ФИЗИЧЕСКИЙ
ФАКУЛЬТЕТ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

ЛУКЬЯНЕНКО
ДМИТРИЙ ВИТАЛЬЕВИЧ

ФИЗФАК МГУ

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТОВ ФИЗИЧЕСКОГО ФАКУЛЬТЕТА МГУ
ГОРБАЧЕВА АЛЕКСАНДРА ВИКТОРОВИЧА
ШИНКАРЕВА ВАЛЕНТИНА ДМИТРИЕВИЧА

Оглавление

Лекция 1. Введение в основы MPI на Python.

С развитием технического прогресса в нынешнее время все чаще и чаще возникают такие задачи, которые на однопроцессорных системах решаются крайне долго. Чтобы сократить время вычисления, обычно прибегают к следующему подходу: стараются решаемую большую задачу или её подзадачи разбить на большое количество более мелких задач, каждую из которых можно было бы обсчитывать независимо друг от друга на различных вычислительных узлах, что, обычно, можно делать параллельно.

Приведём несколько примеров конфигураций систем с несколькими вычислительными узлами. На рис 1.1 изображён кластер Leo, запущенный в научно-исследовательском вычислительном центре МГУ в 2003 году. Свою первую параллельную программу автор лекций запустил именно на этом кластере. По сути, кластер представлял из себя связанный сетью набор системных блоков, каждый из которых содержал 2 мощных, по тем временам, процессора и оперативную память. Он являлся системой *с распределённой памятью*, то есть системой, в которой каждый процессор имеет свою локальную оперативную память, а общей памяти нет.

Рис. 1.1: Кластер Leo (2003),
НИВЦ МГУ имени М.В. Ломоносова

Другим примером будет являться суперкомпьютер ”Ломоносов-2”. На рис. 1.2 приведён пример всего лишь одной стойки этого суперкомпьютера, и, как вы можете видеть, внутри неё содержится более 200 плат, на которых расположены мощнейшие многоядерные процессоры, оперативная память и современные вычислительные видеокарты. То есть это совсем другой технологический уровень и гораздо более сложная конфигурация

узлов.

Рис. 1.2: Суперкомпьютер ”Ломоносов-2”(2020),
НИВЦ МГУ имени М.В. Ломоносова

В качестве третьего примера можно привести ПК читателя данного конспекта. Сейчас практически у каждого на компьютере стоит многоядерный процессор, который можно вопринимать как многопроцессорную систему *с общей памятью*.

Данный курс посвящен различным параллельным алгоритмам решения некоторых типовых задач и программной реализации этих алгоритмов. Это связано с тем, что даже самый простой параллельный алгоритм при реализации может содержать множество трудностей и подводных камней, и разбору этих особенностей, в основном, и посвящен этот курс.

При использовании систем с распределённой памятью каждый вычислительный узел производит вычисления своей подзадачи, при этом им необходимо иногда обмениваться сообщениями. На данный момент для обмена сообщениями крайне популярна технология *MPI (Message Passing Interface)*, её мы будем использовать в параллельных программах данного курса.

Также для распараллеливания используют видеокарты, наиболее популярной технологией для этой задачи является CUDA. В данном случае каждая видеокарта рассматривается как многопроцессорная система с общей памятью, что влечёт за собой некоторые особенности. Нужно отметить, однако, что, если требуется проводить вычисления на нескольких видеокартах, для их взаимодействия вновь нужно использовать технологию MPI.

Технология MPI

Существует большое количество реализаций MPI для таких языков программирования, как C/C++ и Fortran. Для других языков программирования существуют пакеты, позволяющие использовать технологию MPI, обращаясь напрямую к функциям из C/Fortran.

Например, для языка Python существует пакет `mpi4py`, который мы и будем использовать. Мы используем Python из-за того, что он удобен для прототипирования, полученную программу будет гораздо легче отлаживать, а пакет `mpi4py` предоставляет интерфейс для MPI, крайне похожий на реализации MPI для C/Fortran, поэтому соответсвующие программы будет несложно переписать на C/Fortran.

Для запуска программ необходимо поставить на свой ПК одну из библиотек MPI (например, MPICH или OpenMPI), Python и пакеты `numpy` и `mpi4py`. После этого, запуск скрипта `script.py` на N узлах осуществляется командой

`mpiexec -n` N `python script.py`

Итерационный метод решения СЛАУ

В качестве первого алгоритма для распараллеливания рассмотрим итерационный алгоритм для решения системы линейных уравнений $Ax = b$, где A – матрица, x – столбец решений длины N.

Алгоритм (метод сопряжённых градиентов):

$$
\begin{aligned}
r^{(s)} &= \begin{cases} A^T\big(Ax^{(s)} - b\big), & \text{при} \quad s = 1, \\ r^{(s-1)} - \dfrac{q^{(s-1)}}{\big(p^{(s-1)}, q^{(s-1)}\big)}, & \text{при} \quad s \geqslant 2, \end{cases} \\
p^{(s)} &= p^{(s-1)} + \frac{r^{(s)}}{\big(r^{(s)}, r^{(s)}\big)}, \\
q^{(s)} &= A^T\big(Ap^{(s)}\big), \\
x^{(s+1)} &= x^{(s)} - \frac{p^{(s)}}{\big(p^{(s)}, q^{(s)}\big)},
\end{aligned}
\tag{1.1}
$$

где

$x^{(s)}$– последовательность приближённых решений,

$r^{(s)}, p^{(s)}, q^{(s)}$– вспомогательные вектора,

$p^{(0)} = 0$,

$x^{(1)}$– произвольное начальное приближение, возьмём $x = 0$,

$x^{(N)}$– решение СЛАУ.

Данный метод применим для систем с произвольной плотно-заполненной матрицей (как квадратной, так и прямоугольной). Данная задача часто возникает при обработке данных физического эксперимента, где число уравнений (число измерений) может значительно превышать число неизвестных N. После применения алгоритма мы получаем x_N – точное решение для квадратной матрицы, если не делаются ошибки машинного округления в процессе счёта. Если A – прямоугольная матрица, а b содержит ошибки эксперимента, то x_N – приближённое решение, найденное методом наименьших квадратов. Более подробно про этот метод можно почитать, например, в курсе лекций "Численные методы" в лекции 9 (теория) и лекции 26 (пример применения для реальной задачи).

Для распараллеливания этого алгоритма мы потратим несколько последующих лекций. Рассмотрим наиболее вычислительно ёмкие операции алгоритма:

Скалярное произведение:

Для вычисления скалярного произведения векторов длины N, как видно из формулы $(p, q) = \sum_{k=1}^{N} p_k q_k$, необходимо выполнить N операций умножения и $N - 1$ операцию сложения, то есть порядка N операций.

Умножение матрицы на вектор:

Если матрица A размерности $M \times N$, то $(Ax)_j = \sum_{i=1}^{N} A_{ji} x_i$, для $j = 1 \dots M$, то есть нужно совершить порядка $M \cdot N$ операций.

Умножение транспонированной матрицы на вектор:

Аналогично получаем, что нужно совершить порядка $M \cdot N$ операций.

План реализации алгоритма итерационного метода решения СЛАУ

На этой лекции мы подробно разберём операцию умножения матрицы на вектор. На следующей лекции мы разберём скалярное произведение векторов и умножение транспонированной матрицы на вектор, а через лекцию мы соберём все эти подзадачи в единый алгоритм.

Последовательная реализация умножения матрицы на вектор

Последовательная версия программы будет иметь следующий вид:

```
from numpy import empty

f1 = open('in.dat', 'r')
N = int(f1.readline())
M = int(f1.readline())
f1.close()

A = empty((M, N))
x = empty(N)
b = empty(M)

f2 = open('AData.dat', 'r')
for j in range(M):
    for i in range(N):
        A[j, i] = float(f2.readline())
f2.close()

f3 = open('xData.dat', 'r')
for i in range(N):
    x[i] = float(f3.readline())
f3.close()

for j in range(M):
    b[j] = 0
    for i in range(N):
        b[j] += A[j, i]*x[i]

f4 = open('Results.dat', 'w')
for j in range(M):
    print(b[j], file=f4)
f4.close()
```

В файле `in.dat` хранятся два числа – N и M – на разных строках. В файлах `AData.dat`

и `xData.dat` хранятся построчно значения A и x соответсвенно. Естественно, перемножение матрицы на вектор можно было бы сразу выполнить с помощью функции `dot` из пакета `numpy`. Мы так не делаем, чтобы было проще распараллеливать.

Параллельный алгоритм умножения матрицы на вектор

Пусть на каждом процессе хранится свой блок матрицы A (на рис. 1.3 изображён случай с 4 процессами).

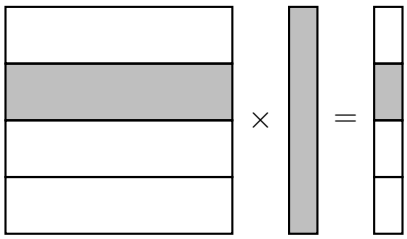


Рис. 1.3: Схема умножения матрицы на вектор

Тогда на каждом процессе будет происходить перемножение одного блока матрицы A на весь вектор x и результатом будет получаться часть вектора b. Эти действия можно делать на различных процессах независимо друг от друга, а в конце нужно будет собрать все части вектора b на одном процессе.

Модели параллельного программирования

Мы будем использовать модель программирования *SPMD (Single Program, Multiple Data)*. То есть в нашем случае все процессы выполняют одну и ту же программу, но с разными данными. Для определения того, какой процесс какую часть программы будет выполнять, у процессов есть определённые идентификаторы.

Программы мы будем организовывать в стиле модели *Master-Worker*. Один из процессов (Master) будет получать входные данные задачи и распределять их по другим процессам (Worker) для обработки. Затем основной процесс получает результаты и проводит финальную обработку.

Основы MPI – простейшая тестовая программа

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numprocs = comm.Get_size()

print(f'Hello from process {rank} of {numprocs}')
```

Мы определяем коммуникатор `comm` – группу процессов, которые участвуют в вычислениях. `MPI.COMM_WORLD` – все процессы, на которых запущена программа. `numprocs` – количество процессов, участвующих в вычислениях, а `rank` – номер (ранг) данного процесса.

В C/Fortran также есть команды `MPI.Init()` и `MPI.Finalize()`, которые должны быть вызваны в начале и в конце параллельных участков программы. В Python `MPI.Init()` вызывается автоматически при импорте `MPI` из `mpi4py`.

Если сохранить этот скрипт под названием `script.py` и запустить в терминале необходимой командой на 5 процессах, то вывод будет следующим:

```
> mpiexec -n 5 python script.py

Hello from process 3 of 5
Hello from process 0 of 5
Hello from process 1 of 5
Hello from process 4 of 5
Hello from process 2 of 5
```

Обратим внимание на то, что у каждого процесса своё адресное пространство, своя память, а значит, значения переменных на разных процессах не равны друг другу. Кроме того, нет никакой упорядоченности, каждый процесс выводит на экран текст по мере выполнения. Если запустить программу ещё раз, вывод программы может быть другим. С этим связано то, что при параллельном вычислении некоторых операций результат может получиться разный из-за ошибок машинного округления (например, при вычислении суммы слагаемых параллельно они могут складываться в разном порядке).

Операции типа точка-точка: Send и Recv

```python
from mpi4py import MPI
from numpy import empty, array, int32, float64, dot

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

```
6  numprocs = comm.Get_size()
7
8  if rank == 0:
9      f1 = open('in.dat', 'r')
10     N = array(int32(f1.readline()))
11     M = array(int32(f1.readline()))
12     f1.close()
13
14 else:
15     N = array(0, dtype=int32)
16     M = array(0, dtype=int32)
```

Если запустить программу сейчас, то на всех процессах, кроме нулевого, N и M будут равняться нулю. Чтобы исправить это, необходимо послать массивы N и M с нулевого процесса остальным. В начале перешлём N на процесс 1:

```
17 if rank == 0:
18     comm.Send([N, 1, MPI.INT], dest=1, tag=0)
19 else:
20     comm.Recv([N, 1, MPI.INT], source=0, tag=0, status=None)
21
22 print(f'Variable N on process {rank} is: {N}')
```

В пакете `mpi4py` есть также функции `send` и `recv` (с маленькой буквы), их отличие в том, что они запаковывают аргументы при передаче и распаковывают при получении. Таким образом можно передавать любые питоновские объекты, а не только массивы `numpy`, но сами объекты занимают больше памяти и операции запаковки и распаковки также занимают время. Кроме того, использование этих функций лишает нас возможности легко переписать программу на C/Fortran, так что мы функции с маленькой буквы будем избегать.

В квадратных скобках перечисляются следующие аргументы: массив `numpy` (данная функция обращается к указателю, который хранится в объекте `numpy.ndarray`, именно поэтому мы должны приводить переменную к массиву, просто `int` или `int32` использовать нельзя), количество элементов массива и идентификатор типа MPI (`MPI.INT` для `numpy.int32`). Далее идут `dest` или `source` – ранг процесса, который получит сообщение, или ранг процесса, отправляющего сообщение. `tag` нужен для того, чтобы разные сообщения можно было различать. Обратите внимание, что мы заранее указали как тип исходного массива, так и тип массива, в который записывается сообщение. При разных типах и с несовпадающим типом MPI сообщение правильно не передастся.

В Python можно писать вместо всех аргументов в квадратных скобках только массив, однако, так как в C/Fortran так делать нельзя, мы будем писать все аргументы для наглядности.

Вернёмся к программе. Если её запустить, то все процессы, кроме нулевого и первого, зависнут (будут вечно ждать `comm.Send`, который на нулевом процессе посылает сообщение только первому процессу, но не остальным).

Операция коллективного взаимодействия Bcast

Можно добавить `for` на нулевом процессе. Тогда, если один вызов функции `Send` занимает время t, то для того, чтобы разослать массив на все процессы, нужно затратить $t \cdot (numprocs - 1)$. Можно оптимизировать этот процесс. Пусть на первом шаге нулевой процесс посылает сообщение первому. На втором шаге нулевой процесс посылает сообщение второму процессу, а первый третьему. На третьем шаге сообщения посылают уже 4 процесса. Нетрудно посчитать, что все затраченное время будет равняться $t \log_2(numprocs)$, что значительно быстрее нашего предыдущего результата.

Этот алгоритм реализован в функции `Bcast`. Перепишем эту часть программы:

```
17 comm.Bcast([N, 1, MPI.INT], root=0)
18 comm.Bcast([M, 1, MPI.INT], root=0)
```

Функция `Bcast` является коллективной операцией, а значит она должна вызываться на всех процессах коммуникатора `comm` (в отличие от операций `Send` и `Recv` – операций типа точка-точка, которые вызываются каждая на конкретном процессе). `root` – процесс, с которого рассылается массив.

Если есть возможность использовать коллективные операции, лучше использовать их, они реализованы гораздо быстрее.

Теперь считаем матрицу A из файла (в начале отправляем только на первый процесс):

```
19 if rank == 0:
20     f2 = open('AData.dat', 'r')
21 
22     A_part = empty((M//(numprocs-1), N), dtype=float64)
23     for j in range(M//(numprocs-1)):
24         for i in range(N):
25             A_part[j, i] = float64(f2.readline())
26     comm.Send([A_part, M//(numprocs-1)*N, MPI.DOUBLE], dest=1, tag=0)
```

Здесь мы для простоты реализуем случай, когда M делится на $(numprocs - 1)$ без остатка.

Теперь перепишем для всех процессов:

```
19 if rank == 0:
20     f2 = open('AData.dat', 'r')
21 
22     for k in range(1, numprocs):
23         A_part = empty((M//(numprocs-1), N), dtype=float64)
24         for j in range(M//(numprocs-1)):
25             for i in range(N):
26                 A_part[j, i] = float64(f2.readline())
27         comm.Send([A_part, M//(numprocs-1)*N, MPI.DOUBLE], dest=k, tag=0)
28 
29     f2.close()
30 
31 else:
32     A_part = empty((M//(numprocs-1), N), dtype=float64)
33     comm.Recv([A_part, M//(numprocs-1)*N, MPI.DOUBLE],
34               source=0, tag=0, status=None)
```

Мы могли бы использовать и здесь коллективные операции, но на практике часто бывает, что матрица A целиком просто не помещается на одном процессе.

Теперь разошлём вектор x с помощью уже знакомой нам функции `Bcast`.

```
35 x = empty(N, dtype=float64)
36 
37 if rank == 0:
38     f3 = open('xData.dat', 'r')
39     for i in range(N):
40         x[i] = float64(f3.readline())
41     f3.close()
42 
43 comm.Bcast([x, N, MPI.DOUBLE], root=0)
```

Параллельная реализация умножения матрицы на вектор

Теперь, наконец, можно перейти к параллельной реализации умножения матрицы на вектор:

```
44 b_part = empty(M//(numprocs-1), dtype=float64)
45 
46 if rank != 0:
47     for j in range(M//(numprocs-1)):
48         b_part[j] = 0
49         for in range(N):
50             b_part[j] += A_part[j, i]*x[i]
```

Так как теперь наглядно видно, что произошло, можно вместо более долгой версии использовать функцию `dot` (которую мы заранее импортировали из пакета `numpy`):

```
44 b_part = empty(M//(numprocs-1), dtype=float64)
45
46 if rank != 0:
47     b_part = dot(A_part, x)
```

Для проверки можно вывести значения `b_part` на разных процессах.

Теперь соберём полученный вектор b на нулевом процессе:

```
48 if rank == 0:
49     b = empty(M, dtype=float64)
50
51     for k in range(1, numprocs):
52         comm.Recv([b_part, M//(numprocs-1), MPI.DOUBLE],
53                   source=k, tag=0, status=None)
54
55         for j in range(M//(numprocs-1)):
56             b[(k-1)*M//(numprocs-1) + j] = b_part[j]
57
58 else: # rank != 0
59     comm.Send([b_part, M//(numprocs-1), MPI.DOUBLE], dest=0, tag=0)
60
61 if rank == 0:
62     f4 = open('Results.dat', 'w')
63     for j in range(M):
64         print(b[j], file=f4)
65     f4.close()
66
67     print(b)
```

Код на строках 48-60 можно записать более оптимально. Во-первых, можно сразу принимать присланный массив в созданный массив `b`, записывая полученные части со сдвигом:

```
48 if rank == 0:
49     b = empty(M, dtype=float64)
50
51     for k in range(1, numprocs):
52         comm.Recv([b[(k-1)*M//(numprocs-1):],
53                   M//(numprocs-1), MPI.DOUBLE],
54                   source=k, tag=0, status=None)
55
56 else:
57     comm.Send([b_part, M//(numprocs-1), MPI.DOUBLE], dest=0, tag=0)
```

Теперь обсудим другой способ улучшения этого участка кода.

Оптимизация сбора информации с помощью функции Probe

Заметим, что пока что нулевой процесс принимает сообщения в строгом порядке. Что делать, есть, например, вначале сообщение придёт от третьего процесса, а не от первого? Можно переписать код в следующем виде: заменим `source` на `MPI.ANY_SOURCE` (любой процесс), а данные от того, какой процесс прислал сообщение, будем передавать через переменную `status`.

```
48 status = MPI.Status()
49
50 if rank == 0:
51     b = empty(M, dtype=float64)
52
53     for k in range(1, numprocs):
54         comm.Recv([b_part, M//(numprocs-1), MPI.DOUBLE],
55                   source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=status)
56
57         for j in range(M//(numprocs-1)):
58             b[(status.Get_source()-1)*M//(numprocs-1) + j] = b_part[j]
59
60 else:
61     comm.Send([b_part, M//(numprocs-1), MPI.DOUBLE], dest=0, tag=0)
```

Теперь хотелось бы опять переписать эту часть кода так, чтобы приём сообщения осуществлялся сразу в массив `b`, без копирования. Для этого нужно заранее проверить сообщение и узнать `status` заранее. Будем использовать функцию `Probe`.

```
48 status = MPI.Status()
49
50 if rank == 0:
51     b = empty(M, dtype=float64)
52
53     for k in range(1, numprocs):
54         comm.Probe(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=status)
55         comm.Recv([b[(status.Get_source()-1)*M//(numprocs-1):],
56                    M//(numprocs-1), MPI.DOUBLE],
57                   source=status.Get_source(),
58                   tag=MPI.ANY_TAG, status=None)
59
60 else:
61     comm.Send([b_part, M//(numprocs-1), MPI.DOUBLE], dest=0, tag=0)
```

Как видно из кода, функция `Probe` помогает нам получить информацию о том, какой процесс уже готов послать сообщение. После этого в аргументы функции `Recv` мы уже с уверенностью можем передать в качестве `source` значение `status.Get_source()`.

Обобщение программы на случай несогласованного числа входных данных и числа процессов, использующихся при расчётах

До сих пор мы считали, что M нацело делится на $(numprocs - 1)$. Теперь рассмотрим более общий случай, когда это может и не выполняться. Для этого после того, как мы считали и распределили значения N и M по процессам, создадим специальные массивы `rcounts` и `displs`, в которых будем хранить длины частей массива на каждом процессе и индексы первых элементов.

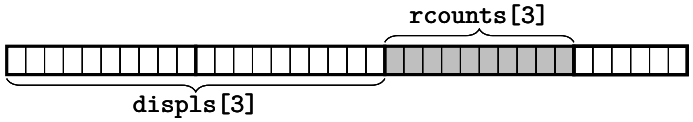


Рис. 1.4: Массивы `rcounts` и `displs`

```
19 if rank == 0:
20     ave, res = divmod(M, numprocs-1)
21     rcounts = empty(numprocs, dtype=int32)
22     displs = empty(numprocs, dtype=int32)
23 
24     rcounts[0] = 0
25     displs[0] = 0
26 
27     for k in range(1, numprocs):
28         if k <= res:
29             rcsounts[k] = ave+1
30         else:
31             rcounts[k] = ave
32 
33         displs[k] = displs[k-1] + rcounts[k-1]
34 
35 else: # rank != 0
36     rcounts = empty(numprocs, dtype=int32)
37     displs = empty(numprocs, dtype=int32)
38 
39 comm.Bcast([rcounts, numprocs, MPI.INT], root=0)
40 comm.Bcast([displs, numprocs, MPI.INT], root=0)
```

Например, если $M = 20$, а *numprocs* $= 4$, то:

```
rcounts = [0, 7, 7, 6]
displs  = [0, 0, 7, 14]
```

В принципе, можно не вызывать функцию `Bcast`, а вычислять массивы `rcounts` и `displs` на каждом процессе.

Продолжаем переписывать программу. Теперь считываем матрицу A:

```
41 if rank == 0:
42     f2 = open('AData.dat', 'r')
43     for k in range(1, numprocs):
44         A_part = empty((rcounts[k], N), dtype=float64)
45         for j in range(rcounts[k]):
46             for i in range(N):
47                 A_part[j, i] = float64(f2.readline())
48 
49         comm.Send([A_part, rcounts[k]*N, MPI.DOUBLE], dest=k, tag=0)
50     f2.close()
51     A_part = empty((rcounts[rank], N), dtype=float64)
52 
53 else: # rank != 0
54     A_part = empty((rcounts[rank], N), dtype=float64)
55     comm.Recv([A_part, rcounts[rank]*N, MPI.DOUBLE],
56               source=0, status=None)
57 
58 ...
59 
60 b_part = dot(A_part)
```

В результате строки 51 на нулевом процессе размер `A_part` будет равен нулю (при желании можно избежать этого действия, изменив программу). Считывание вектора x из файла никак не поменяется, и остаётся только переписать сбор вектора b на нулевом процессе.

Операция коллективного взаимодействия Gatherv

Пока что запишем сбор вектора b в следующем виде:

```
61 status = MPI.Status()
62 
63 if rank == 0:
64     b = empty(M, dtype=float64)
65     for k in range(1, numprocs):
```

```
66          comm.Probe(source=MPI.ANY_TAG, status=status)
67          comm.Recv([b[displs[status.Get_source()]:],
68                     rcounts[status.Get_source()], MPI.DOUBLE],
69                    source=status.Get_source(),
70                    tag=MPI.ANY_TAG, status=None)
71
72  else:
73      comm.Send([b_part. rcounts[rank], MPI.DOUBLE], dest=0, tag=0)
```

Однако, как мы уже знаем, в некоторых случаях некоторые операции можно заменить коллективными, и программа будет работать значительно быстрее. Так можно поступить и тут:

```
61  if rank == 0:
62      b = empty(M, dtype=float64)
63  else:
64      b = None
65
66  comm.Gatherv([b_part, rcounts[rank], MPI.DOUBLE],
67               [b, rcounts, displs, MPI.DOUBLE], root=0)
```

Обратите внимание, что функция `Gatherv` принимает именно массивы `rcounts` и `displs`. В данном примере наличие `displs` может показаться излишним, ведь мы записываем части массива последовательно. Но в некоторых приложениях это может быть не так. Мы увидим такой пример позже в этом курсе.

Также может показаться необычным, что вместо массива `b` на ненулевых процессах передаётся `None`. На самом деле, вся вторая квадратная скобка нужна только для записи на процессе `root=0`, а значит, на других процессах внутри функции `Gatherv` она может быть не определена (Python не дает использовать неопределённые переменные, поэтому задаём `None`).

Операция коллективного взаимодействия Scatterv

Приведём финальную версию программы (с небольшими изменениями):

```
1  from mpi4py import MPI
2  from numpy import empty, array, arange, int32, float64, ones, dot
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  numprocs = comm.Get_size()
7
8  if rank == 0:
```

```
    f1 = open('in.dat', 'r')
    N = array(int32(f1.readline()))
    M = array(int32(f1.readline()))
    f1.close()
else:
    N = array(0, dtype=int32)

comm.Bcast([N, 1, MPI.INT], root=0)

if rank == 0:
    ave, res = divmod(M, numprocs-1)
    rcounts = empty(numprocs, dtype=int32)
    displs = empty(numprocs, dtype=int32)

    rcounts[0] = 0
    displs[0] = 0

    for k in range(1, numprocs):
        if k <= res:
            rcounts[k] = ave + 1
        else:
            rcounts[k] = ave

        displs[k] = displs[k-1] + rcounts[k-1]

else: # rank != 0
    rcounts = None
    displs = None

M_part = array(0, dtype=int32)

comm.Scatterv([rcounts, ones(numprocs), arange(numprocs), MPI.INT],
              [M_part, 1, MPI.INT], root=0)

# comm.Scatter([rcounts, 1, MPI.INT], [M_part, 1, MPI.INT], root=0)
```

Видно, что мы избавились от `M` и массивов `rcounts` и `displs` на ненулевых процессах, но из-за этого нам потребовалась новая переменная `M_part = rcounts[rank]`. Для её инициализации используем новую коллективную операцию `Scatterv`, которая работает противоположно `Gatherv`, то есть рассылает массив по частям на разные процессы с `root`.

В случае, если распределяемые части массива оказываются одинаковой длины, вместо `Gatherv` и `Scatterv` существуют функции `Gather` и `Scatter`. Вариант с этой функцией закомментирован.

Дальше приведём пример, когда матрица A целиком определена на нулевом процессе и рассылается с помощью функции Scatterv:

```
A_part = empty((M_part, N), dtype=float64)

if rank == 0:
    f2 = open('AData.dat', 'r')
    A = empty((M, N), dtype=float64)
    for j in range(M):
        for i in range(N):
            A[j, i] = float64(f2.readline())
    f2.close()
    comm.Scatterv([A, rcounts*N, displs*N, MPI.DOUBLE],
                  [A_part, M_part*N, MPI.DOUBLE], root=0)

else: # rank != 0
    comm.Scatterv([None, None, None, None],
                  [A_part, M_part*N, MPI.DOUBLE], root=0)

x = empty(N, dtype=float64)
if rank == 0:
    f3 = open('xData.dat', 'r')
    for i in range(N):
        x[i] = float64(f3.readline())
    f3.close()

comm.Bcast([x, N, MPI.DOUBLE], root=0)

b_part = dot(A_part, x)

if rank == 0:
    b = empty(M, dtype=float64)
else:
    b = None

comm.Gatherv([b_part, rcounts[rank], MPI.DOUBLE],
             [b, rcounts, displs, MPI.DOUBLE], root=0)

if rank == 0:
    f4 = open('Results.dat', 'w')
    for j in range(M):
        print(b[j], file=f4)
    f4.close()

    print(b)
```

Программу можно найти в материалах к лекциям под номером 1-1 (первое число –

номер лекции, второе число – номер программы на лекции).

Функции передачи сообщений Send, Bsend, Ssend и Rsend

Сделаем одно замечание по поводу функций `Send`/`Recv`. Функция `Recv` выставляется и ждёт приёма сообщения, блокируя процесс. Функция `Send` реализована более хитро. Если внутреннего MPI-буфера достаточно для пересылки сообщения, то сообщение копируется в этот буфер, а процесс продолжает работу, выходя из функции `Send`. Отправка сообщения происходит в фоне. Если внутреннего буфера недостаточно, процесс ждёт, пока на другом процессе не будет вызвана соответствующая функция `Recv`.

У функции `Send` есть несколько вариаций – `Bsend`, `Ssend`, `Rsend`. `Bsend` – посылка сообщений с буферизацией, то есть предварительно выделяется место в памяти (не буфер MPI), куда будет сразу скопирован массив. Тогда процесс будет сразу продолжать работу, не дожидаясь вызова `Recv`. Из минусов: необходимо выделять память дополнительно, т.е. памяти может не хватить; также `Recv` уже может быть вызван, но сообщение все равно будет копироваться. `Ssend` – посылка с синхронизацией, выход из функции осуществляется после фактической отправки сообщения. `Rsend` – посылка сообщения по готовности, сообщение сразу же уходит соответствующему процессу, независимо от того, был ли уже на нём вызван `Recv`. Если `Recv` ещё не был вызван, будет сбой.

Лекция 2. Введение в основы MPI на Python.

На этой лекции мы рассмотрим два вопроса: параллельную реализацию скалярного произведения векторов и умножения транспонированной матрицы на вектор.

