,
                    status, ierr)
print *, "process ", rank, " received before Wait", rcvbuf1
print *, "process ", rank, " received before Wait", rcvbuf2
call MPI_Wait(request, status, ierr)
print *, "process ", rank, " received after Wait", rcvbuf1
print *, "process ", rank, " received after Wait", rcvbuf2
end if
call MPI_Finalize(ierr)
end
```

Результат выполнения:

```
[nemnugin@pd00 ~]$ mpiexec -n 2 ./a.out
 PROCESS   0 SEND    3.14159012
 PROCESS   0 SEND    2.71828008
 PROCESS   1 RECEIVED BEFORE WAIT   3.14159012
 PROCESS   1 RECEIVED BEFORE WAIT   2.71828008
 PROCESS   1 RECEIVED AFTER WAIT   3.14159012
 PROCESS   1 RECEIVED AFTER WAIT   2.71828008
[nemnugin@pd00 ~]$
```

Подпрограммы-пробники

Неблокирующая проверка сообщения

Неблокирующая проверка сообщения выполняется подпрограммой:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status)

MPI_Iprobe(source, tag, comm, flag, status, ierr)
```

Входные параметры этой подпрограммы те же, что и у подпрограммы `MPI_Probe`. Выходные параметры:

- `flag` - флаг;
- `status` - статус.

Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

Размер полученного сообщения (`count`) можно определить с помощью вызова подпрограммы

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int
*count)

MPI_Get_count(status, datatype, count, ierr)
```

Параметры:

- `count` - количество элементов в буфере передачи;
- `datatype` - тип каждого пересылаемого элемента;
- `status` - статус обмена;
- `ierr` - код завершения.

Аргумент `datatype` должен соответствовать типу данных, указанному в операции обмена.

Пример 3

```
program main_mpi
include 'mpif.h'
integer rank, i, k, ierr, tag, dest, status(MPI_status_size)
real x
tag = 0
dest = 2
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
if (rank.eq.0) then
 i = 2002
 call MPI_Send(i, 1, MPI_INTEGER, dest, tag, MPI_COMM_WORLD,
                    ierr)
else if(rank.eq.1) then
x = 3.14159
call MPI_Send(x, 1, MPI_REAL, dest, tag, MPI_COMM_WORLD, ierr)
…
```

```
do k = 1, 2
 call MPI_Probe(MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, status,
                ierr)
 if (status(MPI_source).eq.0) then
 call MPI_Recv(i, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD,
                status, ierr)
 print *, "received ", i, " from 0"
else
 call MPI_Recv(x, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, status,
                ierr)
 print *, "received ", x, " from 1"
 end if
end do
end if
call MPI_Finalize(ierr)
stop
end
```

Результат выполнения:

```
[nemnugin@pd00 ~]$ mpiexec -n 3 ./a.out
 Received   3.14159012 from 1
 Received  2002 from 0
[nemnugin@pd00 ~]$
```

Отложенные обмены

Достаточно часто приходится сталкиваться с ситуацией, когда обмены с одинаковыми параметрами выполняются повторно, например, в цикле. В этом случае можно объединить аргументы подпрограмм обмена в один отложенный запрос, который затем повторно используется для инициализации и выполнения обмена сообщениями.

Отложенный запрос на выполнение неблокирующей операции обмена позволяет минимизировать накладные расходы на организацию связи между процессором и контроллером связи.

Отложенные запросы на обмен объединяют такие сведения об операциях обмена, как адрес буфера, количество пересылаемых элементов данных, их тип, ранг адресата, тег сообщения и коммуникатор.

Запрос для стандартной передачи создается при вызове подпрограммы `MPI_Send_init`:

```
int MPI_Send_init(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm, MPI_Request
*request)

MPI_Send_init(buf, count, datatype, dest, tag, comm, request,
ierr)
```

Отложенный запрос может быть сформирован для всех режимов обмена. Для этого используются подпрограммы `MPI_Bsend_init`, `MPI_Ssend_init` и `MPI_Rsend_init`.

Отложенный обмен инициируется вызовом подпрограммы `MPI_Start`:

```
int MPI_Start(MPI_Request *request)

MPI_Start(request, ierr)
```

Подпрограмма `MPI_Startall`:

```
int MPI_Startall(int count, MPI_request *requests)
```

```
MPI_Startall(count, requests, ierr)
```

инициирует все обмены, связанные с запросами на выполнение неблокирующей операции обмена в массиве `requests`.

Завершается обмен при вызове `MPI_Wait, MPI_Test` и некоторых других подпрограмм.

Заключение

В этой лекции мы рассмотрели:

- примеры использования двухточечных обменов;
- особенности двухточечных неблокирующих обменов;
- реализацию неблокирующих двухточечных обменов в MPI;
- использование подпрограмм-пробников;
- отложенные обмены.

Задания для самостоятельной работы

Решения следует высылать по электронной почте:

snemnyugin@mail.ru

Задания для самостоятельной работы

Задание 2

Разберите работу следующей программы. Запустите ее на выполнение.

home_task.cpp

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 int myid, numprocs, **buf, source, i;
 int message[3] = {0, 1, 2};
 int myrank, data = 2002, count, TAG = 0;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0)
{
 MPI_Send(&data, 1, MPI_INT, 2, TAG, MPI_COMM_WORLD);
}
else if (myrank == 1) {
 MPI_Send(&message, 3, MPI_INT, 2, TAG, MPI_COMM_WORLD);
}
…
```

```
else
{
MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
source = status.MPI_SOURCE;
MPI_Get_count(&status, MPI_INT, &count);
for (i = 0; i < count; i++){
buf[i] = (int *)malloc(count*sizeof(int));
}
MPI_Recv(&buf[0], count, MPI_INT, source, TAG,
MPI_COMM_WORLD, &status);
for (i = 0; i < count; i++){
printf("received: %d\n", buf[i]);
}
 }
 MPI_Finalize();
 return 0;
}
```

Тема следующей лекции

Коллективные обмены в MPI