Последовательная реализация вычисления скалярного произведения векторов

Для простоты будем искать скалярное произведение вектора самого на себя. Последовательная программа будет выглядеть весьма компактно:

```
1  from numpy import arange
2
3  M = 20
4  a = arange(1, M+1)
5
6  ScalP = 0
7  for j in range(M):
8      ScalP += a[j]*a[j]
9
10 print('ScalP =', ScalP)
```

Можно сделать эту операцию ещё проще, через функцию `dot`, но наша реализация позволит нам проще распараллелить алгоритм. Видно, что для каждого `j` слагаемое `a[j]*a[j]` не зависит от остальных.

Для начала отвлечёмся и обсудим вот какой вопрос. Что будет, если эту последовательную программу запустить параллельно? Попробуем это сделать:

```
> mpiexec -n 4 python script.py

ScalP = 2870.0
ScalP = 2870.0
ScalP = 2870.0
ScalP = 2870.0
```

Как видно, одна и та же программа просто запускается на разных процессорах и выполняет ровно одни и те же действия. Это неудивительно, так как пакет MPI мы не импортировали.

Параллельный алгоритм вычисления скалярного произведения векторов

Пусть на разных узлах хранятся разные части векторов. Схему алгоритма вычисления скалярного произведения можно увидеть на рис. 2.1: мы вычислим скалярное произведение частей на каждом процессоре, а затем сложим полученные части на управляющем процессоре.

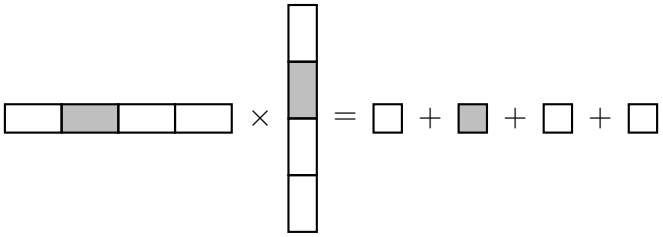


Рис. 2.1: Схема подсчёта скалярного произведения векторов

Видно, что процессы должны обмениваться сообщениями только на этапе финального сложения.

Параллельная реализация вычисления скалярного произведения векторов

Начнём программу так же, как и на первой лекции.

```
1  from mpi4py import MPI
2  from numpy import arange, empty, array, int32, float64, dot
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  numprocs = comm.Get_size()
```

Создаём массив на нулевом процессе:

```
7  if rank == 0:
8      M = 20
9      a = arange(1, M+1, dtype=float64)
10 else:
11     a = None
```

Создаём массивы `rcounts` и `displs` (как на первой лекции):

```
12 if rank == 0:
13     ave, res = divmod(M, numprocs-1)
```

```
14      rcounts = empty(numprocs, dtype=int32)
15      displs = empty(numprocs, dtype=int32)
16
17      rcounts[0] = 0
18      displs[0] = 0
19
20      for k in range(1, numprocs):
21          if k <= res:
22              rcounts[k] = ave + 1
23          else:
24              rcounts[k] = ave
25
26          displs[k] = displs[k-1] + rcounts[k-1]
27
28  else: # rank != 0
29      rcounts = None
30      displs = None
```

Напоминаем, что массив `rcounts` содержит информацию о том, сколько элементов исходного массива хранится на данном процессе, а в массиве `displs` хранятся смещения частей массива относительно исходного (можно легко вспомнить, о чём идет речь, посмотрев на рис. 1.4). Далее создаём переменную `M_part` и расссылаем её значение. Напоминаем, что, по сути, `M_part = rcounts[rank]`.

```
31  M_part = array(0, dtype=int32)
32
33  comm.Scatter([rcounts, 1, MPI.INT], [M_part, 1, MPI.INT], root=0)
```

В данном случае (на строке 31) в Python создаётся массив размерности 0, который воспринимается как число. Это действие мы делаем потому, что обычные числовые типы в Python являются неизменяемыми объектами, поэтому только массивы `numpy.ndarray` могут быть переданы в функции MPI. Например, в C можно передавать указатели на тип `int` напрямую.

Подготовим место в памяти под часть массива `a`, а затем помощью функции `Scatterv` разошлём массив `a` по частям на разные процессы:

```
34  a_part = empty(M_part, dtype=float64)
35
36  comm.Scatterv([a, rcounts, displs, MPI.DOUBLE],
37                [a_part, M_part, MPI.DOUBLE], root=0)
```

Напоминаем, что отличие функций `Scatter` и `Scatterv` в том, что в одном случае массив делится на равные части, а в другом случае нет. Именно из-за этого возникает необходимость создавать массив `rcounts`. Другим отличием является наличие массива `displs`.

В нашем случае сейчас он большой роли не играет, однако он понадобится нам ближе в концу курса, когда мы будем решать двухмерные по пространству уравнения в частных производных.

Задаём временную переменную `ScalP_temp` для хранения полученных слагаемых при вычислении части скалярного произведения на каждом процессе. Проводим соответсвующие вычисления на каждом процессе.

```
38 ScalP_temp = empty(1, dtype=float64)
39 ScalP_temp[0] = dot(a_part, a_part)
```

Вначале реализуем сбор `ScalP_temp` с разных процессов через `Send/Recv`.

```
40 ScalP = 0
41
42 if rank == 0:
43     for k in range(1, numprocs):
44         comm.Recv([ScalP_temp, 1, MPI.DOUBLE],
45                   source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=None)
46         ScalP += ScalP_temp[0]
47 else:
48     comm.Send([ScalP_temp, 1, MPI.DOUBLE], dest=0, tag=0)
```

Обратите внимание, что в данной ситуации порядок сложения слагаемых не важен, а значит, можно не отслеживать `source` с помощью `status`. Однако, из-за машинного округления от запуска к запуску может немного меняться результат.

Операции коллективного взаимодействия Reduce и Allreduce

Можно догадаться, что такой подсчёт суммы будет не самым оптимальным. Можно, например, заменить его на алгоритм сдваивания. На первом этапе будет вычисляться параллельно сумма `ScalP_temp` на первом и втором процессе, одновременно с этим на третьем и четвёртом, и так далее. На втором и последующих этапах полученные промежуточные результаты опять будут складываться между собой по два. Легко посчитать, что время этого алгоритма будет пропорционально $\log_2(numprocs\text{-}1)$.

Не будем изобретать велосипед и воспользуемся встроенной функцией `Reduce`.

```
40 ScalP = array(0, dtype=float64)
41
42 comm.Reduce([ScalP_temp, 1, MPI.DOUBLE],
43             [ScalP, 1, MPI.DOUBLE], op=MPI.SUM, root=0)
44
45 print(f'ScalP = {ScalP:6.1f} on process {rank}')
```

Функция `Reduce` берёт элементы `ScalP_temp` с каждого процесса коммуникатора `comm` и проводит над ними операцию `op=MPI.SUM`. Результат записывается в переменную `ScalP` на процессе `root` (на остальных процессах она не нужна и может быть заменена на `None`). Так как эта функция относится к функциям коллективного взаимодействия, напоминаем, что она должна быть запущена на всех процессах коммуникатора `comm` (в том числе и на нулевом, из-за чего на нулевом процессе мы задаём `ScalP` и вычисляем его значение = 0).

Операции `op` могут быть и другими. Например, в каком-то другом случае можно вычислить максимум элементов. Подробнее о других операциях можно почитать в документации.

Запустим программу и посмотрим, что она выведет.

```
> mpiexec -n 4 python script.py

ScalP =    0.0 on process 1
ScalP =    0.0 on process 3
ScalP =    0.0 on process 2
ScalP = 2870.0 on process 0
```

Как и сказано в описании функции `Reduce`, правильно значение записывается в переменную только на процессоре `root`, в нашем случае, нулевом. Кстати, напоминаем, что программа может работать не только на четырёх процессорах, но и на любом другом числе (большем, чем один).

Если нам потребуется затем разослать значение переменной `ScalP` на все процессы, можно сразу же вызвать функцию `Bcast`. Лучше, однако, будет вместо функции `Reduce` вызвать `Allreduce`, которая, как можно догадаться из названия, рассылает результат на все процессы коммуникатора `comm`:

```
40 ScalP = array(0, dtype=float64)
41
42 comm.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
43                [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
44
45 print(f'ScalP = {ScalP:6.1f} on process {rank}')
```

Очевидно, почему пропал аргумент `root`.

Теперь, если запустить программу, `ScalP` будет определён на всех процессах:

```
> mpiexec -n 4 python script.py

ScalP = 2870.0 on process 1
```

```
ScalP = 2870.0 on process 2
ScalP = 2870.0 on process 3
ScalP = 2870.0 on process 0
```

Для себя можно воспринимать операцию `Allreduce` как последовательное применение `Reduce` и `Bcast`. Внутри она, однако, реализована более оптимально.

Программу можно найти в материалах к лекциям под номером 2-1.

Параллельный алгоритм умножения транспонированной матрицы на вектор

Для начала нужно отметить, что с математической точки зрения умножение обычной матрицы на вектор и умножение транспонированной матрицы на вектор – одна и та же операция. Нас, однако, эта операция интересует в контексте метода сопряжённых градиентов (1.1). Так как матрица A уже хранится в памяти определённым образом, нам бы хотелось подстроить операцию умножения A^T на вектор так, чтобы избежать копирования.

Так как массив `A` уже разбит на блоки, разобъём вектор на блоки, согласованно с разбиением матрицы. Дальше на каждом процессе будет происходить умножение частей матриц на части вектора. Полученные вектора с разных процессов нужно будет сложить. Схему этой операции можно увидеть на рис. 2.2.

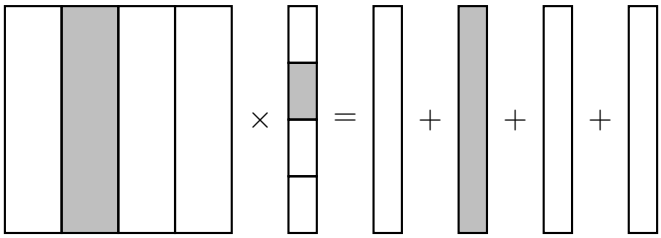


Рис. 2.2: Схема умножения транспонированной матрицы на вектор

Параллельная реализация умножения транспонированной матрицы на вектор

Программа будет начинаться также, как и в случае с нетранспонированной матрицей. Из файла считываем N и M – размерность матрицы A. Далее подготавливаем вспомога-

тельные массивы `rcounts` и `displs`. На каждом процессе задаём переменную `M_part` и присваиваем ей значение, присылая его с нулевого процесса. Это позволяет нам считать значение массива `A` из файла и разослать его по частям всем процессам.

```
from mpi4py import MPI
from numpy import empty, array, int32, float64, dot

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numprocs = comm.Get_size()

if rank == 0:
    f1 = open('in.dat', 'r')
    N = array(int32(f1.readline()))
    M = array(int32(f1.readline()))
    f1.close()
else:
    N = array(0, dtype=int32)

comm.Bcast([N, 1, MPI.INT], root=0)

if rank == 0:
    ave, res = divmod(M, numprocs-1)
    rcounts = empty(numprocs, dtype=int32)
    displs = empty(numprocs, dtype=int32)

    rcounts[0] = 0
    displs[0] = 0

    for k in range(1, numprocs):
        if k <= res:
            rcounts[k] = ave + 1
        else:
            rcounts[k] = ave

        displs[k] = displs[k-1] + rcounts[k-1]

else: # rank != 0
    rcounts = None
    displs = None

M_part = array(0, dtype=int32)
comm.Scatter([rcounts, 1, MPI.INT], [M_part, 1, MPI.INT], root=0)

if rank == 0:
    f2 = open('AData.dat', 'r')
    for k in range(1, numprocs):
        A_part = empty((rcounts[k], N), dtype=float64)
```

```
45          for j in range(rcounts[k]):
46              for i in range(N):
47                  A_part[j, i] = float64(f2.readline())
48
49          comm.Send([A_part, rcounts[k]*N, MPI.DOUBLE], dest=k, tag=0)
50      f2.close()
51      A_part = empty((rcounts[rank], N), dtype=float64)
52
53  else: # rank != 0
54      A_part = empty((rcounts[rank], N), dtype=float64)
55      comm.Recv([A_part, rcounts[rank]*N, MPI.DOUBLE],
56                source=0, status=None)
```

В обработке массива x, однако, будут некоторые изменения. Во-первых, длина вектора x в данной программе – M, а не N. Во-вторых, напомним, что в алгоритме нам нужно, чтобы массив x хранился на разных процессах по частям.

```
57  if rank == 0:
58      x = empty(M, dtype=float64)
59      f3 = open('xData.dat', 'r')
60      for j in range(M):
61          x[j] = float64(f3.readline())
62      f3.close()
63  else:
64      x = None
65
66  x_part = empty(M_part, dtype=float64)
67  comm.Scatterv([x, rcounts, displs, MPI.DOUBLE],
68                [x_part, M_part, MPI.DOUBLE], root=0)
```

Параллельная часть программы занимает всего одну строчку:

```
69  b_temp = dot(A_part.T, x_part)
```

Далее выделяем на нулевом процессе массив b

```
70  if rank == 0:
71      b = empty(N, dtype=float64)
72  else:
73      b = None
```

и воспользуемся функцией коллективного взаимодействия Reduce:

```
74  comm.Reduce([b_temp, N, MPI.DOUBLE],
75              [b, N, MPI.DOUBLE], op=MPI.SUM, root=0)
```

Остаётся только вывести результат на экран.

```
76 if rank == 0:
77     print(b)
```

Мы рассмотрели несколько программ – параллельное умножение матрицы на вектор, параллельное вычисление скалярного произведения и параллельное умножение транспонированной матрицы на вектор. На следующей лекции мы объединим эти программы в одну, написав программную реализацию метода сопряжённых градиентов.

Лекция 3. Параллельная реализация метода сопряжённых градиентов для решения СЛАУ.

На предыдущих лекциях мы провели всю подготовительную работу, которая поможет нам распараллелить итерационный метод решения СЛАУ $Ax = b$ (см. 1.1). Напомним, что A – прямоугольная матрица достаточно большого размера (из-за чего мы и хотим распараллелить алгоритм), число строк больше числа столбцов. В общем случае вектор b содержит какие-то ошибки. В случае, если матрица A – квадратная (т.е., размерности $N \times N$), а вычисления проводятся абсолютно точно, алгоритм даёт точный результат на шаге $s = N$. Если система переопределённая, алгоритм находит минимум функционала $||Ax - b||^2$.

Напомним, какие в алгоритме есть наиболее трудоёмкие операции. Во-первых, это скалярное произведение ($\sim N$ операций). Во-вторых, деление вектора на число и сумма векторов ($\sim N$ операций). В-третьих, умножение матрицы на вектор ($\sim N \cdot M$ операций). В-четвёртых, умножение транспонированной матрицы на вектор (также $\sim N \cdot M$ операций). Все эти операции мы разобрали на предыдущих двух занятиях, написав отдельные программы.

Последовательная реализация метода сопряжённых градиентов

Вначале импортируем необходимые библиотеки и заполним матрицы значениями из файла. Формат хранения уже оговаривался в предыдущих программах. В файле `in.dat` на разных строках хранятся значения N и M, в файлах `AData.dat` и `xData.dat` построчно хранятся значения элементов матрицы A и вектора x соответственно.

```
1  from numpy import zeros, empty, dot, arange, linalg, eye
2  import matplotlib.pyplot as plt
3
4  f1 = open('in.dat', 'r')
5  N = int(f1.readline())
6  M = int(f1.readline())
7  f1.close()
8
9  A = empty((M, N))
10 b = empty(M)
11
12 f2 = open('AData.dat', 'r')
```

```
for j in range(M):
    for i in range(N):
        A[j, i] = float(f2.readline())
f2.close()

f3 = open('bData.dat', 'r')
for j in range(M):
    b[j] = float(f3.readline())
f3.close()
```

Отдельно задаём функцию, которая реализует алгоритм метода сопряжённых градиентов (1.1):

```
def conjugate_gradient_method(A, b, x, N):
    s = 1
    p = zeros(N)

    while s <= N:
        if s == 1:
            r = dot(A.T, dot(A, x) - b)
        else:
            r -= q / dot(p, q)

        p += r / dot(r, r)
        q = dot(A.T, dot(A, p))
        x -= p / dot(p, q)
        s += 1

    return x
```

Далее задаём приближенное значение $x = 0$ и вызываем функцию, которая выполняет подсчёт решения методом сопряжённых градиентов.

```
x = zeros(N)
x = conjugate_gradient_method(A, b, x, N)
```

Результат выводится на экран с помощью пакета `matplotlib`:

```
plt.plot(arange(N), x, '-y', lw=3)
plt.ylim=(-1.5, 1.5)
plt.xlabel('i')
plt.ylabel('x[i]')
plt.show()
```

Параллельная реализация метода сопряжённых градиентов

Полная версия программы, как обычно, выложена на портале teach-in в разделе "Материалы к лекциям откуда её можно скачать вместе с тестовыми входными файлами.

Начинаем программу стандартным образом: импортируем необходимые библиотеки, определяем количество доступных процессов и номер каждого процесса.

```
1  from mpi4py import MPI
2  from numpy import empty, array, int32, float64, zeros, arange, dot
3  import matplotlib.pyplot as plt
4
5  comm = MPI.COMM_WORLD
6  rank = comm.Get_rank()
7  numprocs = comm.Get_size()
```

Сразу задаём функцию, которая будет реализовывать алгоритм метода сопряжённых градиентов. Однако, не будем торопиться писать тело этой функции, оставим её на потом (из-за этого перескакиваем сразу на строку 52).

```
8  def conjugate_gradient_method(A_part, b_part, x_part,
9                                N, N_part, rcounts_N, displs_N):
10     ...
```

Дальше займёмся подготовкой входных данных. Подробно не останавливаемся на деталях, так как подобный код уже не раз встречался ранее. Вначале считываем значения N и M из файла, но на все процессы рассылаем только N.

```
52 if rank == 0:
53     f1 = open('in.dat', 'r')
54     N = array(int32(f1.readline()))
55     M = array(int32(f1.readline()))
56     f1.close()
57 else:
58     N = array(0, dtype=int32)
59
60 comm.Bcast([N, 1, MPI.INT], root=0)
```

Далее определяем то, как будем разбивать входные массивы на части. Выделим эту часть кода в отдельную функцию, так как в процессе написания программы может потребоваться разбивать как массивы длины N, так и массивы длины M.

```
61 def auxiliary_arrays_determination(M, numprocs):
62     ave, res = divmod(M, numprocs-1)
63     rcounts = empty(numprocs, dtype=int32)
64     displs = empty(numprocs, dtype=int32)
```

```
65
66      rcounts[0] = 0
67      displs[0] = 0
68
69      for k in range(1, numprocs):
70          if k <= res:
71              rcounts[k] = ave + 1
72          else:
73              rcounts = ave
74
75          displs[k] = displs[k-1] + rcounts[k-1]
76
77      return rcounts, displs
```

Напоминаем, что вспомогательные массивы `rcounts` и `displs` отвечают за длины частей исходного массива после разбиения и сдвиги относительно нулевого элемента (см. рис. 1.4).

Сразу подсчитаем соответсвующие массивы для N и M:

```
78 if rank == 0:
79      rcounts_M, displs_M = auxiliary_arrays_determination(M, numprocs)
80      rcounts_N, displs_N = auxiliary_arrays_determination(N, numprocs)
81 else:
82      rcounts_M, displs_M = None, None
83      rcounts_N = empty(numprocs, dtype=int32)
84      displs_N = empty(numprocs, dtype=int32)
85
86 comm.Bcast([rcounts_N, numprocs, MPI.INT], root=0)
87 comm.Bcast([displs_N, numprocs, MPI.INT], root=0)
```

На ненулевых процессах, как будет видно дальше, понадобятся вспомогательные массивы только для N. Для M будет достаточно только локальной длины, так что ведём переменные `M_part`:

```
88 M_part = array(0, dtype=int32)
89 comm.Scatter([rcounts_M, 1, MPI.INT], [M_part, 1, MPI.INT], root=0)
```

Считываем значения элементов матрицы A и рассылаем части по процессам:

```
90 if rank == 0:
91      f2 = open('AData.dat', 'r')
92      for k in range(1, numprocs):
93          A_part = empty((rcounts_M[k], N), dtype=float64)
94          for j in range(rcounts_M[k]):
95              for i in range(N):
96                  A_part[j, i] = float64(f2.readline())
```

```
97              comm.Send([A_part, rcounts_M[k]*N, MPI.DOUBLE], dest=k, tag=0)
98          f2.close()
99          A_part = empty((M_part, N), dtype=float64)
100
101     else: # rank != 0
102         A_part = empty((M_part, N), dtype=float64)
103         comm.Recv([A_part, M_part*N, MPI.DOUBLE],
104                   source=0, tag=0, status=None)
```

Ещё раз обратите внимание, что мы пишем программу в стиле SPMD, т.е. Single Program, Multiple Data. Хотя код программы один и тот же, на разных процессах переменные `A_part` не только ссылаются на различные массивы, но и, к тому же, это массивы различных размеров.

Аналогично распределяем значения элементов вектора b по процессам. Отличие только в том, что размер массива `b` значительно меньше, чем массива `A`, а значит, мы можем позволить себе целиком хранить его на нулевом процессе и рассылать с помощью операции `Scatterv`.

```
105 if rank == 0:
106     b = empty(M, dtype=float64)
107     f3 = open('bData.dat', 'r')
108     for j in range(M):
109         b[j] = float64(f3.readline())
110     f3.close()
111 else:
112     b = None
113
114 b_part = empty(M_part, dtype=float64)
115 comm.Scatterv([b, rcounts_M, MPI.DOUBLE],
116               [b_part, M_part, MPI.DOUBLE], root=0)
```

Напомним отличие функций `Scatter` и `Scatterv`. Вторая функция распределяет массив по частям с различной длиной (за это отвечает аргумент `rcounts`), первая – по частям с одинаковой длиной.

Подготовим нулевое приближение x. Задаём значение на нулевом процессе, а дальше раскидываем его по частям на все процессы:

```
117 if rank == 0:
118     x = zeros(N, dtype=float64)
119 else:
120     x = None
121
122 x_part = empty(rcounts_N[rank], dtype=float64)
123 comm.Scatterv([x, rcounts_N, displs_N, MPI.DOUBLE],
```

```
124                 [x_part, rcounts_N[rank], MPI.DOUBLE], root=0)
```

В данном случае можно было бы сразу задать `x_part` как нулевые массивы и исбежать пересылки сообщений. Однако, если бы мы знали другое хорошее начальное приближение, так сделать бы не вышло. Также заметим, что, по-хорошему, после распределения массива по процессам использованную под исходный массив память можно было бы освобождать.

Вызываем функцию метода сопряжённых градиентов. Каждый процесс получает свою часть матрицы A, свою часть вектора b, свою часть начального приближения и вспомогательные переменные.

```
125 x_part = conjugate_gardient_method(A_part, b_part, x_part, N,
126                                    rcounts_N[rank], rcounts_N, displs_N)
```

Остаётся только собрать результат по частям с разных процессов и построить график.

```
127 comm.Gatherv([x_part, rcounts_N[rank], MPI.DOUBLE],
128              [x, rcounts_N, displs_N, MPI.DOUBLE], root=0)
129
130 if rank == 0:
131     plt.plot(arange(N), x, '-y', lw=3)
132     plt.ylim=(-1.5, 1.5)
133     plt.xlabel('i')
134     plt.ylabel('x[i]')
135     plt.show()
```

График можно построить только если вы запускаете программу на своём компьютере. Если вы запускаете программу удалённо на кластере, лучше будет записать результат в файл, а уже потом скопировать его к себе на компьютер и построить график с помощью другой вспомогательной программы.

Операции коллективного взаимодействия Allgatherv и Reduce_Scatter

На текущий момент мы только лишь занимались подготовкой данных. Теперь переходим к разбору функции `conjugate_gradient_method`. Начнём с выделения памяти под все используемые массивы.

```
8  def conjugate_gradient_method(A_part, b_part, x_part,
9                                N, N_part, rcounts_N, displs_N):
10
11     x = empty(N, dtype=float64)
12     p = empty(N, dtype=float64)
```

```
13
14      r_part = empty(N_part, dtype=float64)
15      q_part = empty(N_part, dtype=float64)
16
17      ScalP = array(0, dtype=float64)
18      ScalP_temp = empty(1, dtype=float64)
19
20      s = 1
21      p_part = zeros(N_part, dtype=float64)
```

Массивы x и p мы выделяем на всех процессах, так как они нужны на всех процессах при операции умножения матрицы на вектор. От остальных массивов на каждом конкретном процессе нам будут нужны только фрагменты.

Переходим к написанию цикла. Перед фрагментами кода будем приводить соответсвующие операции в виде формул.

```
22      while s <= N:
```

$$r^{(s)} = \begin{cases} A^T\left(Ax^{(s)} - b\right), & \text{при} \quad s = 1 \\ r^{(s-1)} - \dfrac{q^{(s-1)}}{\left(p^{(s-1)}, q^{(s-1)}\right)}, & \text{при} \quad s \geqslant 2 \end{cases}$$

При старте вектор x хранится по частям, а значит, его нужно собрать. В дальнейшем все его элементы потребуются на всех процессах, поэтому воспользуемся функцией `Allgatherv`. Мы уже встречались с функцией с похожим названием, начинающемся с `All-`, в прошлый раз это была операция `Allreduce`. Аналогично, `Allgatherv` для себя можно воспринимать как последовательное применение операций `Gatherv` и `Bcast` (результат получается один и тот же, но внутри операция `Allgatherv` реализована более оптимально и работает гораздо быстрее).

```
23          if s == 1:
24              comm.Allgatherv([x_part, N_part, MPI.DOUBLE],
25                              [x, rcounts_N, displs_N, MPI.DOUBLE])
```

То есть вектор собирается по частям, а его значение становится доступным на всех процессах, а не только на выбранном `root`, как в случае с `Gatherv`. Поэтому синтаксис вызова `Allgatherv` аналогичен функции `Gatherv`, только отсутсвует аргумент `root`. Существенным отличием является то, что переменные во вторых квадратных скобках должны быть определены на всех процессах. В случае с `Gatherv` они могли быть заданы как `None` на всех процессах, кроме `root`.

Дальше вычисляем произведение части матрицы на вектор. В отдельной программе мы после этой операции собирали результат на нулевом процессе. Здесь мы так делать не

станем, так как вектор b уже подготовлен и тоже распределён по процессам. Более того, в алгоритме произведения транспонированной матрицы на вектор также подразумевается, что вектор распределён по процессам, поэтому сразу умножаем на часть транспонированной матрицы.

```
26              r_temp = dot(A_part.T, dot(A_part, x) - b_part)
```

Получившийся массив `r_temp` – это не итоговый результат, а лишь одно из слагаемых. Сложив `r_temp` с разных процессов, мы получим итоговый вектор `r` (см. рис. 2.2).

Однако, нам нужен не массив `r`, а его части `r_part`. Для этого можно было бы применить последовательно функции `Reduce` и `Scatterv`, а мы применим их аналог, функцию `Reduce_scatter`.

```
27              comm.Reduce_scatter([r_temp, N, MPI.DOUBLE],
28                                  [r_part, N_part, MPI.DOUBLE],
29                                  recvcounts=rcounts_N, op=MPI.SUM)
```

Получается, что мы применяем операцию `op=MPI.SUM` к массивам `r_temp`, находящимся на разных процессах, а затем результат рассылается по частям обратно на эти процессы.

В случае, если итерация не первая, необходимо вычислить скалярное произведение (p, q). Делаем это также, как на предыдущей лекции:

```
30          else: # s != 0
31              ScalP_temp[0] = dot(p_part, q_part)
32              comm.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
33                             [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
```

Остаётся параллельно выполнить деление вектора на число и вычитание векторов:

```
34              r_part -= q_part / ScalP
```

$$p^{(s)} = p^{(s-1)} + \frac{r^{(s)}}{\left(r^{(s)}, r^{(s)}\right)}$$

Вначале выполним уже знакомую нам операцию вычисления скалярного произведения (r, r):

```
35          ScalP_temp[0] = dot(r_part, r_part)
36          comm.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
37                         [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
```

После этого параллельно на всех процессах вычисляем очередное приближение вектора p:

```
38          p_part += r_part / ScalP
```

$$q^{(s)} = A^T\left(Ap^{(s)}\right)$$

Так как для параллельного алгоритма умножения матрицы на вектор нам нужно, чтобы все компоненты вектора p были доступны на каждом процессе, вызываем

```
39          comm.Allgatherv([p_part, N_part, MPI.DOUBLE],
40                          [p, rcounts_N, displs_N, MPI.DOUBLE])
```

После этого выполняем умножение матрицы на вектор и умножение транспонированной матрицы на полученный вектор. Перед нами – самая вычислительно ёмкая операция цикла (в предыдущий раз умножение вычислялось лишь на первой итерации).

```
41          q_temp = dot(A_part.T, dot(A_part, p))
```

Так как в дальнейшем нам понадобятся части `q_part`, снова применяем операцию `Reduce_scatter`.

```
42          comm.Reduce_scatter([q_temp, N, MPI.DOUBLE],
43                              [q_part, N_part, MPI.DOUBLE],
44                              recvcounts=rcounts_N, op=MPI.SUM)
```

$$x^{(s+1)} = x^{(s)} - \frac{p^{(s)}}{\left(p^{(s)}, q^{(s)}\right)}$$

Выполнение последнего действия уже не вызывает никаких сложностей. Приведём весь код целиком:

```
45          ScalP_temp[0] = dot(p_part, q_part)
46          comm.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
47                         [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
48          x_part -= p_part / ScalP
49
50          s += 1
```

После выполнения тела цикла не забудем возвратить из функции `x_part`.

```
51      return x_part
```

Целиком программу можно найти на портале teach-in под номером 3-1.

Обсуждение преимуществ и недостатков предложенной программной реализации

Если посчитать, какие действия на скольких строках кода выполняются, то можно увидеть, что большая часть программы отводится под подготовку данных. Она происходит, в основном, последовательно. Непосредственно параллельная часть находится только внутри функции `conjugate_gradient_method`, однако и помимо неё внутри функции достаточно кода уходит под пересылку сообщений между процессами.

На следующей лекции мы будем подробно говорить о вопросах, связанных с эффективностью распараллеливания и масштабируемостью параллельных алгоритмов. Однако, и сейчас интуитивно понятно, что чем большую часть вычислений можно распараллелить, тем более эффективную программу удастся написать. Все же нужно помнить, что всегда есть участки программы, которые невозможно распараллелить.

Подготовку данных иногда распараллелить получается. При решении реальных практических задач, если требуется заполнять большую матрицу, зачастую можно избежать ситуации, когда матрица хранится в каком-то файле и должна пройти через нулевой процессор, прежде чем будет разослана на все остальные. Если матрица A соответсвует какому-то физическому оператору, мы, скорее всего, знаем, по какому закону действует этот оператор. Значит, согласно этому известному закону, мы можем сразу заполнять матрицу A по частям параллельно на всех процессах.

При оценке эффективности параллельной программы подготовка данных обычно не учитывается. Поэтому сейчас сосредоточимся на теле цикла `while` внутри функции `conjugate_gradient_method` (строки 23-50). Если пренебречь функциями передачи сообщений, у нас распараллелены все действия. Мы ожидаем ускорения, пропорционального числу процессов (например, если мы запустим на 10 процессах, то считаться будет в 10 раз быстрее последовательной программы). Реально так не получится, потому что функции передачи сообщения могут занимать существенное время. Каким бы эффективным ни был алгоритм передачи сообщений, все равно это занимает время, из-за чего процессы простаивают. Если, к тому же, алгоритм передачи сообщений реализован неэффективно (например, через `Send`/`Recv`, а не через коллективные операции взаимодействия), вместо эффективно-работающей параллельной программы получается ”чатик между процессами то есть процессы значительную часть времени просто передают и получают друг от друга сообщения. Такая ситуация на практике очень распространена, поэтому нужно внимательно подходить к анализу эффективности работы параллельной программы.

При работе параллельной программы, если процессоры работают 80% времени, а простаивают 20%, то на практике это – хороший результат. Если процессоры работают

90% времени, а простаивают всего 10%, то всё вообще замечательно. Наша цель – загрузить каждый процесс как можно большим объёмом вычислений, чтобы доля времени, которое тратится на передачу сообщений, была меньше. Вторая наша цель – минимизировать количество обмена сообщениями между процессами и сделать так, чтобы обмен был как можно меньшим объёмом данных.

Время передачи сообщений от одного процесса другому может зависеть от многих факторов. Например, от топологии системы. Для интересующихся лектор советует прослушать курс Вл.В. Воеводина и его коллег из НИВЦ МГУ ”Суперкомпьютеры и параллельная обработка данных”.

Параллельная реализация метода сопряжённых градиентов: упрощённая версия программы

Чтобы уменьшить количество передаваемых сообщений, попробуем распараллелить не все операции. Оставим параллельными операции умножения матрицы на вектор и умножение транспонированной матрицы на вектор, а скалярное произведение будем считать последовательно. Программа выложена на портале teach-in под номером 3-2.

```
1  from mpi4py import MPI
2  from numpy import empty, array, int32, float64, zeros, arange, dot
3  import matplotlib.pyplot as plt
4
5  comm = MPI.COMM_WORLD
6  rank = comm.Get_rank()
7  size = comm.Get_size()
8
9  def conjugate_gradient_method(A_part, b_part, x, N):
10     ...
```

Как и с предыдущей версией программы, вначале обсудим обработку данных, а тело функции `conjugate_gradient_method` оставим на потом (пропускаем строки 9-32). Из-за внесённых изменений нам не потребуются вспомогательные массивы `rcounts_N` и `displs_N`. Все их упоминания из программы исчезнут.

```
33 if rank == 0:
34     f1 = open('in.dat', 'r')
35     N = array(int32(f1.readline()))
36     M = array(int32(f1.readline()))
37     f1.close()
38 else:
39     N = array(0, dtype=int32)
40
```

```
41 comm.Bcast([N, 1, MPI.INT], root=0)
```

Функцию, генерирующую вспомогательные массивы мы оставляем без изменения. Полное тело функции повторно приводить не будем.

```
42 def auxiliary_arrays_determination(M, numprocs):
43     ...
44     return rcounts, displs
45
46 if rank == 0:
47     rcounts_M, displs_M = auxiliary_arrays_determination(M, numprocs)
48 else:
49     rcounts_M, displs_M = None, None
50
51 M_part = array(0, dtype=int32)
52 comm.Scatter([rcounts_M, 1, MPI.INT], [M_part, 1, MPI.INT], root=0)
```

Дальше в программе считываются из файла значения элементов матрицы A и вектор-столбца b. Никаких изменений по сравнению с предыдущей версией программы тут нет, поэтому эту часть программы тут также опускаем (её можно посмотреть в полной версии программы на портале teach-in, или скопировать из предыдущей программы со строк 90-116). В результате на каждом процессе задаются массивы `A_part` и `b_part`.

```
53 ...
```

Существенные изменения произойдут в обработке вектора x. В отличие от предыдущей программы, мы будем хранить все элементы вектор-столбца x на всех процессах.

```
54 x = zeros(N, dtype=float64)
55 x = conjugate_gradient_method(A_part, b_part, x, N)
```

В конце программы, как всегда, строим график. Никаких изменений в этой части кода нет.

Переходим к разбору тела функции, реализующей параллельный алгоритм метода сопряжённых градиентов. Отличия тут в том, что если мы не будем распараллеливать вычисление скалярного произведения, то все вспомогательные вектора будут храниться целиком на каждом процессе. Кроме того, каждый процесс будет делать буквально одни и те же действия, чтобы не обмениваться сообщениями о результатах вычисления скалярных произведений.

```
9  def conjugate_gradient_method(A_part, b_part, x, N):
10
11     r = empty(N, dtype=float64)
12     q = empty(N, dtype=float64)
13
```

```
14      s = 1
15      p = zeros(N, dtype=float64)
16
17      while s <= N:
```

Так как x задан на всех процессах, внутри цикла на первой итерации пропадает одна команда `Allgatherv`.

```
18          if s == 1:
19              r_temp = dot(A_part.T, dot(A_part, x) - b_part)
```

Из-за того, что на каждом процессе теперь нужна не часть вектора p, а вектор p целиком, заменяем функцию `Reduce_scatter` на `Allreduce`.

```
20              comm.Allreduce([r_temp, N, MPI.DOUBLE],
21                             [r, N, MPI.DOUBLE], op=MPI.SUM)
```

Следующие действия выполняются в точности одинаково на всех процессах, так как во всех вспомогательных массивах элементы синхронизированы относительно процессов.

```
22          else:
23              r -= q / dot(p, q)
24
25          p += r / dot(r, r)
```

Дальше идет умножение матрицы на вектор и умножение транспонированной матрицы на полученный вектор. Замену `Reduce_scatter` на `Allreduce` мы уже обсудили.

```
26          q_temp = dot(A_part.T, dot(A_part, p))
27          comm.Allreduce([q_temp, N, MPI.DOUBLE],
28                         [q, N, MPI.DOUBLE], op=MPI.SUM)
```

Тело функции завершается предсказуемо:

```
29          x -= p / dot(p, q)
30          s += 1
31
32      return x
```

С теоретической точки зрения можно сказать, что мы ухудшили реализацию программы, теперь некоторые операции в теле цикла не распараллелены. К тому же, они ещё и дублируются каждым процессом. С практической точки зрения, однако, можно подсчитать, что распараллеленые операции (умножение матрицы на вектор, умножение транспонированной матрицы на вектор) требуют порядка $N \cdot M$ операций. Нераспараллеленые операции (скалярное произведение, сумма и разность векторов, деление вектора на число)

требуют примерно порядка N операций. По смыслу задачи $M > N$, матрица достаточно больших размеров, т.е. $N \gg 1$ (напомним, что если матрица малых размеров, то писать параллельную программу смысла нет). Получаем, что параллельно выполняется примерно $M/(M+1)$ часть операций. При больших M это выражение почти равно 1, а значит, время работы последовательной части программы несущественно.

С другой стороны, операций обмена данных между процессами в этой реализации значительно меньше. Поэтому на практике может оказаться, что данная упрощённая программа работает гораздо быстрее.

Замечание о версии алгоритма решения СЛАУ с регуляризацией

Для полноты картины нужно сделать некоторое замечание о методе сопряжённых градиентов. Мы уже говорили, что, в случае, если матрица A – квадратная, а вычисления проводятся абсолютно точно, метод находит решение за N итераций. Если система переопределённая, алгоритм находит минимум функционала $||Ax - b||^2$.

В теории, операция (из последовательной версии программы)

```
39 x = conjugate_gradient_method(A, b, x, N)
```

и операция, которая реализует метод Гаусса,

```
x = linalg.solve(A.T.dot(A), A.T.dot(b))
```

эквивалентны. Однако, при решении вторым методом, на практике получается дикая разболтка (см. курс ”Численные методы”, лекции 9 и 26). Чтобы подавить эту возникающую неустойчивость, минимизируют не просто квадратичный функционал, а ещё добавляют к выражению сглаживающую добавку.

```
alpha = 10**(-17)
x = linalg.solve(A.T.dot(A) + alpha*eye(N), A.T.dot(b))
```

Можно подобрать такое `alpha`, что результат будет больше похож на искомое решение. На рис. 3.1 выбрано недостаточно большое `alpha`, чтобы показать возникающую неустойчивость.

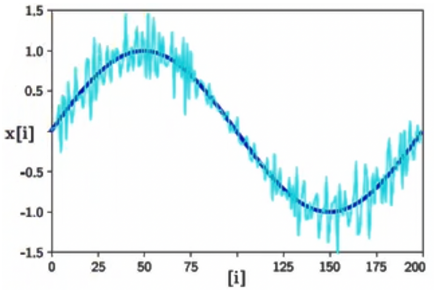


Рис. 3.1: Иллюстрация применения метода Гаусса с регуляризацией

Ещё одна из причин, почему удобнее использовать метод сопряжённых градиентов, это тот факт, что в результате работы алгоритма получается последовательность приближённых решений $x^{(s)}$. При желании программу можно оборвать на любой итерации и получить некоторое приближение. В некоторых случаях решение, визуально неотличимое от точного, можно получить уже после 1/100 части итераций. В случае с методом Гаусса остановить программу и получить промежуточный результат нельзя.

Иногда неустойчивость возникает и при применении метода сопряжённых градиентов. В таком случае также добавляют регуляризирующие слагаемые. Ниже приведены формулы метода сопряжённых градиентов с регуляризацей по А.Н. Тихонову (можно сравнить с формулами для метода без регуляризации 1.1):

$$
\begin{aligned}
r^{(s)} &= \begin{cases} A^T\left(Ax^{(s)} - b\right) + \alpha x^{(s)}, & \text{при} \quad s = 1, \\ r^{(s-1)} - \dfrac{q^{(s-1)}}{\left(p^{(s-1)}, q^{(s-1)}\right)}, & \text{при} \quad s \geqslant 2, \end{cases} \\
p^{(s)} &= p^{(s-1)} + \frac{r^{(s)}}{\left(r^{(s)}, r^{(s)}\right)}, \\
q^{(s)} &= A^T\left(Ap^{(s)}\right) + \alpha p^{(s)}, \\
x^{(s+1)} &= x^{(s)} - \frac{p^{(s)}}{\left(p^{(s)}, q^{(s)}\right)}.
\end{aligned}
\tag{3.1}
$$

Изменить исходную программу и добавить регуляризирующие слагаемые можно достаточно просто. Мы на этом останавливаться не будем.

Итак, на этой лекции мы написали свою программу, которая реализует параллельный алгоритм решения большой системы линейных алгебраических уравнений с переопределённой матрицей. На практике, если перед вами стоит похожая задача, первое, что нужно сделать, это посмотреть, нет ли готового стандартного пакета, который бы мог решить необходимую задачу. Программные пакеты разрабатываются и оптимизируются не один

год, поэтому скорее всего, они решат вашу задачу гораздо быстрее, чем программа, написанная своими силами с нуля. Программы, приведённые в этом курсе, не претендуют на наилучшую эффективность, они лишь нужны для того, чтобы показать базовые приёмы, которые используются, а также ознакомить читателя с как можно большим количеством MPI функций.

Лекция 4. Эффективность и масштабируемость параллельных программ.

На предыдущей лекции мы рассмотрели реализацию параллельного алгоритма решения СЛАУ в виде двух версий программ. На этой лекции мы рассмотрим вопрос об эффективности соответствующих программных реализаций. По итогу этой лекции мы сделаем выводы о том, что мы можем улучшить либо в нашем алгоритме, либо в соответствующих программных реализациях.

Закон Амдала

Для начала давайте определимся с терминологией. Время работы последовательной версии программы обозначим T_1. Время работы параллельной версии программы на системе из n процессоров будем обозначать T_n. Введём понятие *ускорения работы программы*:

$$S_n = \frac{T_1}{T_n}. \tag{4.1}$$

Понятно, что если поделить время работы последовательной версии программы на время параллельной работы программы, то мы получим то, во сколько раз параллельная программа работает быстрее, чем последовательная.

Интуитивно понятно, что если нет никаких накладных расходов, мы ожидаем, что ускорение будет равно $S_n = n$. На практике так, конечно, случается редко. Часто в программе есть части, которые невозможно распараллелить.

Обозначим p – доля параллельных вычислений в программе ($0 \leqslant p \leqslant 1$). Тогда, соответственно, $(1 - p)$ – доля последовательных вычислений. При запуске программы на системе из n процессоров параллельная часть программы будет ускоряться в n раз, а время работы последовательной части программы будет оставаться неизменным. Сформулируем *закон Амдала*:

$$S_n \leqslant \frac{1}{(1-p) + \dfrac{p}{n}}. \tag{4.2}$$

Здесь нельзя поставить знак равенства, так как под ускорением мы подразумеваем реальное ускорение, т.е. ещё есть накладные расходы на пересылку сообщений между процессорами, которые приводят к замедлению работы параллельной программы.

Построим теоретические графики для различных значений p (рис. 4.1). По графикам можно увидеть, что если в программе присутствует хоть небольшая доля последовательных вычислений, то ускорение сверху ограничивается предельно возможным значением. Действительно,

$$S_n \leqslant \lim_{n\to\infty} \frac{1}{(1-p)+\dfrac{p}{n}} = \frac{1}{(1-p)}.$$

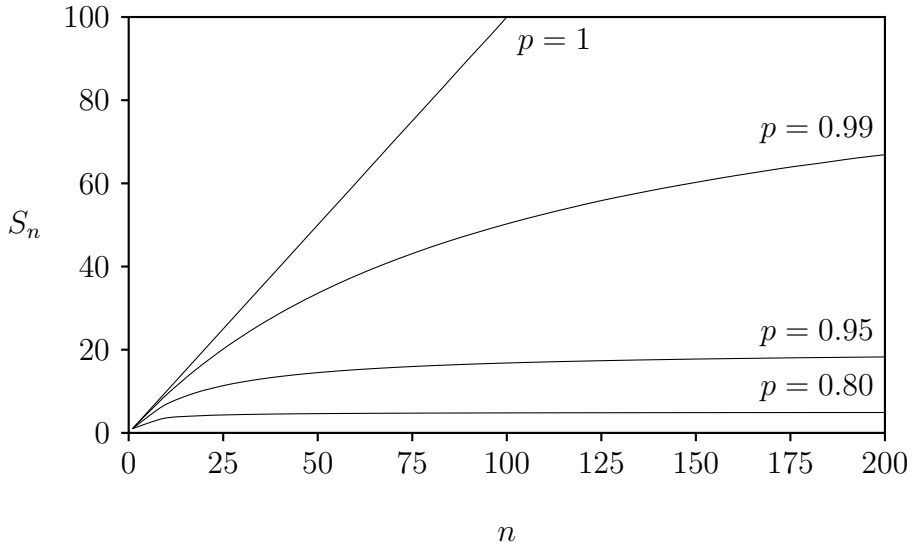


Рис. 4.1: Графики зависимости усорения программы от числа процессов для различных p

Оценки предельно возможного ускорения параллельных алгоритмов, разобранных на Лекции 3

Сделаем оценку предельно возможного ускорения для части программы, реализующей метод сопряжённых градиентов. В примере 3-1 (номера соответствуют номерам файлов, выложенных на портале teach-in в материалах к лекциям) нам удалось распараллелить все операции. Поэтому ожидаем $p = 1$. В теории мы можем ожидать линейного ускорения. На практике, как мы скоро убедимся, мы его не достигнем из-за накладных расходов.

В примере 3-2 мы не стали распараллеливать некоторые операции, чтобы упростить программную реализацию и сэкономить на передаче сообщений между процессами. Внимательно посмотрев на алгоритм 1.1, почитаем количество рапараллеленых и нераспараллеленых операций.

Чтобы посчитать каждое скалярное произведение, необходимо $(2N-1)$ операций. На каждой итерации скалярное произведение встречается 3 раза (кроме первой, на которой только 2 раза). Итого $(2N-1)\cdot(3N-1)$ операций.

На каждой итерации выполняется 3 операции сложения векторов. Если длина векторов N, то нужно сделать N операций сложения. Только на первой итерации присутствует одно сложение двух векторов длины M. В итоге получаем $N\cdot(3N-1)+M\cdot 1$ операций.

Операция умножения матрицы на вектор требует $M(2N-1)$ операций, умножение транспонированной матрицы на вектор – $N(2M-1)$ операций. На каждой итерации выполняется по одному такому действию, а на первой итерации – два. Итого получаем $M(2N-1)\cdot(N+1)+N(2M-1)\cdot(N+1)$ операций.

Сразу упомянем, что мы предполагаем, что все операции выполняются процессором за одинаковое время. Это, на самом деле, не так. Например, операция деления требует гораздо больше тактов процессора, чем умножение или сложение.

Вспоминая о том, что мы распараллелили только умножение матрицы на вектор и умножение транспонированной матрицы на вектор, а также вычитание векторов длины M на первой итерации, получаем:

$$S_n \leqslant \frac{4MN^2+3MN+8N^2-7N+1}{(9N^2-6N+1)+\dfrac{4MN^2+3MN-N^2-N}{n}}.$$

После преобразования можно получить

$$S_n \leqslant \frac{n\left(1+\dfrac{3}{4N}+\dfrac{8}{M}+-\dfrac{7}{MN}+\dfrac{1}{4MN^2}\right)}{n\left(\dfrac{9}{M}-\dfrac{6}{MN}+\dfrac{1}{4MN^2}\right)+1+\dfrac{3}{4N}-\dfrac{1}{4M}-\dfrac{1}{4MN}}.$$

Если $n \ll M$, получаем $S_n \leqslant n$.

Реальное ускорение программных реализаций параллельных алгоритмов, разобранных на Лекции 3

Оценим реальное ускорение S_n частей программы, реализующей метод сопряжённых градиентов. Для этого воспользуемся встроенной функцией `MPI.Wtime`. Сохраним текущее время перед и после вызова функции, тогда их разницей будет время работы функции.

```
if rank == 0:
    start_time = MPI.Wtime()

x = conjugate_gradient_method(A_part, b_part, x, N)

if rank == 0:
    end_time = MPI.Wtime()
    print(f'Elapsed time is {end_time - start_time:.4f} sec')
```

Открытым вопросом остаётся то, эффективно ли использовать для таких тестов многоядерный ПК. Например, если попробовать запустить программы 3-1 и 3-2 на обычном домашнем компьютере, то, скорее всего, последовательная и параллельная программа будут работать примерно одинаковое время. Это связано с особенностями пакета `numpy`. Самые вычислительно ёмкие операции мы выполняем с помощью функции `dot`, а эта функция вычисляет значение многопоточно, то есть задействуются все ядра процессора. То есть мы либо запускаем последовательную версию программы, но функция `dot` считается с использованием нескольких потоков, либо запускаем параллельную версию программы, и тогда на каждом потоке `dot` считается последовательно.

Мы для тестирования будем использовать суперкомпьютер ”Ломоносов-2”. Для коротких задач на нём есть специальный раздел test, на котором можно запускать программы, работающие не дольше 15 минут. Также не разрешается использовать больше 15 узлов на задачу. На каждом узле установлены 14-ядерные процессоры Intel Xeon E5-2697 v3 2.60GHz. Выделяется 64GB оперативной памяти, т.е. 4.5GB на ядро.

В качестве тестовых параметров возьмём $N = 200$, $M = 20\,000\,000$. Тогда получаем, что для хранения матрицы `A` необходимо $N \times M \times 8$ байт $= 29.8$GB. При таких параметрах последовательная версия программы на 1 узле работает 246.7 секунд. Нужно помнить, однако, что последовательная версия программы все равно задействует все 14 ядер, так что в качестве T_1 будем воспринимать это число, домноженное на 14.

График зависимости времени работы параллельной версии программы можно увидеть на рис. 4.2. Число ядер приведено с шагом 14, после каждой засечки добавляется новый узел. С увеличением числа ядер n время T_n падает, значит увеличение количества

ядер даёт нужный эффект. Сиреневым цветом показаны особые случаи. В этих точках один процесс является управляющим, а между остальными матрица распределяется поровну. То есть все ядра, кроме управляющего, загружены одинаково. По графику видно, что в таком случае время работы программы, естественно, становится меньше.

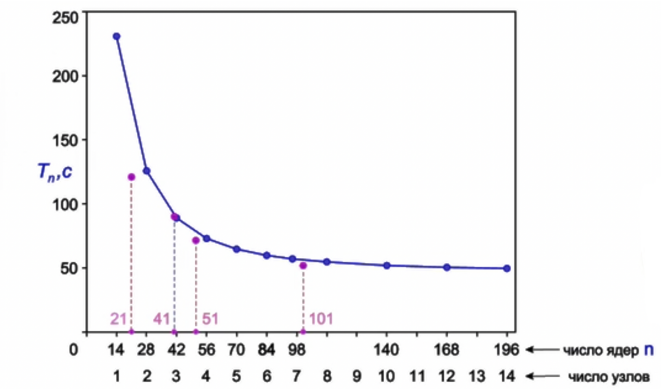


Рис. 4.2: График зависимости времени работы программы 3-1 от числа ядер

Используя оценку времени работы последовательной версии программы, по формуле 4.1 почитаем ускорение S_n. График можно увидеть на рис. 4.3.

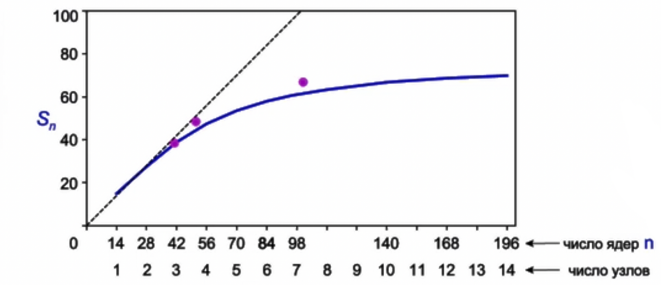


Рис. 4.3: График ускорения работы программы 3-1

Аналогичный график можно построить и для программы 3-2 (рис. 4.4). Как видим для этого примера, если использовать не больше 60 ядер, программы работают одинаково (хотя 3-2 было гораздо легче реализовывать).

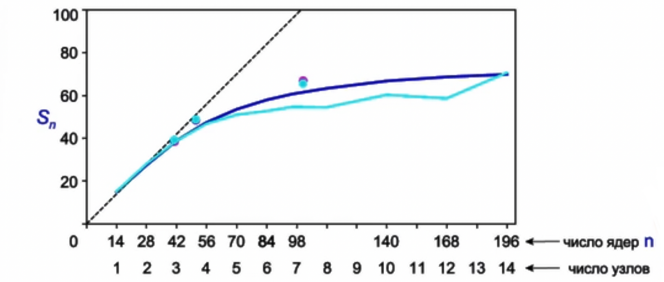


Рис. 4.4: График ускорения работы программ 3-1 (синий) и 3-2 (голубой)
$N = 200$, $M = 20\,000\,000$

Изменим входные параметры. Увеличим N в 2.5 раза, а M уменьшим в 2.5 раза. Таким образом, размер матрицы остался прежним. Операции умножения матрицы на вектор и умножение транспонированной матрицы на вектор также не поменяются, а вот операции скалярного произведения станут более значимыми. Также по размеру увеличатся сообщения, которыми обмениваются процессы. График ускорения немного понизится, см. рис. 4.5.

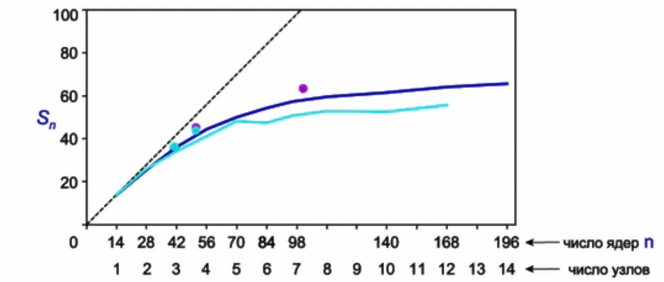


Рис. 4.5: График ускорения работы программ 3-1 (синий) и 3-2 (голубой)
$N = 500$, $M = 8\,000\,000$

В третьем случае ещё увеличим N в 2 раза, а M уменьшим в 4 раза. Тогда размер матрицы уменьшится в 2 раза, а значит, операции умножения также станут в 2 раза менее затратными. Однако, количество итераций возрастёт в 2 раза, так что итоговая вычислительная сложность программы не изменится. При этом доля времени, которое тратится на приём и передачу сообщений, возрастёт значительно. Во-первых, количество итераций возросло, а значит суммарное количество пересылок сообщений тоже. Во-вторых, размеры сообщений возросли, так как они определяются величиной N. Исходя из всего вышеперечисленного понятно, почему график проседает значительно (рис. 4.6). Видно, что больше, чем 25 ядер, использовать просто нет смысла.

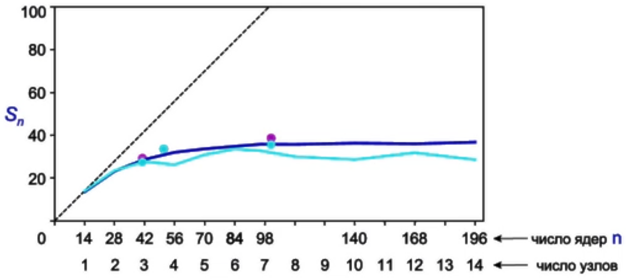


Рис. 4.6: График ускорения работы программ 3-1 (синий) и 3-2 (голубой)
$N = 1\,000$, $M = 2\,000\,000$

Эффективность параллельных программ

Введём понятие *эффективности распараллеливания*:

$$E_n = \frac{S_n}{n}. \tag{4.3}$$

Отличие эффективности распараллеливания E_n от 1 характеризует целесообразность использования n процессоров (ядер), то есть то, насколько полно мы используем вычислительные ресурсы. Напомним, что предельно возможно ускорение $S_n = n$ достигается, когда в программе нет последовательных частей, весь код распараллелен.

Для уже знакомых нам тестовых наборов построим графики зависимости эффективности распараллеливания от числа ядер.

Для первого тестового набора график эффективности представлен рис. 4.7. Есть негласная договорённость, что программы имеет смысл распараллеливать, если эффективность не меньше 80%. Для данного случая, как видно, имеет смысл использовать не больше 70 ядер. При использовании большего числа ядер эффективность уже становится неразумной.

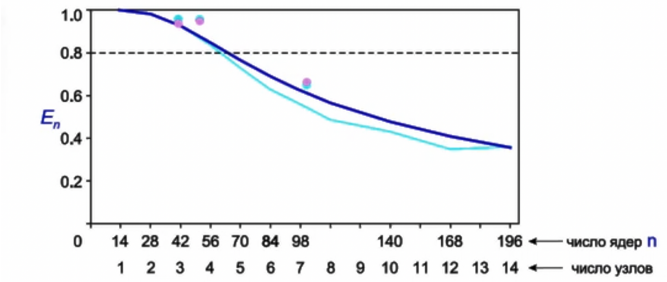


Рис. 4.7: График эффективности распараллеливания программ 3-1 (синий) и 3-2 (голубой) $N = 200$, $M = 20\,000\,000$

Для второго набора параметров (см. рис. 4.8) не имеет смысла использовать больше 60 ядер.

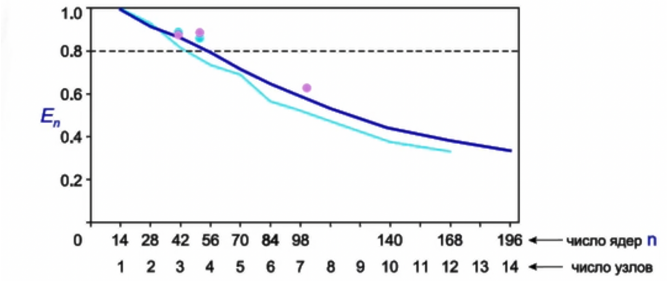


Рис. 4.8: График эффективности распараллеливания программ 3-1 (синий) и 3-2 (голубой) $N = 500$, $M = 8\,000\,000$

Для третьего набора параметров (рис. 4.9) эффективность резко падает уже после использования 40 ядер.

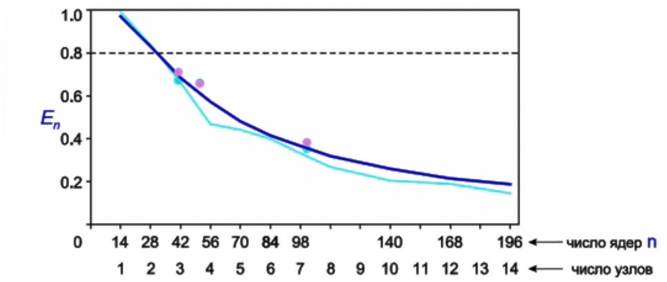


Рис. 4.9: График эффективности распараллеливания программ 3-1 (синий) и 3-2 (голубой) $N = 1\,000$, $M = 2\,000\,000$

Графики эффективности дают понять, какое количество ресурсов (число используе-

мых ядер) имеет смысл использовать для решения нашей задачи (с определённой структурой входных данных и с определённой программной реализацией).

На основе построенных графиков можно понять, что написанные нами программы – не такие уж и идеальные. С ними есть явные проблемы, и программы, и алгоритмы требуют определённого усовершенствования.

Масштабируемость параллельных программ

Под *масштабируемостью* понимается способность параллельной программы сохранять эффективность при увеличении какого-либо параметра. Вводят несколько вариантов определения масштабируемости.

Сильная масштабируемость (strong scaling) – способность параллельной программы сохранять эффективность E_n при увеличении числа процессоров n для задачи фиксированной сложности ($T_1 = const$).

Сильная масштабируемость – самый хороший вид масштабируемости. Представьте, что вам необходимо решить определённую задачу. У вас есть входные данные определённого размера, который фиксируется в результате экспериментальных наблюдений. Получается, что вычислительная сложность задачи – фиксирована, последовательная программа считается за время T_1. Тогда, если есть сильная масштабируемость, то чем на большем числе процессов мы будем запускать программу, тем быстрее будет получаться результат, то есть действительно оправданно использовать дополнительные ресурсы.

На наших примерах мы видим, что сильная масштабируемость присутствует до определённого момента. Например, подключать 2, 3, 4 и так далее ядра имеет смысл, а вот несколько десятков узлов – уже нет.

Слабая масштабируемость (weak scaling) – способность параллельной программы сохранять эффективность E_n при увеличении числа процессоров n и одновременном сохранении объёма работы, приходящегося на каждый процессор ($T_1/n = const$).

Если на одном ядре быстро вычисляется одна задача, то, при наличии слабой масштабируемости, на большом количестве ядер так же эффективно будет решаться аналогичная задача большей размерности.

Масштабируемость вширь (wide scaling) – способность параллельной программы сохранять эффективность E_n при увеличении сложности задачи T_1 для фиксированного числа процессоров ($n = const$).

Проблемы оценки эффективности и масштабируемости

Как мы успели убедиться, оценка эффективности и масштабируемости – задача крайне сложная и неоднозначная. Большую роль играют характеристики входных данных. Например, если бы мы взяли для наших программ 3-1 и 3-2 параметры $N = 10\,000$, $M = 15\,000$, программы бы работали крайне неэффективно, а масштабируемость бы отсутствовала. Это связано с тем, что значение N определяет размеры пересылаемых массивов.

Невозможно написать одну параллельную программу, которая бы решала огромный класс задач сразу.

Ещё раз сделаем ссылку на курс ”Суперкомпьютеры и параллельная обработка данных”. В частности, на 3 лекции Вл.В.Воеводин очень подробно обсуждает все соответсвующие нюансы с примерами.

Также сошлёмся на статью David H. Bailey ”Twelve ways to fool the masses when giving performance results on parallel computers”(RNR Technical Report RNR-91-020, 1991), которая очень часто упоминается, когда говорят о демонстрации эффективности написанных параллельных программ. В этой статье в шутливой форме приводятся 12 рецептов, что нужно сделать, чтобы продемонстрировать эффективность работы программы, даже если она на самом деле написана неэффективно.

Одним из пунктов, советом использовать специфический вид входных данных, мы уже воспользовались. Другой пункт говорит о том, что не нужно демонстрировать эффективность работы всей программы, достаточно продемонстрировать эффективность только ядра программы, т.е. основной части.

Упоминание о самых проблемных MPI-функциях, использование которых снижает эффективность рассмотренных ранее алгоритмов

Появляется желание убрать зависимость эффективности распараллеливания от величины N. Вспомним три основные MPI-функции, которые создают основные проблемы с эффективностью.

Во-первых, это

```
comm.Allgatherv([p_part, N_part, MPI.DOUBLE],
                [p, rcounts_N, displs_N, MPI.DOUBLE])
```

В итоге рассылается массив `N` по всем процессам.

Во-вторых, это

```
comm.Reduce_scatter([q_temp, N, MPI.DOUBLE],
                    [q_part, N_part, MPI.DOUBLE],
                    recvcounts=rcounts_N, op=MPI.SUM)
```

Здесь суммируются массивы длины `N`, которые располагаются на каждом процессе.

В-третьих, это

```
comm.Allreduce([q_temp, N, MPI.DOUBLE],
               [q, N, MPI.DOUBLE], op=MPI.SUM)
```

Аналогичная проблема, снова есть операции с массивами длины `N`.

Хотелось бы избавиться от пересылки массивов такой большой длины. На следующей лекции мы обсудим гораздо более продвинутый алгоритм перемножения матрицы на вектор. Будем делить матрицу одновременно и по горизонтали, и по вертикали.

Лекция 5. Операции с группами процессов и коммуникаторами.

На первых трёх лекциях мы реализовали параллельную версию программы для решения СЛАУ методом сопряжённых градиентов (1.1). Этот метод используют для решения систем большой размерности с плотно заполненной матрицей. На лекции 4 мы исследовали написанные программы на эффективность и сформулировали условия их применимости. В частности, нам удалось увидеть, что процессы приёма и передачи сообщений требуют существенное количество ресурсов. Поэтому на этой лекции мы рассмотрим другой алгоритм умножения матрицы на вектор, который можно использовать в программе для метода сопряжённых градиентов. Новый алгоритм позволит существенно сократить расходы, связанные с приёмом и передачей сообщений.

Параллельный алгоритм умножения матрицы на вектор в случае двумерного деления матрицы на блоки

В предыдущем случае мы разбивали матрицу на блоки между процессами по строкам. Каждому процессу доставалась своя часть матрицы, а вектор, на который умножается матрица, доставался целиком. В результате умножения на каждом процессе получалась своя часть итогового вектора (см. рис. 1.3). В данном случае основной проблемой является то, что вектор, на который умножается матрица, должен хранится целиком на всех процессах. На каждой итерации метода сопряжённых градиентов этот вектор каким-то образом изменяется, после чего необходимо обновлять его на всех процессах перед операцией умножения.

Будем разбивать матрицу на блоки как по строкам, так и по столбцам. Каждый элемент итогового вектора – это, по сути, скалярное произведение строки матрицы на столбец (см. рис. 2.1). Если посмотреть на строку из блоков матрицы, каждый блок может независимо от других процессов умножаться на соответствующую часть вектора x. Затем полученные части с разных процессов нужно будет сложить. Схему итогового алгоритма можно увидеть на рис. 5.1 (в данном случае представлено разбиение для 16 процессов).

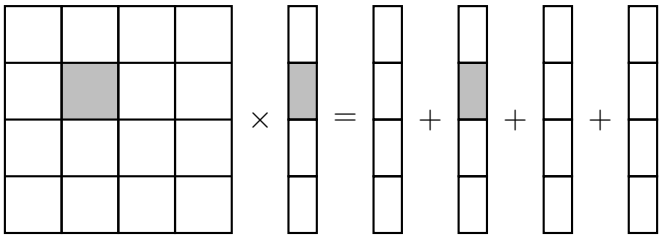


Рис. 5.1: Схема умножения матрицы на вектор

Обратите внимание, что теперь каждая часть вектора x хранится не на всех процессах, но и не на одном, а на всех процессах, где хранится соответсвующий ему фрагмент матрицы. В итоге на каждом процессе хранится блок матрицы, часть вектора x и часть полученного вектора.

Приступим к программной реализации этого алгоритма. Для начала возьмём пример входных данных из первой лекции и напишем алгоритм умножения матрицы на вектор в рамках отдельной программы. Чтобы не повторяться, на этот раз нулевой процесс также будет участвовать в вычислениях, а не только считывать и подготавливать данные. Кстати, это также позволит нам запускать программу на 1 процессе (т.е. почти последовательно).

Традиционно начинаем программу с импорта необходимых библиотек и инициализации MPI:

```
1  from mpi4py import MPI
2  from numpy import empty, array, int32, float64, sqrt, hstack
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  numprocs = comm.Get_size()
```

Напоминаем, что `MPI.COMM_WORLD` – коммуникатор, в который входят все процессы, участвующие в вычислениях. С помощью методов `Get_rank` и `Get_size` можно узнать уникальный номер конкретного процесса и суммарное количество процессов, входящих в данный коммуникатор.

Считываем из файла размерность матрицы A:

```
7  if rank == 0:
8      f1 = open('in_test.dat', 'r')
9      N = array(int32(f1.readline()))
10     M = array(int32(f1.readline()))
11     f1.close()
```

Дальше мы будем определяться, как разбивать матрицу A по блокам. Для простоты определим количество разбиений по строкам и столбцам как корень из числа процессов.

```
12 num_col = num_row = int32(sqrt(numprocs))
```

С учётом этого допущения становится ясно, что программу можно будет запускать не на любом числе процессов, а только в случаях, когда число процессов является квадратом натурального числа (4, 9, 16 и так далее).

Дальше следует уже стандартная функция для подготовки вспомогательных массивов при делении некоторого массива на части. В данном случае существенным отличием является то, что также участвует нулевой процесс. Смысл `rcounts` и `displs` можно вспомнить по рис. 1.4.

```
13 def auxiliary_arrays_determination(M, num):
14     ave, res = divmod(M, num)
15     rcounts = empty(num, dtype=int32)
16     displs = empty(num, dtype=int32)
17 
18     for k in range(num):
19         if k < res:
20             rcounts = ave + 1
21         else:
22             rcounts = ave
23 
24         if k == 0:
25             displs[0] = 0
26         else:
27             displs[k] = displs[k-1] + rcounts[k-1]
28 
29     return rcounts, displs
```

Создаём вспомогательные массивы для N и M на нулевом процессе (разбиваем на `num_row` и `num_col`, а не на `numprocs`). На остальных процессах создаём переменные `M_part` и `N_part`, чтобы не хранить эти массивы целиком.

```
30 if rank == 0:
31     rcounts_M, displs_M = auxiliary_arrays_determination(M, num_row)
32     rcounts_N, displs_N = auxiliary_arrays_determination(N, num_col)
33 else:
34     rcounts_M, displs_M = None, None
35     rcounts_N, displs_N = None, None
36 
37 M_part = array(0, dtype=int32)
38 N_part = array(0, dtype=int32)
```

Знакомство с функцией Split, позволяющей разбить коммуникатор на несколько новых коммуникаторов

На рис. 5.2 приведён пример разбиения матрицы 10×8 для хранения на 9 процессах. Договоримся, что будем располагать блоки матрицы по процессам в указанной очерёдности. На самом деле, это можно бы было делать как угодно.

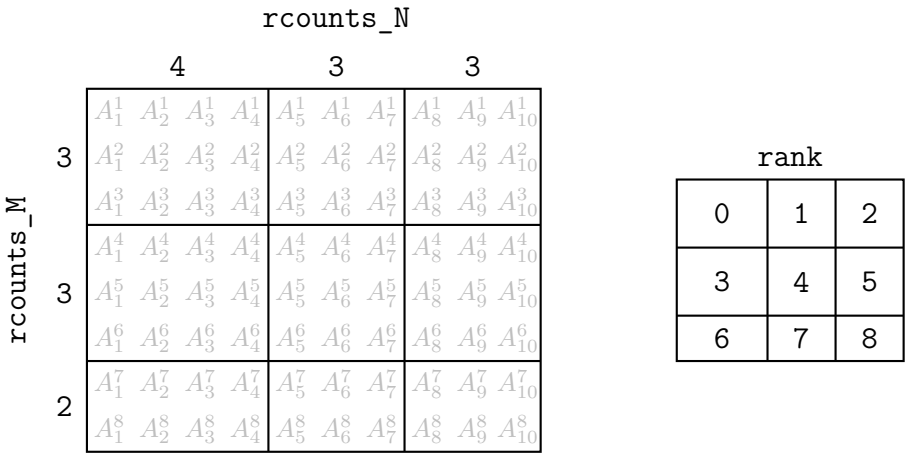


Рис. 5.2: Пример разбиения матрицы A на блоки

Операция рассылки блоков матрицы по процессам вызывает некоторые вопросы. Можно было бы рассылать части матрицы с помощью функций `Send`/`Recv`, но это неэффективно. Хотелось бы воспользоваться функциями коллективного взаимодействия процессов, но они запускаются на всех процессах данного коммуникатора. Ничего не остаётся, как ввести новые коммуникаторы.

Функция `Split` помогает разделить имеющийся у нас коммуникатор `comm` на несколько других (при этом исходный коммуникатор, конечно, никуда не исчезает).

```
39 comm_col = comm.Split(rank % num_col, rank)
```

Первым аргументом (который называется `color`) передаётся целое неотрицательное число. Процессы с одинаковым `color` собираются в отдельную группу, из которой создаётся свой коммуникатор `comm_col`.

Группа процессов – это упорядоченное множество процессов. Коммуникатор – это группа процессов и контекст обмена сообщений.

В нашем случае видно, что в результате поиска остатка от деления `rank % num_col` получатся коммуникаторы с процессами, на которых хранятся блоки из одного столбца.

Упорядоченность процессов среди нового коммуникатора определяет второй аргумент функции `Split`, который называют `key`. Обратите внимание, что внутри нового коммуникатора процессы нумеруются по-другому.

Можно заметить, что в случае 5.2 создаётся ещё 3 коммуникатора. То есть переменная `comm_col` не равна между собой на всех процессах.

При таком делении нам удастся добиться, чтобы с ростом числа используемых процессов для решения задачи накладные расходы отчасти падали.

Аналогично создаём коммуникаторы для разбиения по строкам.

```
40 comm_row = comm.Split(rank // num_col, rank)
```

В результате у нас есть уже 7 коммуникаторов, а каждый процесс входит в 3 из них (`comm`, один из `comm_col` и один из `comm_row`). На рис. 5.3 показаны все созданные коммуникаторы и нумерация процессов внутри них.

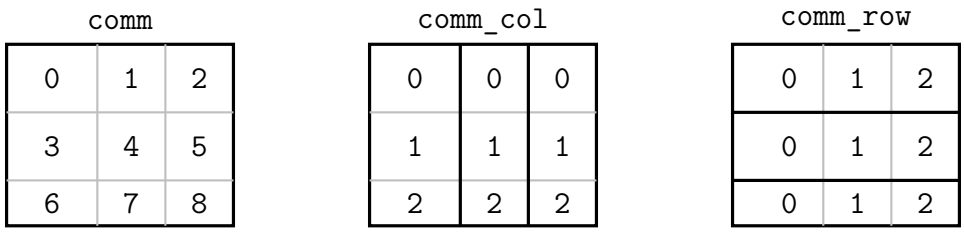


Рис. 5.3: Иллюстрация создания новых коммуникаторов с помощью функции `Split`

Особенности работы функций коллективного взаимодействия процессов в случае, когда эти процессы одновременно входят в различные коммуникаторы

```
41 if rank in range(num_col):
42     comm_row.Scatter([rcounts_N, 1, MPI.INT],
43                      [N_part, 1, MPI.INT], root=0)
```

Данная функция вызывается для коммуникатора `comm_row`, а значит аргумент `root=0` – это нулевой процесс в рамках данного коммуникатора, в общем случае не обязан совпадать с нашим нулевым процессом. В данном случае, правда, условие `if` помогает нам выбрать только процессы с номерами 0, 1, 2, которые вместе входят в один коммуникатор `comm_row` и где нулевой процесс внутри `comm_row`, действительно, совпадает с нулевым

процессом внутри `comm`. Если бы условия `if` не было, эта команда была бы вызвана на всех процессах и запрашивала значение `rcounts_N` также на процессах 3, 6, где это значение равно `None`.

В результате предыдущей операции переменную `N_part` удалось определить на процессах 0, 1, 2, которые являются нулевыми на всех коммуникаторах `comm_col` (см. рис. 5.2). Это позволяет нам вызвать на всех коммуникаторах `comm_col` операцию `Bcast`, так как внутри этих коммуникаторов переменная `N_part` остаётся постоянной.

```
44 comm_col.Bcast([N_part, 1, MPI.INT], root=0)
```

Аналогичные действия проведём для инициализации переменных `M_part` на всех процессах. Только, в отличие от `N_part`, вначале разошлём `rcounts_M` по процессам 0, 3, 6, так как именно на них значения `M_part` будут различаться.

```
45 if rank in range(0, numprocs, num_col):
46     comm_col.Scatter([rcounts_M, 1, MPI.INT],
47                      [M_part, 1, MPI.INT], root=0)
48
49 comm_row.Bcast([M_part, 1, MPI.INT], root=0)
```

В результате переменные `N_part` и `M_part` определены на каждом процессе. Можно также заметить, что операции на строках 44 и 49 проходили параллельно (хотя мы ещё только на этапе подготовки данных).

Теперь можно выделить память под блок матрицы A.

```
50 A_part = empty((M_part, N_part), dtype=float64)
```

Создание нового коммуникатора из группы процессов. Базовые операции с группами процессов

На данный момент перед нами стоит задача считать значения элементов матрицы A из файла на нулевом процессе, а затем разослать данные по блокам на все процессы. В файле данные хранятся подряд: вначале идут элементы первой строки матрицы, потом второй, и так далее. Помним, что отличие данного случая от предыдущих в том, что первый процесс также участвует в вычислениях.

Будем создавать вспомогательные коммуникаторы. В каждый такой коммуникатор будут входить процессы, на которых хранится одна строка блоков (то есть, процессы коммуникатора `comm_row`) плюс нулевой процесс. При этом, для первой строки нулевой коммуникатор будет отправлять часть данных себе, а для последующих просто рассылать

данные другим процессам.

Для этого вначале получим группу процессов, входящих в `comm`.

```
51 group = comm.Get_group()
```

Из созданной группы мы будем компоновать новые коммуникаторы.

Будем считывать сразу одну строку из блоков матрицы целиком (для этого создаём временный вспомогательный массив `a_temp`). Так как дальше будем использовать функцию `Scatterv`, записываем считываемые блоки не подряд, а по-другому (поэтому может показаться, что в квадратных скобках на строках 61-62 происходит что-то странное). Если внимательно посмотреть, мы всего лишь записываем элементы матрицы так, чтобы рассылаемые блоки хранились непрерывно, без промежутков.

```
52 if rank == 0:
53     f2 = open('AData.dat', 'r')
54 
55     for m in range(num_row):
56         a_temp = empty(rcounts_M[m]*N, dtype=float64)
57 
58         for j in range(rcounts_M[m]):
59             for n in range(num_col):
60                 for i in range(rcounts_N[n]):
61                     a_temp[rcounts_M[m]*displs_N[n] +
62                            j*rcounts_N[n] + i] = float64(f2.readline())
```

Если `m` равно нулю, то никаких дополнительных коммуникаторов создавать не нужно, мы просто будем использовать коммуникатор `comm_row`. Так как мы вызываем функцию коллективного взаимодействия, не забудем потом вызвать эту функцию на ненулевых процессах коммуникатора `comm_row` (строки 83-84).

```
63         if m == 0:
64             comm_row.Scatterv([a_temp, rcounts_M[m]*rcounts_N,
65                                rcounts_M[m]*displs_N, MPI.DOUBLE],
66                               [A_part, M_part*N_part, MPI.DOUBLE], root=0)
```

Если `m` не равно нулю, нам необходимо создать новый коммуникатор. На основе группы процессов, которую мы получили из коммуникатора `comm`, создаём вспомогательную подгруппу `group_temp`. Метод группы `Range_incl` позволяет на основе существующей группы создать новую.

```
67         else: # m != 0
68             group_temp = group.Range_incl([(0,0,1),
69                                            (m*num_col,(m+1)*num_col-1,1)])
```

В качестве аргумента передаётся список упорядоченных промежутков из номеров процессов, которые мы включаем в новую группу. В данном случае, во-первых, включаются процессы, начиная с `0` и заканчивая `0` (включительно) с шагом `1` (то есть, только нулевой процесс). Во-вторых, включаются процессы, начиная с `m*num_col` и до `(m+1)*num_col-1` (включительно) с шагом `1`.

Теперь создаём из полученной группы временный коммуникатор `comm_temp`.

```
70              comm_temp = comm.Create(group_temp)
```

Не забудем потом создать эти группы и коммуникаторы на ненулевых процессах (строки 87-89).

Между операцией создания группы и операцией создания коммуникатора есть некоторые существенные отличия. При создании группы, каждый процесс независимо друг от друга просто создаёт свою локальную переменную `group_temp`. Создание нового коммуникатора (`Create`) – коллективная операция, которая должна вызываться одинаково на всех процессах исходного коммуникатора. В результате её вызова между указанными процессами создаётся контекст обмена сообщений.

Дальше остаётся только создать временные массивы `rcounts_N_temp` и `displs_N_temp` для вызова `Scatterv` на созданном временном коммуникаторе. Отличие этих временных массивов от исходных `rcounts_N` и `displs_N` в том, что в начало добавляется ещё один элемент, который отвечает за сообщение на нулевой процесс.

```
71              rcounts_N_temp = hstack((array(0, dtype=int32), rcounts_N))
72              displs_N_temp = hstack((array(0, dtype=int32), displs_N))
73
74              comm_temp.Scatterv([a_temp, rcounts_M[m]*rcounts_N_temp,
75                                  rcounts_M[m]*displs_N_temp, MPI.DOUBLE],
76                                 [empty(0, dtype=float64), 0, MPI.DOUBLE],
77                                 root=0)
```

`Scatterv` – коллективная операция, которая должна быть вызвана на всех процессах. На ненулевых она вызывается на строках 92-94.

После этого временные коммуникаторы можно удалить.

```
78              group_temp.Free()
79              comm_temp.Free()
```

В конце цикла также не забудем закрыть файл.

```
80      f2.close()
```

Мы разобрались с нулевым процессом, но нам осталось дописать часть с ненулевыми процессами. Вспоминаем о том, какие коллективные операции мы вызывали на нулевом процессе, и вызываем их на ненулевых.

```
81  else: # rank != 0
82      if rank in range(num_col):
83          comm_row.Scatterv([None, None, None, None],
84                            [A_part, M_part*N_part, MPI.DOUBLE], root=0)
85
86      for m in range(1, num_row):
87          group_temp = group.Range_incl([(0,0,1),
88                                         (m*num_col,(m+1)*num_col-1,1)])
89          comm_temp = comm.Create(group_temp)
90
91          if rank in range(m*num_col, (m+1)*num_col):
92              comm_temp.Scatterv([None, None, None, None],
93                                 [A_part, M_part*N_part, MPI.DOUBLE],
94                                 root=0)
95
96              comm_temp.Free()
97          group_temp.Free()
```

Обратите внимание, что вызвать `comm_temp.Free()` можно только на процессах, которые реально входят в этот коммуникатор (в отличие от операции создания этого коммуникатора, которая должна была быть вызвана на всех процессах исходного коммуникатора `comm`).

Программа, реализующая параллельный алгоритм умножения матрицы на вектор в случае двумерного деления матрицы на блоки

На данный момент мы считали матрицу A из файла и распределили её по блокам на все процессы. Теперь считаем вектор x на нулевом процессе:

```
98  if rank == 0:
99      x = empty(N, dtype=float64)
100     f3 = open('xData.dat', 'r')
101     for i in range(N):
102         x[i] = float64(f3.readline())
103     f3.close()
104 else:
105     x = None
106
107 x_part = empty(N_part, dtype=float64)
```

На одном коммуникаторе `comm_col` (см. рис. 5.3) хранятся одни и те же части `x_part`.

Операция `Scatterv` должна вызываться на коммуникаторах `comm_row`, но единственный такой коммуникатор, на котором это можно сделать, это коммуникатор, на котором есть исходный нулевой процесс. Поэтому, вызываем `Scatterv` только на этом коммуникаторе, а затем используем `Bcast`. Похожие действия мы выполняли при рассылке переменных `N_part`.

```
108 if rank in range(num_col):
109     comm_row.Scatterv([x, rcounts_N, displs_N, MPI.DOUBLE],
110                       [x_part, N_part, MPI.DOUBLE], root=0)
111
112 comm_col.Bcast([x_part, N_part, MPI.DOUBLE], root=0)
```

Дальше следует умножение блока матрицы на часть вектора.

```
113 b_part_temp = dot(A_part, x_part)
```

Следуя алгоритму (см. рис. 5.1), нужно просуммировать полученные части внутри коммуникатора `comm_row`.

```
114 b_part = empty(M_part, dtype=float64)
115 comm_row.Reduce([b_part_temp, M_part, MPI.DOUBLE],
116                 [b_part, M_part, MPI.DOUBLE], root=0)
```

В результате этой команды мы получаем правильные части вектора b, которые хранятся на процессах 0, 3, 6 (в нашем примере).

Можно собрать полученный вектор на нулевом процессе.

```
117 if rank == 0:
118     b = empty(M, dtype=float64)
119 else:
120     b = None
121
122 if rank in range(0, numprocs, num_col):
123     comm_col.Gatherv([b_part, M_part, MPI.DOUBLE],
124                      [b, rcounts_M, displs_Mn MPI.DOUBLE], root=0)
```

Новая параллельная версия программы, реализующей решение СЛАУ с помощью метода сопряжённых градиентов

Теперь применим новый алгоритм умножения матрицы на вектор для очередной версии параллельной программы для решения СЛАУ методом сопряжённых градиентов.

Умножение матрицы на вектор мы уже рассмотрели, а умножение транспонированной матрицы на вектор будет выглядеть очень похожим образом. Нужно просто поменять местами N и M, а также коммуникаторы `comm_col` и `comm_row`.

Приведём программу для вычисления метода сопряжённых градиентов. Целиком она доступна на портале teach-in в материалах к лекциям.

```
1  from mpi4py import MPI
2  from numpy import empty, array, int32, float64, zeros, arange, dot, sqrt,
     hstack
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  numprocs = comm.Get_size()
7
8  def conjugate_gradient_method(A_part, b_part, x_part,
9                                N_part, M_part, N, comm,
10                               comm_row, comm_col, rank):
11     ...
```

Как обычно, оставим рассмотрение основной функции на потом. Единственное, на что стоит сейчас обратить внимание, это то, что переменная `N` нужна на всех процессах, так как она отвечает за количество итераций.

```
65 if rank == 0:
66     f1 = open('in.dat', 'r')
67     N = array(int32(f2.readline()))
68     M = array(int32(f2.readline()))
69     f1.close()
70 else:
71     N = array(0, dtype=int32)
72
73 comm.Bcast([N, 1, MPI.INT], root=0)
74
75 num_col = num_row = int32(sqrt(numprocs))
```

Функция для генерации вспомогательных массивов никак не изменится, её можно взять из предыдущей программы, рассмотренной на этой лекции.

```
76 def auxiliary_arrays_determination(M, num):
77     ...
```

Дальше идёт без изменений подготовка массивов `rcounts_N`, `displs_N`, `rcounts_M` и `displs_M`, переменных `N_part` и `M_part`. Считывание и рассылка по блокам матрицы A также остаётся без изменений. По сути можно просто скопировать строки 13-97 из предыдущей программы, которую мы рассмотрели на этой лекции.

```
...
```

Перейдем к новой части. В отличие от предыдущей программы, здесь нужно считать вектор b из файла. Ранее мы уже заметили, что рассылка вектора x была похожа на рассылку переменных `N_part`. В данном случае некоторые сходства будут с рассылкой `M_part`.

```
if rank == 0:
    b = empty(M, dtype=float64)
    f3 = open('bData.dat', 'r')
    for j in range(M):
        b[j] = float64(f3.readline())
    f3.close()
else:
    b = None

b_part = empty(M_part, dtype=float64)

if rank in range(0, numprocs, num_col):
    comm_col.Scatterv([b, rcounts_M, displs_M, MPI.DOUBLE],
                      [b_part, M_part, MPI.DOUBLE], root=0)
```

Запомним, что вектор b хранится по частям только на одном из коммуникаторов `comm_col`.

Дальше задаём начальное приближение для вектора x и рассылаем его одному из коммуникаторов `comm_row`.

```
if rank == 0:
    x = zeros(N, dtype=float64)
else:
    x = None

x_part = empty(N_part, dtype=float64)

if rank in range(num_col):
    comm_row.Scatterv([x, rcounts_N, displs_N, MPI.DOUBLE],
                      [x_part, N_part, MPI.DOUBLE], root=0)
```

Вызываем функцию для подсчёта результата. В момент вызова функции блоки матрицы `A_part` хранятся на всех процессах, части вектора `b_part` – только на коммуникаторе `comm_col`, в который входит нулевой процесс, части вектора `x_part` – только на коммуникаторе `comm_row`, в который входит нулевой процесс.

```
x_part = conjugate_gradient_method(A_part, b_part, x_part,
                                   N_part, M_part, N, comm,
                                   comm_row, comm_col, rank)
```

После того, как функция отработает и вернёт часть вектора, его можно собрать целиком на нулевом процессе.

```
106 if rank in range(num_col):
107     comm_row.Gatherv([x_part, N_part, MPI.DOUBLE],
108                      [x, rcounts_N, displs_N, MPI.DOUBLE], root=0)
```

Потом можно сохранить результат и построить график.

Переходим к рассмотрению функции, реализующей метод сопряжённых градиентов. Начнём с аллокации памяти под необходимые массивы.

```
8  def conjugate_gradient_method(A_part, b_part, x_part,
9                                N_part, M_part, N, comm,
10                               comm_row, comm_col, rank):
11     r_part = empty(N_part, dtype=float64)
12     q_part = empty(N_part, dtype=float64)
13
14     if rank in range(num_col):
15         ScalP = array(0, dtype=float64)
16         ScalP_temp = empty(1, dtype=float64)
17
18     s = 1
19     p_part = zeros(N_part, dtype=float64)
```

При разборе цикла будем перед блоками кода приводить соответсвующие математические операции метода сопряжённых градиентов (1.1).

```
20     while s <= N:
```

$$r^{(s)} = \begin{cases} A^T\left(Ax^{(s)} - b\right), & \text{при} \quad s = 1 \\ r^{(s-1)} - \dfrac{q^{(s-1)}}{\left(p^{(s-1)}, q^{(s-1)}\right)}, & \text{при} \quad s \geqslant 2 \end{cases}$$

Так как `x_part` хранятся только на одном из коммуникаторов `comm_row`, разошлём его по всем процессам.

```
21         if s == 1:
22             comm_col.Bcast([x_part, N_part, MPI.DOUBLE], root=0)
```

Вычисляем произведение блока матрицы на часть вектора. Временные массивы `Ax_part` можно было бы создавать только на процессах одного из коммуникаторов `comm_col` (в который входит исходный нулевой процесс).

```
23             Ax_part_temp = dot(A_part, x_part)
24             Ax_part = empty(M_part, dtype=float64)
```

```
25              comm_row.Reduce([Ax_part_temp, M_part, MPI.DOUBLE],
26                              [Ax_part, M_part, MPI.DOUBLE],
27                              op=MPI.SUM, root=0)
28
29              if rank in range(0, numprocs, num_col):
30                  b_part = Ax_part - b_part
31
32              comm_row.Bcast([b_part, M_part, MPI.DOUBLE], root=0)
```

Вычисляем произведение блока транспонированной матрицы на вектор и делаем `Reduce` вдоль столбцов.

```
33              r_part_temp = dot(A_part.T, b_part)
34              comm_col.Reduce([r_part_temp, N_part, MPI.DOUBLE],
35                              [r_part, N_part, MPI.DOUBLE],
36                              op=MPI.SUM, root=0)
```

Получаем вектор r, который хранится по частям в первой строке матрицы процессов (рис. 5.3), т.е. в коммуникаторе `comm_row`, в который входит исходный нулевой процесс.

На последующих итерациях нам необходимо вычислить скалярное произведение векторов p и q. Так как указанные векторы распределены на процессах по частям (в первой строке матрицы процессов), скалярное произведение вычисляется параллельно.

```
37          else: # s != 1
38              if rank in range(num_col):
39                  ScalP_temp[0] = dot(p_part, q_part)
40                  comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
41                                     [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
42                  r_part -= q_part / ScalP
```

Обратите внимание, что нигде не было операций с векторами длины N.

$$p^{(s)} = p^{(s-1)} + \frac{r^{(s)}}{\left(r^{(s)}, r^{(s)}\right)}$$

```
43          if rank in range(num_col):
44              ScalP_temp[0] = dot(r_part, r_part)
45              comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
46                                 [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
47              p_part += r_part / ScalP
```

$$q^{(s)} = A^T\left(Ap^{(s)}\right)$$

Очень похожая операция уже была на первой итерации алгоритма. Отличие тут только в том, что не нужно вычитать часть вектора b, поэтому `Reduce` и `Bcast` вокруг операции вычитания мы заменяем на `Allreduce`.

```
48          comm_col.Bcast([p_part, N_part, MPI.DOUBLE], root=0)
49          Ap_part_temp = dot(A_part, p_part)
50          Ap_part = empty(M_part, dtype=float64)
51          comm_row.Allreduce([Ap_part_temp, M_part, MPI.DOUBLE],
52                             [Ap_part, M_part, MPI.DOUBLE], op=MPI.SUM)
53          q_part_temp = dot(A_part.T, Ap_part)
54          comm_col.Reduce([q_part_temp, N_part, MPI.DOUBLE],
55                          [q_part, N_part, MPI.DOUBLE],
56                          op=MPI.SUM, root=0)
```

$$x^{(s+1)} = x^{(s)} - \frac{p^{(s)}}{\left(p^{(s)}, q^{(s)}\right)}$$

```
57          if rank in range(num_col):
58              ScalP_temp[0] = dot(p_part, q_part)
59              comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
60                                 [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
61              x_part -= p_part / ScalP
```

После этого переходим на следующую итерацию и продолжаем. В конце функции возвращаем значение `x_part`.

```
62          s += 1
63
64      return x_part
```

Мы полностью реализовали другую версию алгоритма метода сопряжённых градиентов. Преимущество в данном случае в том, что все процессы с умножением матрицы ускоряются в *numprocs* раз (как и в программах на лекции 3), операции скалярного умножения ускоряются в $\sqrt{numprocs}$ раз, и, главным образом, пересылаемые сообщения по объёму пропорциональны $\frac{1}{\sqrt{numprocs}}$ при увеличении числа процессов.

Так как в новом алгоритме есть симметрия $N \leftrightarrow M$, новый алгоритм будет хорошо работать при $N \sim M$. Если проверить эту программу на эффективность (как в лекции 4), то для последнего примера (где $N \ll M$) новый алгоритм будет работать хуже. На это можно ответить только то, что нет идеальных алгоритмов, для каждого алгоритма есть своя область применимости.

Лекция 6. Виртуальные топологии

Декартовы топологии.

Рассмотрим инструмент повышения эффективности программной реализации параллельного алгоритма который создает виртуальные топологии. На прошлых занятиях с помощью коммуникатора MPI_COMM_WORLD мы выделяли группы процессов с помощью которых мы организовывали прием и передачу сообщений. Т.е. мы костенным образом определяли логические связи между виртуальными процессорами, однако в реальности каждый MPI процесс выполняется на своем физичесом узле и фактически два связанных MPI процесса в логической среде, могли находится в физической среде довольно далеко друг от друга и соответствующее сообщение может проходить через большое число промежуточных узлов на что тратится лишнее время. Виртуальные топологии позволят нам более простым образом организовать и определить логические связи между процессорами, что зачастую позволяет упростить программную реализацию параллельных алгоритмов, и виртуальные топологии иногда позволяют повысить эффективность програмной реализации. Если системе удается достаточно хорошо отобразить виртуальную топологию на реальную физическую топологию системы, то логически связанные между собой процессоры могут быть и физически расположены рядом, что ускорит передачу сообщений.

Базовые функции для работы с декартовой топологией.

Рассмотрим простейший пример работы с декартовой топологией:

```
1   from mpi4py import MPI
2   from numpy import array, int32
3
4   comm = MPI.COMM_WORLD
5   numprocs = comm.Get_size()
6   rank = comm.Get_rank()
7
8   num_row = 2; num_col = 4
9
10  comm_cart = comm.Create_cart(dims=(num_row, num_col),
11  periods=(True, True), reorder=True)
12
```

```
13      rank_cart = comm_cart.Get_rank()
14
15      neighbour_up, neighbour_down = comm_cart.Shift(direction=0, disp=1)
16      neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
17
```

Рассмотрим подробно новую функцию $Create\_cart$, и рассмотрим ее аргументы. Аргумент $dims$ отвечает за размерность топологии, аргументов может быть много, т.к. передано два аргумента, следовательно у нас задается двумерная сетка процессов. Далее рассмотрим аргумент $periods$, он отвечает за периодичность топологии. Так как задано $True$, следовательно у последнего процесса соседним является также нулевой. Также после создания кортежа у процессов есть свои индификаторы и ранги, отличные от исходных, за что отвечает последний параметр reorder. Если же выставить $False$, то процессоры будут пронумерованы, как в исходной программе, что достаточно плохо, так как физически они могут располагаться достаточно хаотично. При текущих параметрах можно сказать что наша топология имеет вид тора. И с помощью следующей новой функции $comm\_cart.Get\_rank()$ мы получаем номера процессов в нашей топологии. Следующая полезная функция $Shift$, определяющая для локального процесса по $disp$ соседей вдоль соответсвующего направления.

Знакомство с функциями взаимодействия между отдельными процессами $Sendrecv$ и $Sendrecv\_replace$.

Дополним программу следующим образом:

```
18 a = array([(rank_cart % num_col + 1 + i)*2**(rank_cart // num_col)
19 for i in range(2)], dtype=int32)
20     summa = a.copy()
21 for n in range(num_col-1) :
22     comm_cart.Sendrecv_replace([a, 2, MPI.INT],
23         dest=neighbour_right, sendtag=0,
24         source=neighbour_left, recvtag=MPI.ANY_TAG,
25         status=None)
26     summa = summa + a
27 print('Process {} has summa={}'.format(rank_cart, summa))
28
```

Одна из основных операции в наших алгоритмах пересылка результатов вычислений различными процессами. Рассматриваемая функция складывает все элементы во ввсех строках матрицы a. В данной программе сквозным образом идет переменная суммы. Каждый процессор увеличивает ее на определенную величину и передает дальше. Функция

$Sendrecv\_replace$ помогает избежать так называемого $Dedlock$'а, она обращается к области памяти где лежит соответсвующий массив и передается правому соседу, а от левого соседа соответствующее сообщение получается и записывается в тоже место в памяти. И данную команду выполняют все процессоры нашего коммуникатора, соответственно после прохождения по всем процессам на них будут находиться суммы. Фактически это аналог $Allreduce$, оптимизированную под нашу топологию. Аналогичную программу можно реализовать для суммирования по строкам.

Применение декартовой топологии типа двумерного тора для оптимизации взаимодействия процессов в параллельной версии программы, реализующей решение СЛАУ с помощью метода сопряжённых градиентов.

Вернемся к алгоритму рассмотренному в Лекции 5 и усовершенствуем его. Перепишем программу используя фукнции коллективного взаимодействия для новой топологии. Подробно рассмотрим изменения в программе:

```
148             ...
149
150 if rank == 0 :
151     b = empty(M, dtype=float64)
152     f3 = open('bData.dat', 'r')
153     for j in range(M) :
154         b[j] = float64(f3.readline())
155     f3.close()
156 else :
157     b = None
158
159 b_part = empty(M_part, dtype=float64)
160
161 if rank in range(0, numprocs, num_col) :
162     comm_col.Scatterv([b, rcounts_M, displs_M, MPI.DOUBLE],
163         [b_part, M_part, MPI.DOUBLE], root=0)
164
165
166
167 if rank == 0 :
168     x = zeros(N, dtype=float64)
169 else :
170     x = None
171
172 x_part = empty(N_part, dtype=float64)
173
174 if rank in range(num_col) :
175     comm_row.Scatterv([x, rcounts_N, displs_N, MPI.DOUBLE],
```

```
176             [x_part, N_part, MPI.DOUBLE], root=0)
177 comm_col.Bcast([x_part, N_part, MPI.DOUBLE], root=0)
178 x_part = conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
179     N, comm, comm_row, comm_col, rank)
180
181 if rank in range(num_col) :
182     comm_row.Gatherv([x_part, N_part, MPI.DOUBLE],
183         [x, rcounts_N, displs_N, MPI.DOUBLE], root=0)
184
185 if rank == 0 :
186     style.use('dark_background')
187     fig = figure()
188     ax = axes(xlim=(0, N), ylim=(-1.5, 1.5))
189     ax.set_xlabel('i'); ax.set_ylabel('x[i]')
190     ax.plot(arange(N), x, '-y', lw=3)
191     show()
```

Мы считаем, что на всех процессорах коммуниктора матрица разбита по блокам. Нулевой процесс считывает правую часть уравнения $Ax = b$. Далее подготавливается место в памяти под вектор b, и затем распределяем этот массив по процессам. Аналогично с вектором x, теперь эти кусочки передадим все кусочки и остальным процессам с помощью команды $Bcast$. Т.е. теперь вектора b и x на всех процессах. Затем вызываем метод сопряженных градиентов с учетом того, что теперь вектора на всех процессах. Теперь рассмотрим какие изменения будут в функции реализующей метод сопряженных градиентов.

$r^{(s)}$, $p^{(s)}$, $q^{(s)}$ — вспомогательные вектора,

$p^{(0)} = 0$,

$x^{(1)}$ — начальное приближение,

$$r^{(s)} = \begin{cases} A^T\left(A\,x^{(s)} - b\right), & s = 1, \\ r^{(s-1)} - q^{(s-1)} \big/ \left(p^{(s-1)}, q^{(s-1)}\right), \end{cases}$$

$$p^{(s)} = p^{(s-1)} + r^{(s)} \big/ \left(r^{(s)}, r^{(s)}\right),$$

$$q^{(s)} = A^T\left(A\,p^{(s)}\right),$$

$$x^{(s+1)} = x^{(s)} - p^{(s)} \big/ \left(p^{(s)}, q^{(s)}\right),$$

$x^{(N)}$ — решение СЛАУ

```
9  def conjugate_gradient_method(A_part, b_part, x_part, N_part,
10         M_part,N, comm, comm_row, comm_col, rank) :
11
12     r_part = empty(N_part, dtype=float64)
13     p_part = empty(N_part, dtype=float64)
14     q_part = empty(N_part, dtype=float64)
15
```

```
16      ScalP = array(0, dtype=float64)
17      ScalP_temp = empty(1, dtype=float64)
18
19      s = 1
20
21      p_part = zeros(N_part, dtype=float64)
22
23      while s <= N :
24          if s == 1 :
25              Ax_part_temp = dot(A_part, x_part)
26              Ax_part = empty(M_part, dtype=float64)
27              comm_row.Allreduce([Ax_part_temp, M_part, MPI.DOUBLE],
28              [Ax_part, M_part, MPI.DOUBLE], op=MPI.SUM)
29              b_part = Ax_part - b_part
30              r_part_temp = dot(A_part.T, b_part)
31              comm_col.Allreduce([r_part_temp, N_part, MPI.DOUBLE],
32              [r_part, N_part, MPI.DOUBLE], op=MPI.SUM)
33          else :
34              ScalP_temp[0] = dot(p_part, q_part)
35              comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
36              [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
37              r_part = r_part - q_part/ScalP
38      return x_part
39
```

Теперь место в памяти организуется на всех процессах матрицы процессов. $Reduce$ заменили на $Allreduce$ и результат умножения матрицы A на вектор x теперь на всех процессах нашего коммуникатора. Также убрали соответствующий if и $Bcast$. И теперь скалярное произведение рассчитывается не в нулевой строчке, а на всех процессорах. Это не повлияет на скорость программы, т.к. раньше процессы которые не выполняли скалярное произведение простаивали, а теперь они выполняют такие же действия. По сути строка 30 это последовательная часть в нашей программе, так как выполняемые там действия выполняются на всех процессах. Операция вычисления скалярного произведения:

$$p^{(s)} = p^{(s-1)} + r^{(s)} / \left(r^{(s)}, r^{(s)}\right).$$

Фактически осталась без изменений:

```
40  ScalP_temp[0] = dot(r_part, r_part)
41  comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
42      [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
43  p_part = p_part + r_part/ScalP
44
```

Скалярные произведения считаются теперь всюду. Следующая операция:

$$q^{(s)} = A^T \left(A\, p^{(s)}\right)$$

```
45 Ap_part_temp = dot(A_part, p_part)
46 Ap_part = empty(M_part, dtype=float64)
47 comm_row.Allreduce([Ap_part_temp, M_part, MPI.DOUBLE],
48     [Ap_part, M_part, MPI.DOUBLE], op=MPI.SUM)
49 q_part_temp = dot(A_part.T, Ap_part)
50 comm_col.Allreduce([q_part_temp, N_part, MPI.DOUBLE],
51     [q_part, N_part, MPI.DOUBLE], op=MPI.SUM)
```

Последний $Reduce$ был заменен на $Allreduce$. И последняя операция:

$$x^{(s+1)} = x^{(s)} - p^{(s)} / \left(p^{(s)}, q^{(s)}\right)$$

```
52     ScalP_temp[0] = dot(p_part, q_part)
53     comm_row.Allreduce([ScalP_temp, 1, MPI.DOUBLE],
54     [ScalP, 1, MPI.DOUBLE], op=MPI.SUM)
55     x_part = x_part - p_part/ScalP
```

Теперь выполняется на всех процессах коммуникатора.
Таким образом пример 5.1 был изменен таким образом, чтобы остались только удобные для нас функции коллективного взаимодействия процессов. Теперь модернезируем это программу, чтобы использовались виртуальные топологии:

```
1     from mpi4py import MPI
2     from numpy import empty, array, zeros, int32, float64, arange, dot,
    sqrt, hstack
3     from matplotlib.pyplot import style, figure, axes, show
4
5     comm = MPI.COMM_WORLD
6     numprocs = comm.Get_size()
7     rank = comm.Get_rank()
8
9     num_row = num_col = int32(sqrt(numprocs))
10
11    comm_cart = comm.Create_cart(dims=(num_row, num_col),
12    periods=(True, True), reorder=True)
13
14    rank_cart = comm_cart.Get_rank()
15
16    def conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
17        N, comm_cart, num_row, num_col) :
```

Замечание: определили число столбцов и строк нашей топологии как корень из числа процессов, таким образом наша программа запустится только на числе процессов равному квадрату какого-либо натурального числа. Эту строчку нужно продумывать в зависимости от физической топологии вычислительной машины, на которой предстоит работать.

Вомзожная ошибка: определении функции создающей топологии в методе сопряженных градиентов и если $reorder = True$, наши данные считанные ранее данные могут быть перемешаны. Если же поставить параметр $False$, то ускорения не будет. К методу сопря-

```
100     if rank_cart == 0 :
101         f1 = open('in.dat', 'r')
102         N = array(int32(f1.readline()))
103         M = array(int32(f1.readline()))
104         f1.close()
105     else :
106         N = array(0, dtype=int32)
107 
108     comm_cart.Bcast([N, 1, MPI.INT], root=0)
109 
```

Теперь вместо обращения к рангу процесса обращаемся к рангу в нашей декартовой топологии. Вспомогаетльная функция $auxiliary_arrays_determinant$ остается без изменений.

```
125 rank_cart == 0 :
126 unts_M, displs_M = auxiliary_arrays_determination(M, num_row)
127 unts_N, displs_N = auxiliary_arrays_determination(N, num_col)
128 e :
129 unts_M = None; displs_M = None
130 unts_N = None; displs_N = None
131 
132 art = array(0, dtype=int32); N_part = array(0, dtype=int32)
133 
134 m_col = comm_cart.Split(rank_cart % num_col, rank_cart)
135 m_row = comm_cart.Split(rank_cart // num_col, rank_cart)
136 
137 rank_cart in range(num_col) :
138 m_row.Scatter([rcounts_N, 1, MPI.INT],
139 part, 1, MPI.INT], root=0)
140 rank_cart in range(0, numprocs, num_col) :
141 m_col.Scatter([rcounts_M, 1, MPI.INT],
142 part, 1, MPI.INT], root=0)
143 
144 m_col.Bcast([N_part, 1, MPI.INT], root=0)
145 m_row.Bcast([M_part, 1, MPI.INT], root=0)
146 
147 art = empty((M_part, N_part), dtype=float64)
148 
149 up = comm_cart.Get_group()
```

Разделяем наш коммуникатор на вспомогательные, на которые будет разделена матрица системы, только теперь используется коммуникатор декартовой топологии. С учетом этого и если система удачно отображена на физическую топологию, то эта функция бу-

дет работать быстрее. Часть кода считывающая матрицу A и вектора b и x осталась без изменений, кроме замены просто ранга на ранг топологии, и распределения по первому столбцу виртуальной топологии и передачи остальным, что обсуждалось в примере 6.1. Вернемся к функции реализующей метод сопряженных градиентов:

```
16 def conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
17         N, comm_cart, num_row, num_col) :
18 
19 neighbour_up, neighbour_down = comm_cart.Shift(direction=0, disp=1)
20 neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
21 
22 ScalP_temp = empty(1, dtype=float64)
23 
24 s = 1
25 
26 p_part = zeros(N_part, dtype=float64)
27 
28 while s <= N :
29     if s == 1 :
30         Ax_part_temp = dot(A_part, x_part)
31         Ax_part = Ax_part_temp.copy()
32         for n in range(num_col-1) :
33             comm_cart.Sendrecv_replace([Ax_part_temp, M_part, MPI.DOUBLE],
34                 dest=neighbour_right, sendtag=0,
35                 source=neighbour_left, recvtag=MPI.ANY_TAG,
36                 status=None)
37             Ax_part = Ax_part + Ax_part_temp
38             b_part = Ax_part - b_part
39             r_part_temp = dot(A_part.T, b_part)
40             r_part = r_part_temp.copy()
41         for m in range(num_row-1) :
42             comm_cart.Sendrecv_replace([r_part_temp, N_part, MPI.DOUBLE],
43                 dest=neighbour_down, sendtag=0,
44                 source=neighbour_up, recvtag=MPI.ANY_TAG,
45                 status=None)
46             r_part = r_part + r_part_temp
47     else :
48         ScalP_temp[0] = dot(p_part, q_part)
49         ScalP = ScalP_temp.copy()
50         for n in range(num_col-1) :
51             comm_cart.Sendrecv_replace([ScalP_temp, 1, MPI.DOUBLE],
52                 dest=neighbour_right, sendtag=0,
53                 source=neighbour_left, recvtag=MPI.ANY_TAG,
54                 status=None)
55             ScalP = ScalP + ScalP_temp
56         r_part = r_part - q_part/ScalP
57 
```

Теперь каждый MPI процесс определяет своих соседей, теперь необходимо выделять место в памяти только для одного вспомогательного массива p_part. Каждый процесс копирует промежуточный кусочек и с помощью команды $Sendrecv\_replace$ передает соответствующий вектор своему правому соседу,а от левого соседа получает и записывает в соответствующий вектор. На каждом процессе будет Ax_part, который является результатом суммы всех массивов, т.е. мы реализовали функцию $Allreduce$, и в случае удачной топологии это повысит скорость программы. Далее производятся вычисления массивов r_part_temp и т.д. Затем снова вдоль каждого столбца, мы складываем результаты, после чего они снова собираются на всех процессах. В случае следующих операции в аналогичном стиле реализуется скалярное произведение. Полностью аналогично выполняем вычисления векторов $p^{(s)}$, $q^{(s)}$ и $x^{(s+1)}$.

Мы рассмотрели построение виртуальных топологии, которые позволяют в лучшем виде отобразить логические связи алгоритма на физическую топологии. Хотя это и менее удобно, однако это повышает эффективность программы. Они используются только в том случае когда это может повысить эффективность программной реализации, за счет понижения расходов на прием-передачу сообщений. Если хорошего отображения нет, то и преимущества мы не получаем, однако если такое отображение есть мы можем получить значительное приемущество во времени вычислений. Очень важно не забывать адаптировать программу при переходе с одной физической топологии на другую, так как это может привести к излишнему повышению времени работы.

Лекция 7. Параллельный вариант метода прогонки для решения СЛАУ с трехдиагональной матрицей.

Последовательный алгоритм.

На предыдующих лекциях были рассмотрены базовые подходы к распараллеливаню алгоритмов, которые используются при операциях с плотно заполненными матрицами. На следующей лекции будут рассмотреные особенности численной реализации задач для уравнений в частных производных. И одной из особенностей таких задач является то, что при их решении часто возникает необходимость с оперированием с разреженными матрицами. Под разреженной матрицей мы подразумеваем такую матрицу, в которой большинство элементов нули, а не нулевые элементы расположены определенным, заранее известным образом. И это позволяет построить более эффективный алгоритм численного решения. В частности при решении задач для уравнений в частных производных, одномерных по пространству, часто возникает необходимость решения задач с трёхдиагональной матрицей. В который ненулевые элементы расположены на главной диагонали и двух побочных кодиагоналях. Базовый алгоритм с трёхдиагональной матрицей нельзя распараллелить, однако его модифицированную версию распараллелить можно. Рассмотрим систему:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \dots \\ d_{n-1} \\ d_n \end{pmatrix}$$

Мы можем применить алгоритмы рассмотренные ранее, однако это не коструктивно в силу того, что те алгоритмы не учитывают спицифическую структуру, и их сложность $O(N^3)$. С учетом нашего знания о положении не нулевых элементов мы сможем построить алгоритм требующий $O(N)$ операций. Рассмотрим самый интуитивно понятный алгоритм: с помощью прямого хода обнулям сначала нижнюю диагональ, а потом обратным ходом обнулим верхнюю диагональ. После этого матрица системы будет иметь диагональный вид, и далее вычисляем неизвестные делением соответвутсвующее число правой части делить на коэффициент стоящий при неизвестном. Реализация решения с помощью трехдиагональной матрицей очень простая:

```
1 def consecutive_algorithm(a, b, c, d) :
2
```

```
N = len(d)

for n in range(1, N) :
    coef = a[n]/b[n-1]
    b[n] = b[n] - coef*c[n-1]
    d[n] = d[n] - coef*d[n-1]


return x
```

Замечания: количество операций в продеманстрированном алгоритме равно $9N-3$, однако формулы можно переписать в другом виде и решение будет найдено за $8N$ операций, этот алгоритм рассматривался в курсе численные методы во 2 лекции. Главная проблема, что алгоритм в принципе не распараллеливается, т.к. есть информационные зависимости. Если каждому процессу давать задание вычислять свои b_n и d_n, но это сделать невозможно, т.к. требуется информация с предыдующего шага(b_{n-1}). Поэтому наша задача изменить алгоритм таким образом, чтобы его можно было распраллелить, однако в этом случае вычислительная сложность возрастет. Распараллеливание мы подразумеваем на произвольное число процессов. Конечно, существует и вариант распраллеливания именно этих вычислений: мы идем одновременно по нижней кодиагонали вниз с первого элемента, а по верхней кодиагонали вверх с нижнего элемента, в середине матрицы процессы обмениваются информацией и двигаются дальше, однако этот алгоритм можно использовать только для двух процессов, что дает сравнимо малый прирост в скорости. В идеале мы хотим получать ускорение программы пропорциональное числу работающих процессов.

Параллельный алгоритм.

Вернемся к нашей системе:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \dots \\ d_{n-1} \\ d_n \end{pmatrix}$$

Для простоты возьмем $n = 9$, и распределим по трём процессам следующим образом:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots \\ 0 & a_3 & b_3 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & c_6 & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} \dots & a_7 & b_7 & c_7 & 0 \\ 0 & \dots & a_8 & b_8 & c_8 \\ 0 & \dots & \dots & a_9 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

У каждого процесса теперь будут не три вектора длины N, а три вектора разделенные на число процессов. Нулевой процесс может обнулить нижнюю кодиагональ может, а вот остальные этого уже не смогут сделать. Шаг 1. На каждом процессе мы будем обнулять нижнюю кодагональ не с нулевого элемента на этом процессоре, а с первого. Для этого на каждом процессе из первой строки вычитаем нулевую с соответсвующим коэффициентом. После полного прохождения всеми процессорами получаем (красным цветом отмечены измененные в этом шаге элементы):

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & \dots & \dots \\ 0 & 0 & b_3 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & c_6 & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & c_7 & 0 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

Шаг 2. Далее мы обнуляем в каждом блоке обнуляем элементы выше главной диагонали, обнуляем элементы лежащие не в предпоследней строчке, а в предпредпоследней, что получаем в результате(зеленым цветом выделены элементы измененные на этой итерации):

$$\begin{pmatrix} b_1 & 0 & c_1 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & \dots & \dots & \\ 0 & 0 & b_3 & c_3 & \dots & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & 0 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & c_6 & \dots & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & 0 & c_7 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

Шаг 3. Теперь мы хотим обнулить элементы c_3 и c_6, проблема заключается в том, что теперь из строчки находящейся на нулевом процессе необходимо передать информацию с первого процесса. Следовательно нам нужно обменяться этими данными. И после этого обмена мы можем провести эту операцию(синим цветом обозначены изменения):

$$\begin{pmatrix} b_1 & 0 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & 0 & \dots & \dots & \\ 0 & 0 & b_3 & 0 & 0 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & 0 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & 0 & \dots & \dots & c_6 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & 0 & c_7 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

Шаг 4. После этих действий, которые повлекли обмен маленькими сообщениями между соседними процессами мы возьмем элементы находящиеся в последней строке блока данных каждого процесса и собираем их на нулевом процессе. В результате мы видим следующую небольшую маленькую систему, которую придется решить последовательно:

$$\begin{pmatrix} b_3 & c_3 & 0 \\ a_6 & b_6 & c_6 \\ 0 & a_9 & b_9 \end{pmatrix} \begin{pmatrix} x_3 \\ x_6 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_3 \\ d_6 \\ d_9 \end{pmatrix}$$

Часть системы мы решаем последовательно, но если N очень большое, и мы решаем систему на сотне процессов, то итоговая матрица матрица решенная последовательно будет давать относительно большой выйгрыш во времени. К обсуждению этого мы вернемся в последней теме этой лекции. Далее соответствующий элементы решения передаем процессам. И в результате простых математических операций находим решением например: $x_5 = \dfrac{d_5 - a_5 x_4 - c_5 x_8}{b_5}$. В итоге все сводится к последовательным алгоритмам решаемым на соответствующих процессах.

Особенности программной реализации.

Обсудим тонкости программной реализации алгоритма:

```
1 def parallel_tridiagonal_matrix_algorithm(a_part,
2     b_part, c_part, d_part) :
3
4     N_part = len(d_part)
```

```
5
6       for n in range(1, N_part) :
7           coef = a_part[n]/b_part[n-1]
8           a_part[n] = -coef*a_part[n-1]
9           b_part[n] = b_part[n] - coef*c_part[n-1]
10          d_part[n] = d_part[n] - coef*d_part[n-1]
11
12      for n in range(N_part-3, -1, -1):
13          coef = c_part[n]/b_part[n+1]
14          c_part[n] = -coef*c_part[n+1]
15          a_part[n] = a_part[n] - coef*a_part[n+1]
16          d_part[n] = d_part[n] - coef*d_part[n+1]
17
18      if rank > 0 :
19          temp_array_send = array([a_part[0], b_part[0],
20          c_part[0], d_part[0]],dtype=float64)
21      if rank < numprocs-1 :
22          temp_array_recv = empty(4, dtype=float64)
```

Программа подразумевает, что вектора распределены по всем процессам. Для упрощения программной реализации создаем все векторы одинаковой длинны, т.е. например на первом процессе у нас будет существовать один дополнительный элемент, который никак не будет влиять на итоговый результат. Также алгоритм адоптирован к неравному распределению строк системы по процессорам. В строках 6-11 реализован прямой ход метода Гаусса (Шаг 1). В строках 12-16 реализован обратный ход метода Гаусса (Шаг 2). Строки 18-22 реализуют подготовку данных к обмену между процессами.

```
23      if rank == 0 :
24          comm.Recv([temp_array_recv, 4, MPI.DOUBLE], source=1,
25              tag=0, status=None)
26      if rank in range(1, numprocs-1) :
27          comm.Sendrecv(sendbuf=[temp_array_send, 4, MPI.DOUBLE],
28              dest=rank-1, sendtag=0,
29              recvbuf=[temp_array_recv, 4, MPI.DOUBLE],
30              source=rank+1, recvtag=MPI.ANY_TAG, status=None)
31      if rank == numprocs-1 :
32          comm.Send([temp_array_send, 4, MPI.DOUBLE],
33              dest=numprocs-2, tag=0)
34
35      if rank < numprocs-1 :
36          coef = c_part[N_part-1]/temp_array_recv[1]
37          b_part[N_part-1] = b_part[N_part-1] - coef*temp_array_recv[0]
38          c_part[N_part-1] = - coef*temp_array_recv[2]
39          d_part[N_part-1] = d_part[N_part-1] - coef*temp_array_recv[3]
40
41      temp_array_send = array([a_part[N_part-1], b_part[N_part-1],
42       c_part[N_part-1], d_part[N_part-1]],dtype=float64)
```

В строках 23-33 реализована передача сообщений между процессами, с помощью функции $Sendrecv$, чтобы избежать «дедлока», и чтобы не изменять данные которые находятся на процессах. Далее в строках 35-39 производится рассчет (Шаг 3). Затем подготавливаем массив на отправку, чтобы в дальнейшем воспользоватся для него методом последовательной прогонки.

```
43      if rank == 0 :
44          A_extended = empty((numprocs, 4), dtype=float64)
45      else :
46          A_extended = None
47
48      comm.Gather([temp_array_send, 4, MPI.DOUBLE],
49          [A_extended, 4, MPI.DOUBLE], root=0)
50
51      if rank == 0:
52          x_temp = consecutive_tridiagonal_matrix_algorithm(
53              A_extended[:,0], A_extended[:,1], A_extended[:,2],
54              A_extended[:,3])
55      else :
56          x_temp = None
57
58      if rank == 0 :
59          rcounts_temp = empty(numprocs, dtype=int32)
60          displs_temp = empty(numprocs, dtype=int32)
61          rcounts_temp[0] = 1
62          displs_temp[0] = 0
63          for k in range(1, numprocs) :
64              rcounts_temp[k] = 2
65              displs_temp[k] = k - 1
66      else :
67          rcounts_temp = None; displs_temp = None
68
69      if rank == 0 :
70          x_part_last = empty(1, dtype=float64)
71          comm.Scatterv([x_temp, rcounts_temp, displs_temp, MPI.DOUBLE],
72              [x_part_last, 1, MPI.DOUBLE], root=0)
73      else :
74          x_part_last = empty(2, dtype=float64)
75          comm.Scatterv([x_temp, rcounts_temp, displs_temp, MPI.DOUBLE],
76              [x_part_last, 2, MPI.DOUBLE], root=0)
```

Затем собираем элементы для решения последовательной прогонки в матрицу A. И вызываем последовательный метод решения системы. В строках 58-67 происходит подготовка вспомогательных временных массивов $rcounts_temp$ и $displs_temp$. С помощью которых происходит распределение элементов x_i по процессам, нулевому и последнему процессу отправляется по одному элементу, остальным по два элемента. И затем мы распределяем

вектор по процессам.

```
77 x_part = empty(N_part, dtype=float64)
78
79 if rank == 0 :
80     for n in range(N_part-1) :
81         x_part[n] = (d_part[n] - c_part[n]*x_part_last[0])/b_part[n]
82     x_part[N_part-1] = x_part_last[0]
83 else :
84     for n in range(N_part-1) :
85     x_part[n] = (d_part[n] - a_part[n]*x_part_last[0]
86         - c_part[n]*x_part_last[1])/b_part[n]
87 x_part[N_part-1] = x_part_last[1]
88
89 return x_part
```

И в результате мы находим вектор x.

К вопросу об эффективности распараллеливания.

Вернемся к вопросу эффективности распараллеливания алгоритма. Сделаем оценку предельно возможного ускорения для именно нашей программы, реализующей параллельный вариант метода прогонки. N - размерность нашей системы, n - число процессов для которых запущена программа.

Прямой ход : $(N-1)\cdot 6$.

Обратный ход : $(N-2)\cdot 6$.

После обмена данными : $(n-1)\cdot 6$.

Последовательная прогонка : $9n-3$ ($8n$).

Вычисление решения: $(N-1)\cdot 5+n-2$.

В числитель записываем распараллеливаемое число операций, а в знаменатель распараллеливаемую часть плюс операции, которые невозможно распараллелить : $S_n \leqslant \dfrac{17N+16n-34}{(9n-3)+\dfrac{17N+7n-31}{n}}$,

$$S_n \leqslant \frac{17 + \dfrac{16n}{N} - \dfrac{34}{N}}{\dfrac{9n}{N} + \dfrac{4}{N} + \dfrac{17}{n} - \dfrac{31}{nN}}.$$

$S_n \leqslant n$ при условии $n \ll N$. Следовательно в пределе мы получаем линейное ускорение, и последовательной частью программы мы можем принебречь.

Лекция 8. Подходы к распараллеливанию алгоритмов решения задач для уравнений в частных производных.

На этой лекции мы продолжаем рассматривать различные подходы к распараллеливанию алгоритмов, которые используются для решения уравнений в частных производных.

В качестве примера рассмотрим следующую задачу параболического типа.

Пример одномерной по пространству начально-краевой задачи для уравнения в частных производных параболического типа

$$\begin{cases} \varepsilon\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial u}{\partial t} = -u\dfrac{\partial u}{\partial x} - u^3, \quad x \in (a,b), \quad t \in (t_0, T], \\ u(a,t) = u_{\mathit{left}}(t), \quad u(b,t) = u_{\mathit{right}}(t), \quad t \in (t_0, T], \\ u(x,t_0) = u_{\mathit{init}}(x), \quad x \in [a,b]. \end{cases} \tag{8.1}$$

Задача ставится в прямоугольнике $(x,t) \in [a,b] \times [t_0,T]$. В этой области нам нужно восстановить неизвестную функцию $u(x,t)$. Естественно, если мы можем найти функцию аналитически, численные методы и параллельные вычисления не нужны. Мы предполагаем, что искомую функцию мы найти не можем.

Эта задача уже встречалась нам в курсе ”Численные методы”на лекции 19. Там мы рассмотрели два различных подхода при решении подобных задач – построение численного решения с помощью явных и неявных схем. На этой лекции мы рассмотрим распараллеливание явной численной схемы, а на следующей лекции обсудим особенности распараллеливания неявной численной схемы. Кратко напомним текущий алгоритм (подробнее его можно посмотреть в лекции 19).

Последовательный алгоритм решения, основанного на реализации явной схемы

Вводим сетку по пространственной и по временной переменной. Если через все узлы мы проведём прямые, параллельные соответсвующим осям, то их пересечения дадут

нам узлы пространственно-временной сетки (см. рис. 8.1). Наша задача – найти приближенные значения функции в узлах сетки (сеточные значения) $u_n^m = u(x_n, t_m)$.

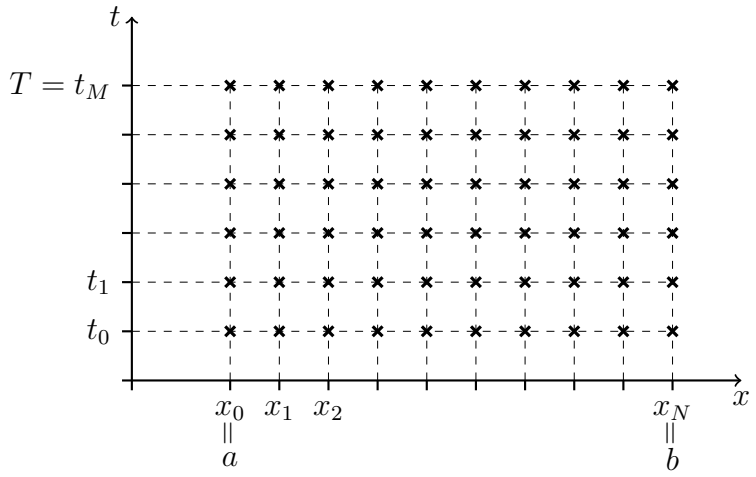


Рис. 8.1: Пространственно-временная сетка для задачи 9.1

Сеточные значения на верхней, нижней и правой границах нам точно известны, мы их можем получить из начальных и граничных условий задачи. Остальные значения можно получить по следующей формуле:

$$u_n^{m+1} = u_n^m + \frac{\varepsilon\tau}{h^2}\left(u_{n+1}^m - 2u_n^m + u_{n-1}^m\right) + \frac{\tau}{2h}u_n^m\left(u_{n+1}^m - u_{n-1}^m\right) + \tau\left(u_n^m\right)^3. \tag{8.2}$$

Она связывает между собой значения в 4 узлах сетки (см. рис. 8.2). Последовательно применяя эту формулу, можно найти, для начала, все сеточные значения для $t = t_1$. Для этого берём указанную формулу с параметрами $m = 0$, $n = 1 \dots N - 1$. Сразу обращаем внимание на то, что можно осуществлять распараллеливание по n.

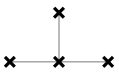


Рис. 8.2: Шаблон для явного алгоритма решения задачи 9.1

В результате мы нашли все значения на временном слое $t = t_1$, и можно искать сеточные значения на следующем временном слое. Будем повторять эти действия, пока не дойдем до последнего слоя $t = t_M$.

Напоминаем, что явные схемы в плане результата – достаточно плохие, но они очень просты для реализации и, в том числе, для распараллеливания. Неявные схемы, распарал-

леливание которых мы рассмотрим на следующей лекции, дают гораздо более хороший результат на сетках меньшей размерности.

Приведём последовательную версию программы, которая реализует этот алгоритм.

Программная реализация последовательного алгоритма решения

Программная реализация взята из курса ”Численные методы пример 19-1.

```
1  from numpy import empty, linspace, sin, pi
2  import time
3
4  def u_init(x):
5      return sin(3*pi*(x-1/6))
6
7  def u_left(x):
8      return -1
9
10 def u_right(x):
11     return +1
```

Время последовательной программы будем замерять с помощью встроенной библиотеки `time`.

```
12 start_time = time.time()
```

Далее определяем параметры задачи:

```
13 a, b = (0, 1)
14 t_0, T = (0, 6)
15 eps = 10**(-1.5)
16
17 N, M = (200, 20_000)
18
19 x, h = linspace(a, b, N+1, retstep=True)      # h = (b - a) / N
20 t, tau = linspace(t_0, T, M+1, retstep=True)  # tau = (T - t_0) / M
```

Задаём массив сеточных значений u_n^m и заполняем уже известные значения:

```
21 u = empty((M+1, N+1))
22
23 for n in range(N+1):
24     u[0, n] = u_init(x[n])
25
26 for m in range(M+1):
27     u[m, 0] = u_left(t[m])
```

```
28      u[m, N] = u_right(t[m])
```

Далее последовательно в двойном цикле вычисляем сеточные значения (по формуле 8.2):

```
29  for m in range(M):
30      for n in range(1, N):
31          d2 = (u[m, n+1] - 2*u[m, n] + u[m, n-1]) / h**2
32          d1 = (u[m, n+1] - u[m, n-1]) / (2*h)
33
34          u[m+1, n] = u[m, n] + eps*tau*d2 + tau*u[m, n]*d1 + tau*u[m, n]**3
```

и выводим затраченное на решение задачи время

```
35  print(f"Elapsed time is {time.time() - start_time:.4f}")
```

Так как предполагается, что данная программа может запускаться как на персональном компьютере, так и на кластере, сохраним данные в файл.

```
36  from numpy import savez
37  savez("results_of_calculations", x=x, u=u)
```

Далее приведём отдельный скрипт для построения графика:

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   from matplotlib.animation import FuncAnimation
5
6   results = np.load("results_of_calculations.npz")
7
8   x = results["x"]
9   u = results["u"]
10
11  M = np.shape(u)[0]
12
13  fig, ax = plt.subplots()
14
15  def animate(i):
16      ax.clear()
17      ax.set_ylim(-2, 2)
18      line, = ax.plot(x, u[i*M//100, :], color = 'blue', lw=1)
19      return line
20
21  anim = FuncAnimation(fig, animate, interval=40, frames=100)
22  anim.save("results.gif", dpi=300)
```

Параллельный алгоритм решения, основанного на реализации явной схемы

На этой лекции мы рассмотрим два подхода распараллеливания этой задачи. Начнём с менее эффективного, чтобы рассмотреть классические ошибки, которые возникают в процессе распараллеливания.

Пусть каждый MPI процесс будет заниматься вычислениями в своей области (см. рис. 8.3).

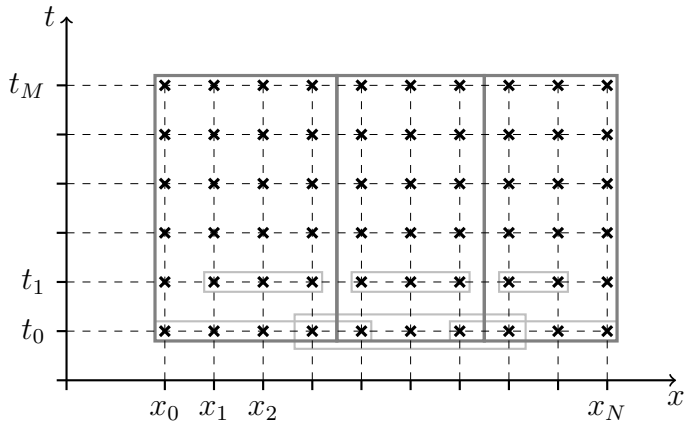


Рис. 8.3: Схема разделения данных по MPI процессам

Изначально все начальные данные будут храниться на нулевом процессе. С учётом шаблона (рис. 8.2) необходимо будет передавать значения на предыдущем временном слое не так, как показано в разделении, а ещё захватывать по одному значению слева и справа. Мы будем реализовывать эту операцию с помощью метода `Scatterv`. Кроме того, нам потребуется аргумент `displs`, так как сдвиги не до конца очевидны.

После того, как каждый процесс проведёт необходимые вычисления, сеточные значения нужно будет собрать на нулевом процессе. На этом этапе массивы `rcounts` и `displs` будет выглядеть уже привычным для нас образом.

Видно, что алгоритм не очень эффективен. Например, можно было бы просто передавать граничные элементы двум соседним процессам. Мы этот подход используем уже во втором варианте распараллеливания.

Программная реализация параллельного алгоритма решения

```
1  from mpi4py import MPI
2  from numpy import empty, array, int32, float64, linspace, sin, pi, hstack
3
4  comm = MPI.COMM_WORLD
5  numprocs = comm.Get_size()
6  rank = comm.Get_rank()
7
8  def u_init(x):
9      return sin(3*pi*(x-1/6))
10
11 def u_left(x):
12     return -1
13
14 def u_right(x):
15     return +1
```

Начинаем отсчёт времени:

```
16 if rank == 0:
17     start_time = MPI.Wtime()
```

Задаём параметры задачи:

```
18 a, b = (0, 1)
19 t_0, T = (0, 6)
20 eps = 10**(-1.5)
21
22 N, M = (800, 300_000)
23
24 x, h = linspace(a, b, N+1, retstep=True)
25 t, tau = linspace(t_0, T, M+1, retstep=True)
```

Почему мы выбираем такое большое `M`? Если вспомнить лекцию 4, можно предположить, что такой выбор обусловлен тем, что мы хотим получить хорошие результаты для эффективности распараллеливания. На самом деле это вызвано тем, что мы используем неэффективную явную схему решения задачи. При выборе существенно меньших значений `M` счёт разваливается. На следующей лекции мы рассмотрим неявные схемы, где качественно-восстановленное решение можно будет получить при значительно меньших значениях `M`.

Дальше задаём стандартные массивы `rcounts` и `displs`, которые отвечают за то, какие значения на каждом шаге алгоритма каждый процесс будет вычислять. Нулевой процесс также участвует в вычислениях. Напоминаем, что, по сути, `N_part = rcounts[rank]`, но при его использовании целиком массив `rcounts` мы не рассылаем.

```
26 if rank == 0:
```

```
27      ave, res = divmod(N+1, numprocs)
28      rcounts = empty(numprocs, dtype=int32)
29      displs = empty(numprocs, dtype=int32)
30
31      for k in range(0, numprocs):
32          if k < res:
33              rcounts[k] = ave + 1
34          else:
35              rcounts[k] = ave
36
37          if k == 0:
38              displs[k] = 0
39          else:
40              displs[k] = displs[k-1] + rcounts[k-1]
41  else:
42      rcounts, displs = None, None
43
44  N_part = array(0, dtype=int32)
45  comm.Scatter([rcounts, 1, MPI.INT], [N_part, 1, MPI.INT], root=0)
```

Также введём другие вспомогательные массивы, которые используются при рассылке данных с нулевого процесса. Для процессов, на которых хранятся средние части сеточных значений, количество пересылаемых значений увеличится на два, для крайних – на одно.

```
46  if rank == 0:
47      rcounts_from_0 = empty(numprocs, dtype=int32)
48      displs_from_0 = empty(numprocs, dtype=int32)
49
50      rcounts_from_0[0] = rcounts[0] + 1
51      displs_from_0[0] = 0
52
53      for k in range(1, numprocs-1):
54          rcounts_from_0[k] = rcounts[k] + 2
55          displs_from_0[k] = displs[k] - 1
56
57      rcounts_from_0[-1] = rcounts[-1] + 1
58      displs_from_0[-1] = displs[-1] - 1
59
60  N_part_aux = array(0, dtype=int32)
61  comm.Scatter([rcounts_from_0, 1, MPI.INT],
62               [N_part_aux, 1, MPI.INT], root=0)
```

На нулевом процессе выделяем память для всех сеточных значений.

```
63  if rank == 0:
64      u = empty((M+1, N+1), dtype=float64)
```

```
65
66      for n in range(N+1):
67          u[0, n] = u_init(x[n])
```

На ненулевых процессах введём пустой массив.

```
68  else: # rank != 0
69      u = empty((M+1, 0), dtype=float64)
```

Дальше на каждом процессе заготавливаем части массива.

```
70  u_part = empty(N_part, dtype=float64)
71  u_part_aux = empty(N_part_aux, dtype=float64)
```

На каждом шаге цикла по времени вначале рассылаем необходимые массивы по процессам

```
72  for m in range(M):
73      comm.Scatterv([u[m], rcounts_from_0, displs_from_0, MPI.DOUBLE],
74                    [u_part_aux, N_part_aux, MPI.DOUBLE], root=0)
75
76      for n in range(1, N_part_aux-1):
77          d2 = (u_part_aux[n+1] - 2*u_part_aux[n] + u_part_aux[n-1]) / h**2
78          d1 = (u_part_aux[n+1] - u_part_aux[n-1]) / (2*h)
79
80          u_part[n-1] = u_part_aux[n] + eps*tau*d2 + \
81                        tau*u_part_aux[n]*d1 + u_part_aux[n]**3
```

Обратите внимание, что в новый массив значение записывается со сдвинутым индексом. Мы так делаем из-за того, что в массиве для нового слоя элементов на 2 (или на 1) меньше. Дальше добавляем в массив граничные условия и собираем вычисленные сеточные значения на нулевом процессе.

```
82      if rank == 0:
83          u_part = hstack((u_left(t[m+1]), u_part[:N_part-1]))
84      elif rank == numprocs-1:
85          u_part = hstack((u_part[:N_part-1], u_right(t[m+1])))
86
87      comm.Gatherv([u_part, N_part, MPI.DOUBLE],
88                   [u[m+1], rcounts, displs, MPI.DOUBLE], root=0)
```

В заключение выведем необходимую техническую информацию и сохраним полученные данные.

```
89  if rank == 0:
90      end_time = MPI.Wtime()
91
```

```
92      print(f"{N=}, {M=}")
93      print(f"Number of MPI processes is {numprocs}")
94      print(f"Elapsed time is {end_time - start_time:.4f} sec.")
95
96      from numpy import savez
97      savez("results_of_calculations", x=x, u=u)
```

Видно, что без функции `Gatherv` нам не обойтись, если мы хотим построить график. Однако функцию `Scatterv` можно бы было заменить на обмен сообщениями лишь между соседними процессами - достаточно обмениваться одним значением с правым процессом и ещё одним значением с левым процессом.

Другой вариант параллельного алгоритма решения

При решении реальных задач значений может быть очень много, так что матрица целиком просто не будет помещаться на нулевой процесс. Будем хранить результаты расчётов распределёно. Каждый процесс будет хранить сеточные значения слоёв по x, где он проводит вычисления (см. рис. 8.3), а также по одному слою от значений соседних процессов.

Также введём виртуальную топологию (см. лекцию 6). Разрешая переупорядочить процессы, мы позволяем системе отобразить виртуальную топологию на физическую наиболее оптимальным образом. Тогда процессы с близкими номерами будут и физически расположены рядом. Это позволит уменьшить время приёма-пересылки сообщений.

```
7  comm_cart = comm.Create_cart(dims=[numprocs], periods=[False],
8                               reorder=True)
9  rank_cart = comm_cart.Get_rank()
```

Здесь программа не очень большая. Лектор тестировал, выигрыш при использовании данной топологии составляет 3-4%.

Дальше при вызове функций `Scatter` не забываем заменить коммуникатор `comm` на `comm_cart` (строки 45, 61), а также номер процесса `rank` на `rank_cart` в строках 16, 26, 46.

Также нам потребуется разослать массив сдвигов `displs_from_0`.

```
63 displs_aux = array(0, dtype=int32)
64 comm_cart.Scatter([displs_from_0, 1, MPI.INT],
65                   [displs_aux, 1, MPI.INT], root=0)
```

Сдвиги нам понадобятся для заполнения сеточных значений по начальному условию.

Далее выделяем память под соответсвующую часть массива сеточных значений на каждом процессе. И заполняем значения на нулевом временном слое.

```
63 u_part_aux = empty((M+1, N_part_aux), dtype=float64)
64
65 for n in range(N_part_aux):
66     u_part_aux[0, n] = u_init(x[displs_aux+n])
```

На нулевом и (numprocs-1)-ом процессе также можно сразу вычислить граничные сеточные значения. В прошлой реализации этого сделать не получалось, так как на каждой итерации на каждом процессе был выделен массив только для заданного временного слоя.

```
63 if rank_cart == 0:
64     for m in range(1, M+1):
65         u_part_aux[m, 0] = u_left(t[m])
66
67 elif rank_cart == numprocs-1:
68     for m in range(1, M+1):
69         u_part_aux[m, -1] = u_right(t[m])
```

Начинаем основную вычислительную часть программы с вычисления сеточных значений на новом слое:

```
63 for m in range(M):
64
65     for n in range(1, N_part_aux-1):
66         d2 = (u_part_aux[m, n+1] - 2*u_part_aux[m, n] +
67                                    u_part_aux[m, n-1]) / h**2
68         d1 = (u_part_aux[m, n+1] - u_part_aux[m, n-1]) / (2*h)
69
70         u_part_aux[m+1, n] = u_part_aux[m, n] + eps*tau*d2 + \
71                              tau*u_part_aux[m, n]*d1 + u_part_aux[m, n]**3
```

Чтобы перейти к следующему шагу, все процессы должны обменяться данными. Первый процесс обменивается лишь одним значением:

```
63     if rank_cart == 0:
64         comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
65                                     1, MPI.DOUBLE],
66                            dest=1, sendtag=0,
67                            recvbuf=[u_part_aux[m+1, N_part_aux-1:],
68                                     1, MPI.DOUBLE],
69                            source=1, recvtag=MPI.ANY_TAG, status=None)
```

Напоминаем, что функция Sendrecv позволяет одновременно вызвать комманды Send и

Recv, при этом не допуская deadlock (взаимной блокировки процессов). Внутри эта функция использует асинхронные операции.

”Средние”процессы посылают и получают по два значения:

```
63      if rank_cart in range(1, numprocs-1):
64          comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1], 1, MPI.DOUBLE],
65                             dest=rank_cart-1, sendtag=0,
66                             recvbuf=[u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
67                             source=rank_cart-1,
68                             recvtag=MPI.ANY_TAG, status=None)
69
70          comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
71                                      1, MPI.DOUBLE],
72                             dest=rank_cart+1, sendtag=0,
73                             recvbuf=[u_part_aux[m+1, N_part_aux-1:],
74                                      1, MPI.DOUBLE],
75                             source=rank_cart+1,
76                             recvtag=MPI.ANY_TAG, status=None)
```

Последний процесс:

```
63      if rank_cart == numprocs-1:
64          comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1], 1, MPI.DOUBLE],
65                             dest=numprocs-2, sendtag=0,
66                             recvbuf=[u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
67                             source=numprocs-2,
68                             recvtag=MPI.ANY_TAG, status=None)
```

Немного обсудим отличия этого подхода с тем, что мы обсуждали вначале лекции. как видно, вычислительная часть не изменилась совсем, изменилась только логика коммуникации процессов. Раньше мы использовали функцию Scatterv. Несмотря на то, что она реализована достаточно эффективно, она все равно остаётся функцией коллективного взаимодействия, то есть часть процессов простаивает в любом случае. В новой же версии все коммуникации происходят параллельно (операции попарные, не берут больше времени при увеличении числа процессов).

В конце работы программы собираем решение по частям на нулевом процессе. В данной реализации мы соберём только решение в финальный момент времени, однако можно было бы собрать и в промежуточные (не обязательно все).

```
63 if rank_cart == 0:
64     u_T = empty(N+1, dtype=float64)
65 else:
66     u_T = None
67
68 if rank_cart == 0:
```

```
69      comm_cart.Gatherv([u_part_aux[M, :-1], N_part, MPI.DOUBLE],
70                        [u_T, rcounts, displs, MPI.DOUBLE], root=0)
71  if rank in range(1, numprocs-1):
72      comm_cart.Gatherv([u_part_aux[M, 1:-1], N_part, MPI.DOUBLE],
73                        [u_T, rcounts, displs, MPI.DOUBLE], root=0)
74  if rank == numprocs-1:
75      comm_cart.Gatherv([u_part_aux[M, 1:], N_part, MPI.DOUBLE],
76                        [u_T, rcounts, displs, MPI.DOUBLE], root=0)
77
```

Дальше остаётся только вывести затраченное время и сохранить полученные данные в файл. Соответсвующий код уже приводился.

Сравнение эффективности реализованных алгоритмов

Мы рассмотрели две программные реализации. Первая реализация, несомненно, гораздо менее эффективна. Во второй мы постарались учесть ошибки первой. Построим графики ускорения работы программы и убедимся, что время работы убывает с ростом процессов.

Расчёты проводились на суперкомпьютере ”Ломоносов-2”в тестовой очереди (подробнее см. лекцию 4). В данном случае, выбрав параметры сетки N = 800, M = 300_000, мы получаем, что сеточные значения занимают $N \times M \times 8$ байт $= 1.79$ GB (напомним, что такой выбор M обусловлен особенностями алгоритма, а не желанием показать значительное ускорение).

Время работы последовательной программы – 649.3 секунды.

На графике 8.4 к своему изумлению замечаем, что первая реализация (интуитивно менее эффективная) работает быстрее до 14 узлов. Это связано с тем, что на суперкомпьютере на одном узле располагается 14 CPU-ядер, а значит и оперативная память у них общая. Как только мы выходим за пределы одного узла, возникает потребность в приёме и передаче сообщений не внутри одного узла, где память только копируется, а между несколькими узлами. В таком случае более эффективная реализация, конечно, побеждает.

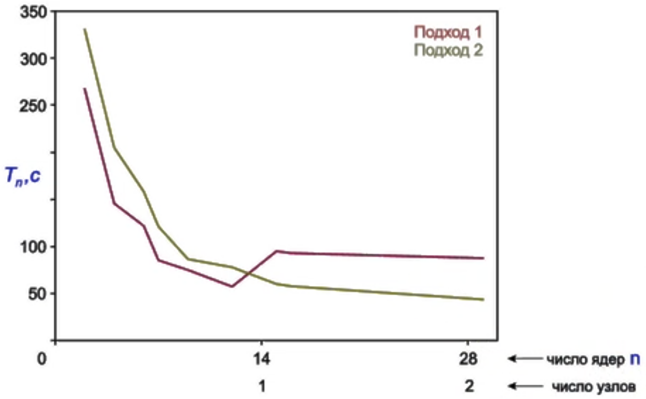


Рис. 8.4: График времени работы параллельных программ

Теперь построим тот же самый график для большего количества ядер (см. рис. 8.5). Видно, что программа, реализованная в рамках первого подхода, после 140 ядер считает даже дольше, чем однопроцессорная.

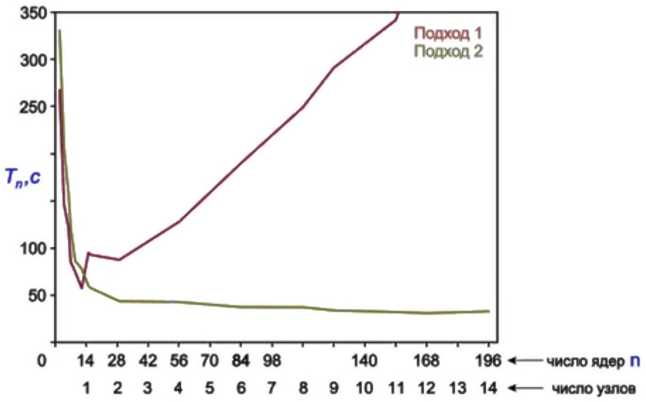


Рис. 8.5: График времени работы параллельных программ

Значения времени работы программы второго подхода выходят на асимптоту. $T_\infty \approx 30$ секунд – время пересылки сообщений, не зависит от числа процессов. Для первого подхода время коммуникации только увеличивается.

Насколько хорош второй подход, если около `numprocs=200` время пересылки сообщений занимает 30 секунд, а время счета - около 3 секунд? Понятно, что в данный момент мы рассматриваем тестовый пример. В дальнейшем можно значительно увеличить количество сеточных значений, но время на пересылку сообщений не изменится.

Также приведём график ускорения работы программ (см. рис. 8.6).

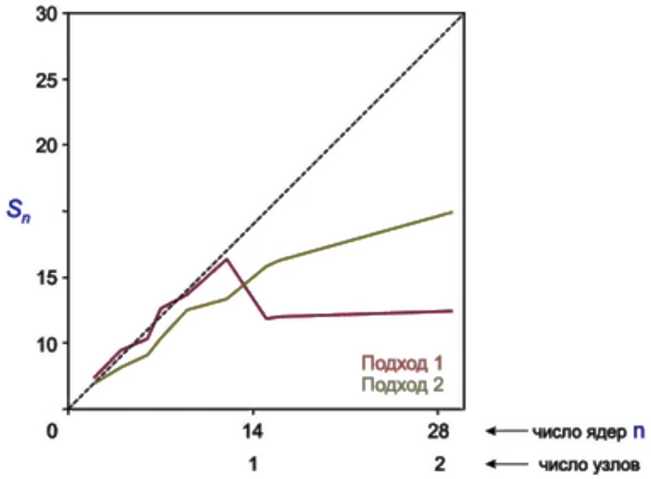


Рис. 8.6: График ускорения параллельных программ

На этой лекции мы рассмотрели явные схемы решения уравнений в частных производных. На следующей лекции мы рассмотрим реализацию неявных схем, где нам потребуется решать систему с трёхдиагональной матрицей (наработки предыдущей лекции).

Лекция 9. Подходы к распараллеливанию алгоритмов решения задач для уравнений в частных производных.

Пример одномерной по пространству начально-краевой задачи для уравнения в частных производных параболического типа.

Сегодня мы продолжим рассматривать различные подходы к распараллеливанию алгоритмов, которые возникают при решении задач для уравнений в частных производных. Обратимся к уже рассмотреной задаче:

$$\begin{cases} \varepsilon \dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial u}{\partial t} = -u\dfrac{\partial u}{\partial x} - u^3, \quad x \in (a,b), \quad t \in (t_0, T], \\ u(a,t) = u_{\textit{left}}(t), \quad u(b,t) = u_{\textit{right}}(t), \quad t \in (t_0, T], \\ u(x,t_0) = u_{\textit{init}}(x), \quad x \in [a,b]. \end{cases} \tag{9.1}$$

Один из вариантов численного решения явной схемы которой мы рассматривали на прошлом занятии. Постановка задачи рассматривалась в лекции 8.

Последовательный алгоритм решения, основанный на реализации неявной схемы.

На этой лекции мы будем распараллеливать неявную схему $ROS1$ для данного нахождения сеточных значений $u_n^m = u(x_n, t_m))$ уравнения:

$$\Big[E - \alpha\tau \vec{f}_{\vec{y}}(\vec{y}_m, t_m)\Big]\vec{w}_1 = \vec{f}(\vec{y}_m, t_m + \frac{\tau}{2}), \qquad m = \overline{0, M-1}, \vec{y}_{m+1} = \vec{y}_m + \tau Re\vec{w}_1. \tag{9.2}$$

Где $\vec{y}_m = \big(u_1^m, u_2^m, \ldots, u_{N-2}^m, u_{N-1}^m\big)$. Теперь мы можем используя эти формулы по набору сеточных значений на текущем временном слое найти значения на следующем временном слое. Продолжая этот процесс для каждого M можно получить решение задачи. Отличительная черта схемы в этом занятии заключается в отсутствии возможности нахождения решения по отдельности. И получить сеточные решения можно только целиком, потому что нужно решить СЛАУ. Однако эта схема, хотя и требует более сложную программную реализацию, но имеет более высокую точность и устойчива, при $\alpha = \dfrac{1+}{2}$. И мы можем получать лучший результат на менее густых сетках. Также и явная и неявная схема по вычислительной трудоемкости одинаковы. При переходе со одного временного слоя на следующий нам требуется N^1 операций для обоих схем, так как матрица системы для

неявной схемы имеет трехдиагональный вид. Запишем формулы для $\vec{f}$, $\vec{f_y}$, их вывод можно посмотреть во второй половине 19 лекции по численным методам.

$$f[0] = \varepsilon\frac{y[1] - 2y[0] + u_{left}(t)}{h^2} + y[0]\frac{y[1] - u_{left}(t)}{2h} + y[0]^3,$$
$$f[n] = \varepsilon\frac{y[n+1] - 2y[n] + y[n-1]}{h^2} + y[n]\frac{y[n+1] - y[n-1]}{2h} + y[n]^3, \quad n = \overline{2, N-2},$$
$$f[N-2] = \varepsilon\frac{u_{right}(t) - 2y[N-2] + y[N-3]}{h^2} + y[N-2]\frac{u_{right}(t) - y[N-3]}{2h} + y[N-2]^3.$$

$$\vec{f_y}[0,0] = 1 - \alpha\tau\Big(\frac{-2\varepsilon}{h^2} + \frac{y[1] - u_{left}(t)}{2h} + 3y[0]^2\Big),$$
$$\vec{f_y}[0,1] = -\alpha\tau\Big(\frac{\varepsilon}{h^2} + \frac{y[0]}{2h}\Big),$$
$$\vec{f_y}[n,n-1] = -\alpha\tau\Big(\frac{\varepsilon}{h^2} - \frac{y[n]}{2h}\Big), \qquad n = \overline{1, N-3},$$
$$\vec{f_y}[n,n] = 1 - \alpha\tau\Big(\frac{-2\varepsilon}{h^2} + \frac{y[n+1] - y[n-1]}{2h} + 3y[n]^2\Big), \qquad n = \overline{1, N-3},$$
$$\vec{f_y}[n,n+1] = -\alpha\tau\Big(\frac{\varepsilon}{h^2} + \frac{y[n]}{2h}\Big), \qquad n = \overline{1, N-3},$$
$$\vec{f_y}[N-2,N-3] = -\alpha\tau\Big(\frac{\varepsilon}{h^2} - \frac{y[N-2]}{2h}\Big),$$
$$\vec{f_y}[n,n] = 1 - \alpha\tau\Big(\frac{-2\varepsilon}{h^2} + \frac{u_{right}(t) - y[N-3]}{2h} + 3y[N-2]^2\Big).$$

где τ - шаг сетки по времени, h - шаг сетки по пространству, ε - параметр задачи, α определяет схему.

Программная реализация последовательного алгоритма решения.

Рассмотрим процесс решения задачи:

```
from numpy import empty, linespace, sin, pi
...
for m in range(M):
    codiagonal_down, diagonal, codiagonal_up = diagonal_preparation(
        y, t[m], h, N, u_left, u_right, eps, tau, alpha)
    w_1  = consecutive_tridiagonal_matrix_algorithm(
        codiagonal_down, diagonal, codiagonal_up,
        f(y, t[m] + tau/2, h, N, u_left, u_right, eps))
    y = y + tau*w_1.real
    u[m  + 1, 0] = u_left(t[m+1])
    u[m + 1, 1:N] = y
```

```
        u[m + 1, N] = u_right(t[m + 1])
```

На каждом шаге при помощи $\vec{y}_m$ и параметров задачи подготавливается трехдиагональная матрица системы, после применяется метод прогонки и находится вектор $\vec{w}_1$. После этого рассчитывается новый вектор $\vec{y}$, и заполняется массив сеточных значений u_m^n.

```
    def f(y, t, h, N, u_left, u_right, eps):
        f = empty (N - 1, dtype = float64)
        f[0] = eps * (y[1]  -2 * y[0]  + u_left(t))/h**2 + \
            y[0]* (y[1] - u_left(t))/(2 *h) + y[0]**3
        for n in range(1, N - 2):
            f[n] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +\
                y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
        f[N - 2] = eps * (u_right(t) - 2*y[N - 2] + y[N - 3])/h**2 +\
            y[N - 2]*(u_right(t) - y[N - 3])/(2*h)+y[N - 2]**3
        return f

    def diagonal_preparation(y, t, h, N, u_left, u_right, eps, tau, alpha)
:
        a = empty(N - 1); b = empty(N - 1); c = empty(N - 1)
        b[0] = 1. - alpha*tau*(-2*eps/h**2 + (y[1] - u_left(t)/(2*h) + 3*y
[0]**2))
        c[0] = - alpha * tau*(eps/h**2  + y[0]/(2*h))
        for n in range(1, N - 2):
            a[n] = -alpha*tau*(eps/h**2 -y[n]/(2*h))
            b[n] = 1. - alpha*tau*(-2*eps/h**2 + (y[n + 1] - y[n - 1])/(2*
h) + 3*y[n]**2)
            c[n] = -alpha*tau*(eps/h**2 + y[n]/(2*h))
        a[N - 2] = -alpha*tau*(eps/h**2 -y[N - 2]/(2*h))
        b[N - 2] = 1. - alpha*tau*(-2*eps/h**2 + (u_right(t) - y[N - 3])
/(2*h) + 3*y[N - 2]**2)
        return a,b,c

    def consecutive_tridiaginal_matrix_alghorithm(a, b, c, d):
        N = len(d)
        x = empty(N, dtype = float64)
        for n in range(1,N):
            coef = a[n]/b[n - 1]
            b[n] = b[n] - coef*c[n - 1]
            d[n] = d[n] - coef*d[n - 1]
        x[N - 1] = d[N - 1]/b[N - 1]
        for n in range(N-2, -1, -1)
            x[n] = (d[n] - c[n]*x[n+1])/b[n]
        return x
```

Выше приведены функции с помощью которых выполняются вычисления. При работе с комплексными числами тип данных с $float64$ необходимо изменить на $complex128$.

Параллельный алгоритм решения, основанный на реализации неявной схемы.

Идея программной реализации взята с прошлой лекции. Каждый MPI процесс делает вычисления только для какой-то части узлов по пространство. При переходе по временным слоям нам необходимо решать СЛАУ для нахождения матрицы w_1. Решать СЛАУ мы будем параллельно используя программу с 7 лекции. На каждом процессе мы будем формировать правую часть системы и коэффициенты уравнения и в результате на каждом процессе будем получать свой промежуток вектора y. Но при формировании нашего вектора необходимо будет пользоваться не только сеточные значения кусочка вектора y, содержащиеся на текущем временном слое, но и узлы с соседних процессов. Таким образом каждому процессу для перехода со слоя на слой будет требоваться не только узлы находящиеся на временном слое, но и соседнии с ними. Поэтому у нас будет распределение данных по процессам по двух типов. С одной стороны это будут массивы узлов в которых вычисляются приближенные значения функции для следующего временного слоя. С другой стороны нам будет требоваться массив узлов с текущего временного слоя большего размера, захватывающий соседей.

Программная реализация параллельного алгоритма решения.

```
1     from mpi4py import MPI
2     from numpy import empty, array, int32, float64 linespace, sin, pi
3     from matplotlib.pyplot import style, figure, axes, show
4
5     comm = MPI.COMM_WORLD
6     numprocs = comm.Get_size()
7
8     comm_cart = comm.Create_cart(dims = [numprocs], periods = [False],
    reorder=True)
9     rank_cart = comm_cart.Get_rank()
10
11    def u_init(x):
12        u_init = sin(3*pi*(x-1/6))
13        return u_init
14
15    def u_left(t):
16        u_left = -1.
```

```
17          return u_left
18
19      def u_right(t):
20          u_right = 1.
21          return u_right
22      ...
23
```

В отличие от наших прошлых программ не хватает строчки $rank = comm.Get\_rank()$, ее мы заменили потому что на основе нашего коммуникатора $COMM\_WORLD$ мы создаем декартову топологию вида линейки. В которой разрешена перенумерация процессов, чтобы заданная нами топология как можно лучше ложилась на физическую топологию компьютера. Однако нужно иметь ввиду что такое не всегда возможно и иногда результата у такого действия может не быть. Функции для формирования правых частей линейных алгебраических уравнений будут показаны позже.

```
184     if rank_cart == 0:
185         state_time = MPI.Wtime()
186
187     a = 0; b = 1
188     t_0 = 0; T = 2/0
189     eps = 10**(-1.5)
190
191     N = 200; M = 300; alpha = 0.5
192
193     h = (b - a)/N; x = linespace(a, b, N + 1)
194     tau = (T -t_0)/M; t = linespace(t_0, T, M + 1)
195
196     if rank_cart == 0:
197         ave, res = divmod(N + 1, numprocs)
198         rcounts = empty(numprocs, dtype=int32)
199         displs = empty(numprocs, dtype = int32)
200         for k in range(0, numprocs):
201             if k < res:
202                 rcounts = ave + 1
203             else:
204                 rcounts = ave
205             if k == 0:
206                 displs[k] = 0
207             else:
208                 displs[k] = displs[k - 1] + rcount[k - 1]
209     else:
210         rcounts = None; displs = None
211
212     N_part = array(0, dtype=int32)
213
214     comm_cart.Scatter([rcounts, 1, MPI.INT], [N_part, 1, MPI.INT], root =
```

```
    0)
215
```

Программа начинается с установки отправной точки по времени. Затем определяются параметры задачи и численного решения. После идут два привычных шага. Массивы $rcounts$ содержат информацию о числе узлов за вычисление которых ответственен каждый MPI процесс. Также на каждом процессе выделяем место в памяти под число узлов за которое ответсвенен каждый процесс. И в конце отправляем эти массивы на все процессы.

```
216     if rank_cart == 0:
217         rcounts_aux = empty(numprocs, dtype=int32)
218         displs_aux = empty(numprocs, dtype = int32)
219         rcounts_aux[0] = rcounts[0] + 1
220         displs_aux[0] = 0
221         for k in range(1, numprocs-1):
222             rcounts[k] = rcounts[k] + 2
223             displs_aux[k] = displs[k] - 1
224         rcounts_aux[numprocs-1] = rcounts[numprocs-1] + 1
225         displs_aux[numprocs-1] = displs[numprocs-1] - 1
226     else:
227         rcounts_aux = None; displs_aux = None
228
229     N_part_aux = array(0, dtype=int32)
230
231     comm_cart.Scatter([rcounts_aux, 1, MPI.INT], [N_part_aux, 1, MPI.INT],
    root = 0)
232     comm_cart.Scatter([displs_aux, 1, MPI.INT], [displs_aux, 1, MPI.INT],
   root = 0)
233
234     u_part_aux = empty((M +1, N_part_aux), dtype = float64)
235
236     for n in range(N_part_aux):
237         u_part_aux[0, n] = u_init(x[displs_aux + n])
238
239     y_part = u_part_aux[0, 1:N_part_aux-1]
240
```

Эти масивы содержат в себе информацию о наборе узлов с помощью которого делаются вычисления на следующем временном слое. Соответственно на первом и последнем слое требуется один дополнительный узел, а на промежуточных по два дополнительных узла. И аналогично создается переменная для хранения числа узлов с помощью которых производятся вычисления. Массив u_part_aux – это множество узлов пространственно-временной сетки за рассчет в которых ответственен соответствующий MPI процесс. Далее заполняем с помощью начального условия массив u_part_aux, и формируем вспомо-

гательный массив y. Для него мы исключали индексы 0 и N, что и учитывается в программе.

```
    for m in range(M):
        codiagonal_down_part, diagonal_part, codiagonal_up_part =
            diagonal_preparation(u_part_aux[m], t[m], h,
                N_part_aux, u_left, u_right, eps, tau, alpha)
        w_1_part = parallel_tridiagonal_matrix_algorithm(
            codiagonal_down_part, diagonal_part, codiagonal_up_part,
            f_part(u_part_aux[m], t[m]+tau/2, h, N_part_aux, u_left,
            u_right, eps))
        y_part = y_part + tau*w_1.real

        u_part_aux[m + 1, 1:N_part_aux - 1] = y_part
        if rank_cart == 0:
            u_part_aux[m+1, 0] = u_left(t[m+1])
        if rank_cart == numprocs - 1:
            u_part_aux[m+1, N_part_aux-1] = u_right(t[m+1])

        if rank_cart ==0:
            comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
                1, MPI.DOUBLE], dest=1, sendtag=0, recvbuf=
                [u_part_aux[m+1, N_part_aux-1:], 1, MPI.DOUBLE],
                source=1, recvtag=MPI.ANY_TAG, status=None)
        if rank_cart in range(1, numprocs-1):
            comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
                1, MPI.DOUBLE], dest=rank_cart-1, sendtag=0, recvbuf=
                [u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
                source=rank_cart-1, recvtag=MPI.ANY_TAG, status=None)
            comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_part_aux-2],
                1, MPI.DOUBLE], dest=rank_cart+1, sendtag=0, recvbuf=
                [u_part_aux[m+1, N_part_aux-1:], 1, MPI.DOUBLE],
                source=rank_cart+1, recvtag=MPI.ANY_TAG, status=None)
        if rank_cart == numprocs-1:
            comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
                1, MPI.DOUBLE], dest=numprocs-2, sendtag=0, recvbuf=
                [u_part_aux[m+1, 0:], 1, MPI.DOUBLE],
                source=numprocs-2, recvtag=MPI.ANY_TAG, status=None)


```

Рассмотрим реалзиацию одностадийной схемы Розенброка. Сначала создаем части диагоналей, которые затем вместе с частью правой части уравнения используются для рассчета вектора w_1. Затем рассчитываем части вспомогательного вектора y_part, и заполняем соответствующие сеточные значения функции. После дозаполняем функции с помощью известных граничных и начальных условий. После этого идет передача сообщений между процессорами, для дополнения векторов значениями, которые требуются для рассчета

на следующем временном слое. Важное замечание: в нашем случае несмотря на громоздкость выражения с ростом количества процессоров количество действий не увеличивается, потому что каждый процесс, кроме первого и последнего, фактически передают и принимают два числа вне зависимости от числа процессов.

```
276     if rank_cart == 0:
277         u_T = empty(N+1, dtype=float64)
278     else:
279         u_T = None
280 
281     if rank_cart == 0:
282         comm_cart.Gatherv([u_part_aux[M, 0:N_part_aux-1],N_part,
283             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
284     if rank_cart in range(1, numprocs-1):
285         comm_cart.Gatherv([u_part_aux[M, 1:N_part_aux-1],N_part,
286             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
287     if rank_cart == numprocs-1:
288         comm_cart.Gatherv([u_part_aux[M, 1:N_part_aux],N_part,
289             MPI.DOUBLE], [u_T, rcounts, displs, MPI.DOUBLE], root=0)
290 
291     if rank_cart == 0:
292         end_time = MPI.Wtime()
293         print('N={}, M={}'.format(N,M ))
294         print('Number of MPI process is {}'.format(numprocs))
295         print('Elapsed time is {:.4f} sec'.format(end_time-start_time))
296 
297         style.use('dark_background')
298         fig = figure()
299         ax = axes(xlim(a,b), ylim=(-2.0, 2.0))
300         ax.set_xlabel('x'); ax.set_ylabel('u')
301         ax.plot(x, u_T, color='y', ls='-', lw=2)
302         show()
303 
304 
```

Для проверки программы мы выделяем память под массив под сеточные значения функции в финальный момент времени T, затем собираем со всех процессов наш финальный вектор. И в конце выводим время работы и график решения в финальный момент времени.

```
133     def f_part(y, t, h, N_part_aux, u_left, u_right, eps):
134         N_part = N_part_aux - 2
135         f_part = empty(N_part, dtype=float64)
136         if rank_cart == 0:
137             f_part[0] = f[0] = eps * (y[1]  -2 * y[0]  + u_left(t))/h**2 +
    \
138             y[0]*(y[1] - u_left(t))/(2 *h) + y[0]**3
139             for n in range(2, N_part_aux-1):
```

```
140                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
141                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
142         if rank_cart in range(1, numprocs-1):
143             for n in range(2, N_part_aux-1):
144                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
145                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
146         if rank_cart == numprocs-1:
147             for n in range(1, N_part_aux-2):
148                 f_part[n-1] = eps * (y[n + 1] - 2*y[n] + y[n - 1])/h**2 +
    \
149                     y[n]*(y[n + 1] - y[n - 1])/(2*h)+y[n]**3
150             f_part[N_part-1] = eps*(u_right(t) - 2*y[N_part_aux-2]
151                 + y[N_part_aux-3])/h**2 + y[N_part_aux-2]*(u_right(t) -
152                 y[N_part_aux-2])/(2*h)+y[N_part_aux-2]**3
153
```

Важно понимать, что передается не весь вектор y, а только его часть длины $N\_part\_aux - 2$, для соответствующего процесса. Для первого и нулевого процессора учитывается тот факт, что нам известны левое и правое граничное условие. Функция определяющая диагонали матрицы переписывается полностью аналогично и представленна в полной программе, которая находится в материалам к лекции. Последовательный метод прогонки совпадает с написанным ранее, а параллельный метод прогонки скопирован из 7 лекции, только $comm$ заменено на $comm\_cart$, а $rank$ на $rank\_cart$.

Обсуждение эффективности программной реализации.

Время работы последовательной версии программы на одном ядре составило 560с., при параметрах $N = 20000, M = 5000$. Видно, что с увеличением числа ядер время уменьшается, при том примерно до 126 ядер график получился весьма монотонным, доходит до 24с., а после начинает монотонно расти. Ранее передача сообщений между процессорами была организована таким образом, что время передачи сообщений не зависила от числа участвующих в вычислениях процессов. Получалось, что с ростом числа процессов время почти стремится к нулю и остается только время на прием/передачу сообщений. Рост после 126 ядер обусловлен тем, что внутри функции решающей алгоритм с трехдиагональной матрицей несколько раз используются команды $Gather$ и $Scatter$ и накладные расходы по их использованию возрастают с числом используемых для вычислений процессов. Ускорение рассчитывается как время работы на n процессорах деленное на время работы на одном процессоре. График нарисованный пунктиром это случай линейного ускорения. В нашем случае сначала ускорение (красны график) ниже линейного

случая, а затем начинает уменьшаться, т.к. начинают сказываться накладные расходы на прием/передачу сообщений. Сначала же мы не достигаем линейного ускорения, потому что наш алгоритм требует конкретно $17N$ операций при переходе со одного временного слоя на другой, а до этого рассмотренный алгоритм требовал $8N$ операции. Таким образом умножив значения на $\frac{17}{8}$ мы получаем зеленый график, и видно что ускорение в нашем алгоритме более существенное. Эффективность распараллеливания мы считаем как $\frac{S_n}{n}$, в случае линейного ускорения эффективность получается равной 1. Есть условность считать эффективность порядка 0.8 очень хорошей. И так как нас интересует опять именно эффективность работы нашей программы, и то сколько времени процессоры простаивают и не делают рассчетов, мы должны снова домножить наш график на $\frac{17}{8}$. И зеленый график показывает эффективность именно нашей программы. Очень важно понимать что именно мы подразумеваем под эффективностью и для чего это нам нужно.

Лекция 10. Подходы к распараллеливанию алгоритмов решения задач для уравнений в частных производных.

Продолжаем рассматривать построение и программную реализацию параллельных алгоритмов, которые могут понадобиться при решении задач для уравнений в частных производных.

На предыдущих двух лекциях мы рассматривали задачи в частных производных, одномерные по пространству. На этой лекции нам предстоит рассмотреть обобщение на многомерный случай.

Пример двумерной по пространству начально-краевой задачи для уравнения в частных производных параболического типа

$$\begin{cases} \varepsilon\left(\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}\right)-\dfrac{\partial u}{\partial t}=-u\left(\dfrac{\partial u}{\partial x}+\dfrac{\partial u}{\partial y}\right)-u^3, \quad (x,y)\in(a,b)\times(c,d), \quad t\in(t_0,T], \\ u(a,y,t)=u_{left}(y,t), \quad u(b,y,t)=u_{right}(y,t), \quad y\in(c,d), \quad t\in(t_0,T], \\ u(x,c,t)=u_{bottom}(x,t), \quad u(x,d,t)=u_{top}(x,t), \quad x\in[a,b], \quad t\in(t_0,T], \\ u(x,y,t_0)=u_{init}(x,y), \quad (x,y)\in[a,b]\times[c,d]. \end{cases} \tag{10.1}$$

Из постановки видно, что задача нелинейная, а значит аналитически её решить не представляется возможным. На помощь приходят численные методы. Подробно решение конкретно этой задачи было разобрано в курсе ”Численные методы лекция 27. Напомним лишь общую идею.

Последовательный алгоритм решения, основанного на реализации явной схемы

Введём пространственно-временную сетку (см. рис. 10.1).

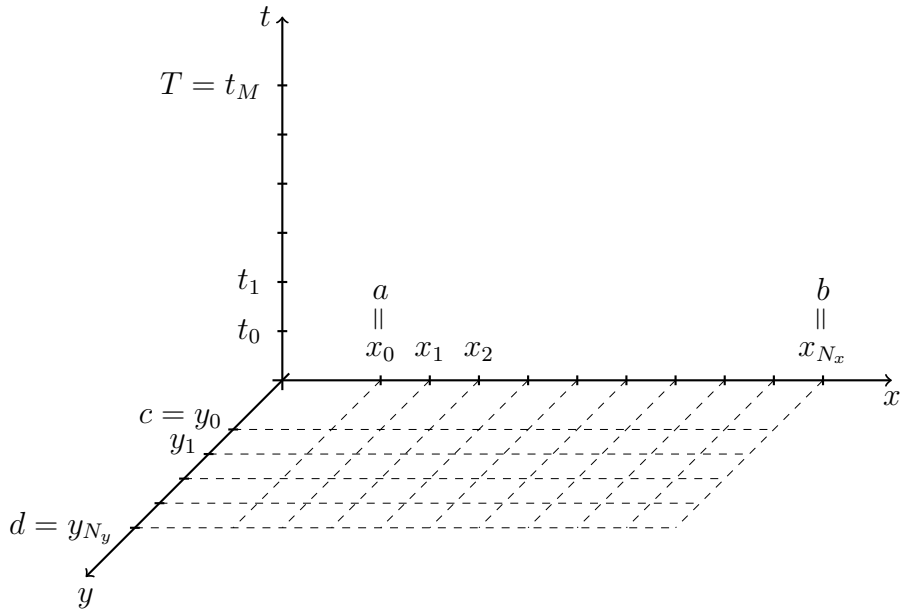


Рис. 10.1: Пространственно-временная сетка для задачи 10.1

На нулевом слое по времени все значения известны абсолютно точно из начальных условий. На последующих слоях абсолютно точно известны только значения на границе пространственной области.

Для построения разностной схемы будем использовать шаблон, изображённый на рис. 10.2.

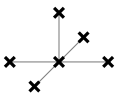


Рис. 10.2: Шаблон для явной схемы алгоритма решения задачи 10.1

Применяя указанный шаблон, получим:

$$u_{i,j}^{m+1} = u_{i,j}^m + \tau \left(\varepsilon \left(\frac{u_{i+1,j}^m - 2u_{i,j}^m + u_{i-1,j}^m}{h_x^2} + \frac{u_{i,j+1}^m - 2u_{i,j}^m + u_{i,j-1}^m}{h_y^2} \right) + \right.$$
$$\left. + u_{i,j}^m \left(\frac{u_{i+1,j}^m - u_{i-1,j}^m}{2h_x} + \frac{u_{i,j+1}^m - u_{i,j-1}^m}{2h_y} \right) + \left(u_{i,j}^m\right)^3 \right).$$

Подразумевается, что значения на временном слое m нам известны, тогда мы можем найти все внутренние значения на временном слое $m+1$. Обратим внимание, что каждое

значение $u_{i,j}^{m+1}$ можно вычислять независимо от других, а значит, существует возможность распараллеливания программы.

Рассмотрим несколько подходов для распараллеливания программы. Первый из них будет опираться на подходы, рассмотренные на 8 лекции, и обобщать их. Во втором же подходе мы попробуем существенно учитывать то, что задача двумерна по пространству.

Но начнём, все-таки, с последовательной версии программы.

Программная реализация последовательного алгоритма решения

```
from numpy import empty, linspace, tanh, savez
import time

def u_init(x, y):
    return 0.5*tanh(1/eps*((x-0.5)**2 + (y-0.5)**2 - 0.35**2)) - 0.17

def u_left(y, t):
    return 0.33

def u_right(y, t):
    return 0.33

def u_top(x, t):
    return 0.33

def u_bottom(x, t):
    return 0.33

start_time = time.time()

a = -2; b = 2; c = -2; d = 2
t_0 = 0; T = 4; eps = 10**(-1.5)

N_x = 50; N_y = 50; M = 500
```

Такая большая разница между параметрами сетки по пространству и по времени взята не просто так, мы делаем так для того, чтобы численный счёт не разваливался. Это – особенность явной схемы.

Далее заполняем значения на нулевом временном слое и значения на границе:

```
x, h_x = linspace(a, b, N_x+1, retstep=True)
y, h_y = linspace(c, d, N_y+1, retstep=True)
t, tau = linspace(t_0, T, M+1, retstep=True)
```

```
28
29 u = empty((M+1, N_x+1, N_y+1))
30
31 for i in range(N_x+1):
32     for j in range(N_y+1):
33         u[0, i, j] = u_init(x[i], y[j])
34
35 for m in range(M+1):
36     for j in range(1, N_y):
37         u[m, 0, j] = u_left(y[j], t[m])
38         u[m, N_x, j] = u_right(y[j], t[m])
39
40     for i in range(0, N_x+1):
41         u[m, i, N_y] = u_top(x[i], t[m])
42         u[m, i, 0] = u_bottom(x[i], t[m])
```

Наконец, пробегаем по всем слоям и вычисляем значения функции внутри пространственной области:

```
43 for m in range(M):
44     for i in range(1, N_x):
45         for j in range(1, N_y):
46             d2x = (u[m, i+1, j] - 2*u[m, i, j] + u[m, i-1, j]) / h_x**2
47             d2y = (u[m, i, j+1] - 2*u[m, i, j] + u[m, i, j-1]) / h_y**2
48
49             d1x = (u[m, i+1, j] - u[m, i-1, j]) / (2*h_x)
50             d1y = (u[m, i, j+1] - u[m, i, j-1]) / (2*h_y)
51
52             u[m+1, i, j] = u[m, i, j] + tau*(eps*(d2x + d2y) +
53                            u[m, i, j]*(d1x + d1y) + u[m, i, j]**3)
54
55 print(f"Elapsed time is {time.time() - start_time:.4f} sec")
56 savez("results", x=x, y=y, t=t, u=u)
```

После завершения работы программы можно построить график:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from matplotlib.animation import FuncAnimation
4
5  frames = 100
6  results = np.load("results.npz")
7
8  x = results["x"]
9  y = results["y"]
10 t = results["t"]
11 u = results["u"]
12
```

```
13 fig, ax = plt.subplots()
14
15 xx, yy = np.meshgrid(x, y, indexing="ij")
16
17 def plot_frame(s):
18     ax.clear()
19
20     ax.set_xlabel("x")
21     ax.set_ylabel("y")
22     ax.set_aspect("equal")
23
24     m = s * (u.shape[0] // frames)
25
26     ax.pcolor(xx, yy, u[m], shading="auto")
27     ax.set_title(f"m = {m}")
28
29 anim = FuncAnimation(fig, plot_frame, interval=60, frames=frames)
30 anim.save("results.gif")
```

Параллельный алгоритм решения в случае одномерного деления пространственной области на блоки

Рассмотрим один временной слой и разобьём его на части по процессам (см. рис. 10.3). Бледно-серым показаны значения, которые необходимы нескольким процессам, то есть их необходимо будет пересылать.

Несложно заметить, что длина пересылаемых сообщения будет равна $N_y - 1$ (при этом граничные процессы передают и получают по одному сообщению на каждой итерации по времени, а все остальные – по два).

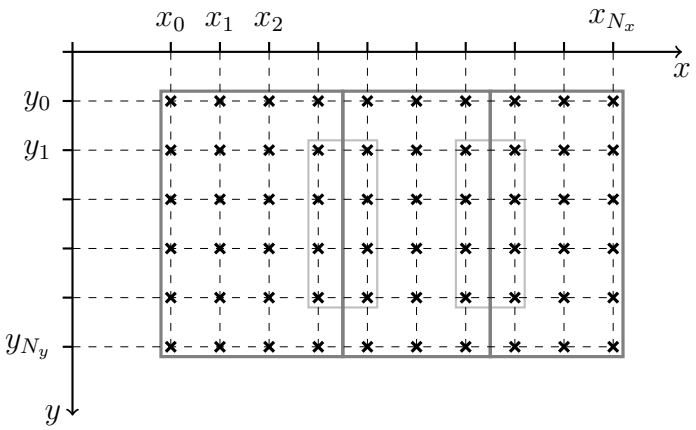


Рис. 10.3: Схема разделения данных по MPI процессам

Программная реализация параллельного алгоритма решения

Полная версия программы выложена в разделе ”Материалы к лекции”под номером 10-2.

```
1  from mpi4py import MPI
2  from numpy import empty, int32, float64, linspace, tanh, savez
3
4  comm = MPI.COMM_WORLD
5  numprocs = comm.Get_size()
```

Вводим виртуальную декартову топологию типа ”линейка”(т.к. одномерная). Аргумент `reorder=True` позволяет расположить физически близкие процессоры под близкими номерами.

```
6  comm_cart = comm.Create_cart(dims=[numprocs],
7                               periods=[False], reorder=True)
8  rank_cart = comm_cart.Get_rank()
9
10 ...
```

После этого снова задаём все параметры задачи (строки 4-17, 21-27 в последовательной программе). Изменится только измерение времени:

```
11 if rank_cart == 0:
12     start_time = MPI.Wtime()
```

Снова введём функции для вычисления вспомогательных массивов `rcounts` и `displs` и применим её для вычисления длин и сдвигов для частей массива `u`.

```
13 def auxiliary_arrays_determination(M, num):
14     ave, res = divmod(M, num)
15 
16     rcounts = empty(num, dtype=int32)
17     displs = empty(num, dtype=int32)
18 
19     for k in range(num):
20         rcounts[k] = ave + 1 if k < res else ave
21         displs[k] = 0 if k == 0 else displs[k-1] + rcounts[k-1]
22 
23     return rcounts, displs
24 
25 rcounts_N_x, displs_N_x = auxiliary_arrays_determination(N_x+1, numprocs)
26 
27 N_x_part = rcounts_N_x[rank_cart]
```

Помним, что нужно также хранить дополнительные столбцы слева и справа.

```
28 if rank_cart in [0, numprocs-1]:
29     N_x_part_aux = N_x_part + 1
30 else:
31     N_x_part_aux = N_x_part + 2
32 
33 displs_N_x_aux = displs_N_x - 1
34 displs_N_x_aux[0] = 0
35 
36 displ_x_aux = displs_N_x_aux[rank_cart]
37 
38 u_part_aux = empty((M+1, N_x_part_aux, N_y+1), dtype=float64)
```

Заполняем вспомогательный массив значениями в начальный момент времени и на границах:

```
39 for i in range(N_x_part_aux):
40     for j in range(N_y+1):
41         u_part_aux[0, i, j] = u_init(x[displ_x_aux+i], y[j])
42 
43 for m in range(M):
44     for j in range(1, N_y):
45         if rank_cart == 0:
46             u_part_aux[m, 0, j] = u_left(y[j], t[m])
47         if rank_cart == numprocs-1:
48             u_part_aux[m, 0, j] = u_right(y[j], t[m])
49 
50     for i in range(N_x_part_aux):
51         u_part_aux[m, i, N_y] = u_top(x[displ_x_aux+i], t[m])
52         u_part_aux[m, i, 0] = u_bottom(x[displ_x_aux+i], t[m])
```

Переходим к основной части программы.

```
for m in range(M):

    for i in range(1, N_x_part_aux-1):
        for j in range(1, N_y):
            d2x = (u_part_aux[m, i+1, j] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i-1, j]) / h_x**2
            d2y = (u_part_aux[m, i, j+1] -
                   2*u_part_aux[m, i, j] +
                   u_part_aux[m, i, j-1]) / h_y**2

            d1x = (u_part_aux[m, i+1, j] -
                   u_part_aux[m, i-1, j]) / (2*h_x)
            d1y = (u_part_aux[m, i, j+1] -
                   u_part_aux[m, i, j-1]) / (2*h_y)

            u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
                                    tau*(eps*(d2x + d2y) +
                                    u_part_aux[m, i, j]*(d1x + d1y) +
                                    u_part_aux[m, i, j]**3)
```

Остаётся только передать значения соседним процессам. Напомним, что в данной реализации простаивающих процессов нет, а размер пересылаемых сообщений не зависит от количества процессов.

```
    if rank_cart > 0:
        comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
                                    N_y+1, MPI.DOUBLE],
                           dest=rank_cart-1,
                           sendtag=0,
                           recvbuf=[u_part_aux[m+1, 0],
                                    N_y+1, MPI.DOUBLE],
                           source=rank_cart-1,
                           recvtag=MPI.ANY_TAG,
                           status=None)

    if rank_cart < numprocs-1:
        comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_x_part_aux-2],
                                    N_y+1, MPI.DOUBLE],
                           dest=rank_cart+1,
                           sendtag=0,
                           recvbuf=[u_part_aux[m+1, N_x_part_aux-1],
                                    N_y+1, MPI.DOUBLE],
                           source=rank_cart+1,
                           recvtag=MPI.ANY_TAG,
                           status=None)
```

```
94
95 if rank_cart == 0:
96     end_time = MPI.Wtime()
```

Не всегда разумно собирать все полученные данные на одном процессе, так как для них может просто не хватить памяти. Можно собрать данные только на некоторых временных слоях. Мы, для простоты, просто соберём данные в финальный момент времени.

```
97 if rank_cart == 0:
98     u_T = empty((N_x+1, N_y+1), dtype=float64)
99
100 if rank_cart == 0:
101     comm_cart.Gatherv([u_part_aux[M, :-1], N_x_part*(N_y+1), MPI.DOUBLE],
102                       [u_T, rcounts_N_x*(N_y+1), displs_N_x*(N_y+1),
103                        MPI.DOUBLE], root=0)
104
105 elif rank_cart in range(1, numprocs-1):
106     comm_cart.Gatherv([u_part_aux[M, 1:-1], N_x_part*(N_y+1), MPI.DOUBLE],
107                       None, root=0)
108
109 elif rank_cart == numprocs-1:
110     comm_cart.Gatherv([u_part_aux[M, 1:], N_x_part*(N_y+1), MPI.DOUBLE],
111                       None, root=0)
112
113 if rank_cart == 0:
114     print(f"Elapsed time is {end_time - start_time:.4f} sec")
115     savez("results", x=x, y=y, u_T=u_T)
```

Обратите внимание, что мы спланировали алгоритм таким образом, что передаваемые массивы расположены в памяти непрерывно (индекс `j` в массиве `u_part_aux` – последний). В противном случае массивы `rcounts` и `displs` были бы намного сложнее.

Параллельный алгоритм решения в случае двумерного деления пространственной области на блоки

Предыдущий алгоритм был лишь обобщением одномерного по пространству алгоритма, рассмотренного в лекции 8. Попробуем более строго учесть то, что область по пространству двумерная. Будем делить область на двумерные блоки. В дальнейшем это позволит нам использовать декартову топологию типа ”сетка”.

Схему разбиения можно посмотреть на рис. 10.4, обмен данными между соседними процессами показан бледно-серыми квадратами.

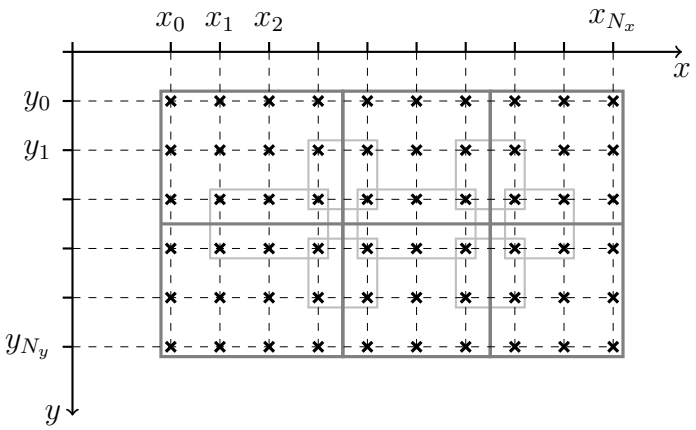


Рис. 10.4: Схема разделения данных по MPI процессам, двумерный случай

Оценим объём пересылаемой информации на каждой итерации по времени. Пусть $N_x = N_y = N$, n – количество процессов. Вводим топологию одинаково по x и по y, то есть предполагаем, что корень из n извлекается. Тогда длина каждого блока – $\frac{N}{\sqrt{n}}$. Так как мы на каждой итерации передаём и получаем по 4 сообщения, полученный объём – $\frac{8N}{\sqrt{n}}$. Напомним, что в предыдущем алгоритме мы передавали по 2 сообщения длины N, то есть объём передаваемой информации равнялся $4N$.

В новом подходе объём передаваемых сообщений будет уменьшаться при увеличении числа процессов.

Программная реализация параллельного алгоритма решения

Будем считать, что число процессов – квадрат натурального числа больше единицы.

```
from mpi4py import MPI
from numpy import empty, int32, float64, linspace, tanh, sqrt, savez

comm = MPI.COMM_WORLD
numprocs = comm.Get_size()

num_row = num_col = int32(sqrt(numprocs))

comm_cart = comm.Create_cart(dims=(num_row, num_col),
                             periods=(False, False), reorder=True)
rank_cart = comm_cart.Get_rank()

...
```

После введения виртуальной топологии снова задаём все параметры задачи (строки 4-17, 21-27 в последовательной программе), а также сохраняем начальное время и задаём функцию для вычисления вспомогательных массивов (строки 11-23 в предыдущей параллельной программе).

Создаём вспомогательные массивы, в этот раз уже по двум пространственным переменным:

```
14 rcounts_N_x, displs_N_x = auxiliary_arrays_determination(N_x+1, num_col)

15 rcounts_N_y, displs_N_y = auxiliary_arrays_determination(N_y+1, num_row)
```

Далее определим координаты текущего процесса внутри топологии и опередим размер и сдвиги частей массива u.

```
16 my_row, my_col = comm_cart.Get_coords(rank_cart)
17
18 N_x_part = rcounts_N_x[my_col]
19 N_y_part = rcounts_N_y[my_row]
20
21 if my_col in [0, num_col-1]:
22     N_x_part_aux = N_x_part + 1
23 else:
24     N_x_part_aux = N_x_part + 2
25
26 if my_row in [0, num_row-1]:
27     N_y_part_aux = N_y_part + 1
28 else:
29     N_y_part_aux = N_y_part + 2
30
31 displs_N_x_aux = displs_N_x - 1
32 displs_N_x_aux[0] = 0
33
34 displs_N_y_aux = displs_N_y - 1
35 displs_N_y_aux[0] = 0
36
37 displ_x_aux = displs_N_x_aux[my_col]
38 displ_y_aux = displs_N_y_aux[my_row]
39
40 u_part_aux = empty((M+1, N_x_part_aux, N_y_part_aux), dtype=float64)
```

Вычисляем и задаём начальные и граничные условия:

```
41 for i in range(N_x_part_aux):
42     for j in range(N_y_part_aux):
43         u_part_aux[0, i, j] = u_init(x[displ_x_aux+i], y[displ_y_aux+j])
44
45 for m in range(1, M+1):
```

```
46      for j in range(1, N_y_part_aux-1):
47          if my_col == 0:
48              u_part_aux[m, 0, j] = u_left([displ_y_aux+j], t[m])
49          if my_col == num_col-1:
50              u_part_aux[m, -1, j] = u_right([displ_y_aux+j], t[m])
51
52      for i in range(N_x_part_aux):
53          if my_row == 0:
54              u_part_aux[m, i, 0] = u_bottom([displ_x_aux+i], t[m])
55          if my_row == num_row-1:
56              u_part_aux[m, i, -1] = u_top([displ_x_aux+i], t[m])
```

Приступаем к основной части программы. Вычисления на каждой итерации почти не поменяются (изменится только диапазон индексов по y):

```
57  for m in range(M):
58
59      for i in range(1, N_x_part_aux-1):
60          for j in range(1, N_y_part_aux-1):
61              d2x = (u_part_aux[m, i+1, j] -
62                     2*u_part_aux[m, i, j] +
63                     u_part_aux[m, i-1, j]) / h_x**2
64              d2y = (u_part_aux[m, i, j+1] -
65                     2*u_part_aux[m, i, j] +
66                     u_part_aux[m, i, j-1]) / h_y**2
67
68              d1x = (u_part_aux[m, i+1, j] -
69                     u_part_aux[m, i-1, j]) / (2*h_x)
70              d1y = (u_part_aux[m, i, j+1] -
71                     u_part_aux[m, i, j-1]) / (2*h_y)
72
73              u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
74                                      tau*(eps*(d2x + d2y) +
75                                      u_part_aux[m, i, j]*(d1x + d1y) +
76                                      u_part_aux[m, i, j]**3)
```

А вот передача сообщений изменится существенно. В лекции 6 для определения соседей мы использовали функцию `Shift`. Здесь, для разнообразия, мы выберем другой подход. Так как мы знаем, сколько в топологии строк и столбцов (а нумерация процессов идет последовательно), мы можем вычислить номер соседнего процесса вручную.

Вначале передадим данные процессам налево и направо. Эта операция уже выполнялась, так что затруднений возникнуть не должно.

```
77      if my_col > 0:
78          comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1],
79                                      N_y_part_aux, MPI.DOUBLE],
```

```
80                              dest=my_row*num_col + (my_col-1),
81                              sendtag=0,
82                              recvbuf=[u_part_aux[m+1, 0],
83                                       N_y_part_aux, MPI.DOUBLE],
84                              source=my_row*num_col + (my_col-1),
85                              recvtag=MPI.ANY_TAG,
86                              status=None)
87
88      if my_col < num_col-1:
89          comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, -2],
90                                      N_y_part_aux, MPI.DOUBLE],
91                             dest=my_row*num_col + (my_col+1),
92                             sendtag=0,
93                             recvbuf=[u_part_aux[m+1, -1],
94                                      N_y_part_aux, MPI.DOUBLE],
95                             source=my_row*num_col + (my_col+1),
96                             recvtag=MPI.ANY_TAG,
97                             status=None)
```

С операциями пересылки вверх и вниз могут возникнуть сложности. При извлечении среза по среднему индексу пересчёт индексов считается неочевидно. Чтобы избежать трудностей, будем просто копировать данные во временные массивы и пересылать их содержимое. Если этого не сделать, данные не будут храниться непрерывно в памяти.

```
98      if my_row > 0:
99          temp_array_send = u_part_aux[m+1, :, 1].copy()
100         temp_array_recv = empty(N_x_part_aux, dtype=float64)
101
102         comm_cart.Sendrecv(sendbuf=[temp_array_send,
103                                     N_x_part_aux, MPI.DOUBLE],
104                            dest=(my_row-1)*num_col + my_col,
105                            sendtag=0,
106                            recvbuf=[temp_array_recv,
107                                     N_x_part_aux, MPI.DOUBLE],
108                            source=(my_row-1)*num_col + my_col,
109                            recvtag=MPI.ANY_TAG,
110                            status=None)
111
112         u_part_aux[m+1, :, 0] = temp_array_recv
113
114     if my_row < num_row-1:
115         temp_array_send = u_part_aux[m+1, :, -2].copy()
116         temp_array_recv = empty(N_x_part_aux, dtype=float64)
117
118         comm_cart.Sendrecv(sendbuf=[temp_array_send,
119                                     N_x_part_aux, MPI.DOUBLE],
120                            dest=(my_row+1)*num_col + my_col,
```

```
                                sendtag=0,
                                recvbuf=[temp_array_recv,
                                         N_x_part_aux, MPI.DOUBLE],
                                source=(my_row+1)*num_col + my_col,
                                recvtag=MPI.ANY_TAG,
                                status=None)

        u_part_aux[m+1, :, -1] = temp_array_recv
```

После основной части измеряем финальное время и собираем значения функции в финальный момент времени. Чтобы отправляемые данные располагались в памяти непрерывно, отправляем столбцы с каждого процесса по очереди. Вообще говоря, можно реализовать этот блок по-другому, более эффективно.

```
if rank_cart == 0:
    end_time = MPI.Wtime()
    print(f"Elapsed time is {end_time - start_time:.4f} sec")

    u_T = empty((N_x+1, N_y+1), dtype=float64)

    for row in range(num_row):
        for col in range(num_col):
            if row == col == 0:
                for i in range(N_x_part):
                    u_T[i, :N_y_part] = u_part_aux[M, i, :N_y_part]
            else:
                for i in range(rcounts_N_x[col]):
                    comm_cart.Recv([u_T[displs_N_x[col]+i,
                                        displs_N_y[row]:],
                                    rcounts_N_y[row], MPI.DOUBLE],
                                   source=row*num_col+col,
                                   tag=0,
                                   status=None)

    savez(x=x, y=y, u_T=u_T)

else: # rank_cart != 0
    for i in range(N_x_part):
        if my_row == 0:
            comm_cart.Send([u_part_aux[M, 1+i], N_y_part, MPI.DOUBLE],
                           dest=0, tag=0)

        else:
            if my_col == 0:
                comm_cart.Send([u_part_aux[M, i, 1:],
                                N_y_part, MPI.DOUBLE],
                               dest=0, tag=0)
```

```
162                 else:
163                     comm_cart.Send([u_part_aux[M, 1+i, 1:],
164                                     N_y_part, MPI.DOUBLE],
165                                    dest=0, tag=0)
```

Обсуждение эффективности двух рассмотренных параллельных программных реализаций

Тестирование проводилось на суперкомпьютере ”Ломоносов-2”. Время работы последовательной программы на 1 ядре – 791 с. (для параметров $N_x = N_y = 200$, $M = 4000$, также было изменено значение параметра $T = 4$).

На рис. 10.5 приведён график времени работы двух подходов, на рис. 10.6 – график ускорения. Видно, что первый подход на 100 ядрах выходит на плато, так как время на коммуникацию остаётся постоянным. Во втором подходе, в отличие от первого, время продолжает убывать.

Также можно заметить, что при малом количестве ядер ускорение близко к линейному, а потом заметно проседает. Так происходит из-за того, что мы выходим за границы одного узла.

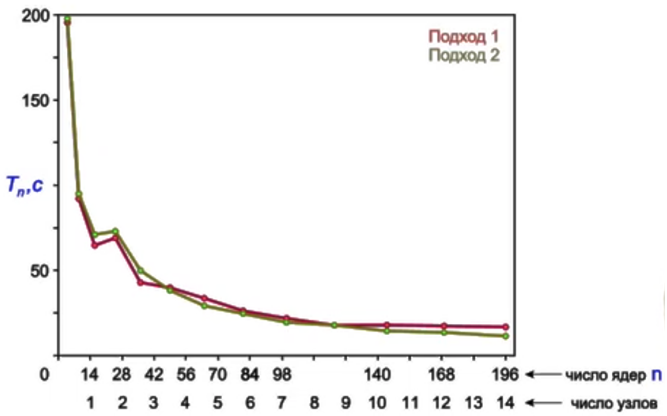


Рис. 10.5: График времени работы параллельных программ

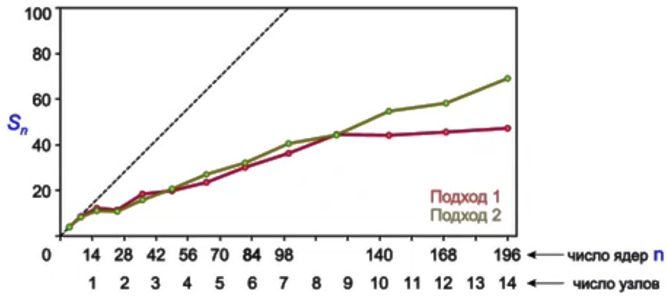


Рис. 10.6: График ускорения параллельных программ

На рис. 10.7 можно увидеть график эффективности. Напомним, что значения эффективности распараллеливания в интервале $[0.8, 1]$ говорят о удачной программной реализации. В нашем случае эффективность близка к 0.4, что для программы, написанной за одну лекцию, довольно неплохо.

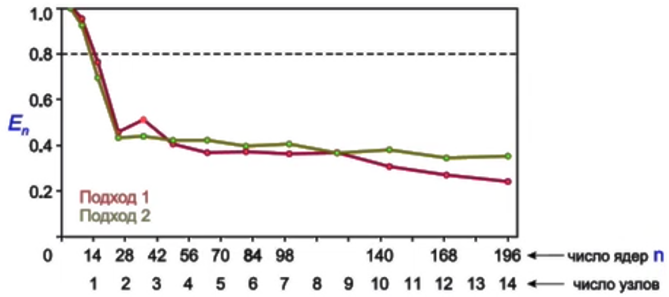


Рис. 10.7: График эффективности работы параллельных программ

Лекция 11. Асинхронные операции

На этой и следующей лекциях мы обратим внимание на особенности использования асинхронных операций для приема-передачи сообщений между различными MPI-процессами. Часть этих асинхронных операций на следующей лекции мы используем для того, чтобы достаточно существенно повысить эффективность программных реализацией тех параллельных алгоритмов, которые мы рассматривали на предыдущих лекциях.

Необходимость использования асинхронных операций

Рассмотрим простой пример, на разборе нюансов которого, мы обсудим особенности использования асинхронных операций. Предположим, что у нас есть некоторый набор MPI-процессов и каждому из них дадим следующее задание: пусть каждый MPI-процесс посылает свой идентификатор другому MPI-процессу, у которого ранг на единичку меньше текущего процесса, и получает соответствующее сообщение от процесса, у которого идентификатор на единичку больше. В принципе задача очень простая, но она симулирует ту ситуацию, с которой мы постоянно встречались, начиная с 6 лекции — нам все время нужно было реализовывать операции приема-передачи сообщений между различными процессами. Например, у нас процессы были в топологии линейка, или в топологии кольца, или двухмерная сетка, или двухмерный тор, и во многих алгоритмах, которые мы рассматривали, каждый MPI-процесс должен был передавать сообщения одному из своих соседей и получать сообщения от одного из них. Этот пример как раз и моделируют соответствующую ситуацию, поэтому давайте посмотрим на ее программную реализацию.

Попробуем реализовать этот пример, когда в топологии кольца каждый MPI-процесс передает свой ранг одному из соседей (для определенности левому) и получает ранг соседнего процесса (для определенности правого). Для этого, как обычно, пишем наши ритуальные функции

```
from mpi4py import MPI
from numpy import  array, int32

comm = MPI.COMM_WORLD
numprocs = comm.Get_size()
rank = comm.Get_rank()
```

коммуникатор MPI.COMM_WORLD содержит информацию о всех процессах, на которых запущена наша программа; каждый процесс по итогу вызова comm.Get_size() знает число процессов, участвующих в вычислениях, и каждый процесс по итогу вызова comm.Get_rank() знает свой идентификатор в этом коммуникаторе, то есть свой ранг.

Далее выделим место в памяти под два массива, каждый из которых будет содержать по одному числу

```
    a = array(rank, dtype=int32); b = array(100, dtype=int32)
```

массив a будет содержать ранг процесса, вместо массива b можно было бы формально выделить место в памяти под одно число, куда будет записываться принятое сообщение, используя, например, функцию empty. Однако здесь выделяется место в памяти под одно число и сразу в это место кладется 100. В процессе обсуждения соответствующих новых для нас функций мы будем рассматривать типовые ошибки, например, когда сообщение не принято или какие-то еще сбои, поэтому, чтобы легче было понять принято или не принято, выведено ли то сообщение, которое нам нужно, мы не просто выделили место в памяти, а положили 100, так что, если мы будем получать в качество ответа 100, мы будем знать, что произошел какой-то сбой.

```
if rank == 0:
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=numprocs-1, tag=0)
elif rank == numrocs - 1 :
    comm.Recv([b, 1, MPI.INT], source=0, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=numprocs-2, tag=0)
else :
    comm.Recv([b, 1, MPI.INT], source=rank+1, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=rank-1, tag=0)

print('I, process with rank {}, got number {}'.format(rank, b))
```

Каждый процесс выставляет команду comm.Recv, чтобы получить сообщение от процесса, у которого ранг на единицу больше, чем его собственный, а также записать это сообщение в область памяти, ассоциированной с массивом b, и выставляется команда comm.Send, чтобы послать процессу, у которого ранг на единичку меньше, сообщение, состоящее из одного элемента, который взят из области памяти, ассоциированной с массивом a, то есть фактически посылается ранг.

Здесь понятно, что процесс с рангом 0 получает сообщение от процесса с рангом 1, а посылает соответствующее сообщение последнему процессу с рангом $numprocs - 1$. Напомним, число процессов $numprocs$, нумерация от 0 до $numprocs - 1$. Последний процесс

получает сообщение от 0-го, так как у нас топология типы кольца, а передает сообщение процессу с номером $numprocs - 2$. Таким образом, на каждом процессе сначала должна быть выполнена команда Recv, чтобы получить сообщение, а затем — команда Send, чтобы послать соответствующее сообщение. После этого будет вывод print, в котором указывается, какое число получил каждый процесс.

Если мы попытаемся запустить скрипт 'mpiexec -n 4 python script.py', то ожидаем, что каждый из 4-х MPI-процессов выведет число, которое на 1 больше его ранга, однако этого не произойдет, программа зависнет. Напомним, как работают функции Send и Recv, они являются блокирующими, как только мы выставили команду Recv, выход из этой функции будет осуществлен только в том случае, когда от соответствующего процесса реально придет сообщение и будет записано в область памяти, ассоциированной с выделенным для этого массивом. Получается, что абсолютно все процессы выставили команду Recv, сообщение ни от кого не приходит (команда Send еще не сработала), поэтому выход из этой функции не произошел.

Как можно выкрутиться из этой ситуации? Например, если бы у нас обмен сообщениями такого типа был не в топологии кольца, а в топологии линейка

```
if rank == 0:
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)

elif rank == numrocs - 1 :

    comm.Send([a, 1, MPI.INT], dest=numprocs-2, tag=0)
else :
    comm.Recv([b, 1, MPI.INT], source=rank+1, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=rank-1, tag=0)

print('I, process with rank {}, got number {}'.format(rank, b))
```

нулевой процесс налево послать сообщение не может, так как в топологии линейка нет зацикливания, так же и последнему процессу не от кого получить сообщение по той же причине.

Теперь, если запустить этот скрипт 'mpiexec -n 4 python script.py', то он отработает, и мы получим

```
    I, process with rank 3, got number 100
    I, process with rank 2, got number 3
    I, process with rank 1, got number 2
    I, process with rank 0, got number 1
```

Понятно, что процесс с рангом 3 ничего ничего не получает, он выводит текущее значение в массиве b, которое равно 100.

Несмотря на то, что все отработало, проблема осталась. Пусть, для наглядности, мы запустили программу на большом числе процессов, из них только последний выставил сначала команду Send, остальные ждут приема сообщения, сначала его получит предпоследний, выйдет из Recv и отправит сообщение следующему и так далее до нулевого процесса, при этом, когда пара процессов обменивается сообщениями, остальные простаивают, именно по этой причине порядок вывода начинается с последнего и оканчивается нулевым.

Рассмотрим снова организацию приема-передачи в топологии кольца, но поменяем местам Send и Recv местами на последнем процессе:

```
if rank == 0:
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=numprocs-1, tag=0)
elif rank == numrocs - 1 :
    comm.Send([a, 1, MPI.INT], dest=numprocs-2, tag=0)
    comm.Recv([b, 1, MPI.INT], source=0, tag=0, status=None)
else :
    comm.Recv([b, 1, MPI.INT], source=rank+1, tag=0, status=None)
    comm.Send([a, 1, MPI.INT], dest=rank-1, tag=0)

print('I, process with rank {}, got number {}'.format(rank, b))
```

Запустив скрипт ’mpiexec -n 4 python script.py’, получим:

```
    I, process with rank 2, got number 3
    I, process with rank 1, got number 2
    I, process with rank 0, got number 1
    I, process with rank 3, got number 0
```

Однако и здесь вывод предопределен, весь процесс приема-передачи будет происходить последовательно по «цепочке», начиная с предпоследнего процесса, двигаясь по кругу, заканчивая последним.

Заметим также, что при следующей ситуации

```
if rank == 0:
    comm.Send([a, 1, MPI.INT], dest=numprocs-1, tag=0)
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)
elif rank == numrocs - 1 :
    comm.Send([a, 1, MPI.INT], dest=numprocs-2, tag=0)
```

```
    comm.Recv([b, 1, MPI.INT], source=0, tag=0, status=None)
else :
    comm.Send([a, 1, MPI.INT], dest=rank-1, tag=0)
    comm.Recv([b, 1, MPI.INT], source=rank+1, tag=0, status=None)

print('I, process with rank {}, got number {}'.format(rank, b))
```

если передаваемое сообщение достаточно мало, чтобы оно поместилось во внутренний буфер, то выход из Send произойдет сразу, и программа заработает, при этом все операции действительно будут происходить одновременно, в противном случае — нет. Поэтому и необходимы асинхронные операции, позволяющие обходить эти моменты.

Асинхронные операции

Напомним аргументы, входящие в Send

```
    comm.Send([a, 1, MPI.INT], dest=numprocs-1, tag=0)
```

первый аргумент — структура передаваемого сообщения: она обращается к началу области памяти, ассоциированной с массивом a, и, начиная с этой области памяти, берет один элемент типа integer, это сообщение должно быть послано процессу с идентификатором dest, идентификатор сообщения tag=0 (сообщений к этому процессу может быть несколько, их нужно различать). Асинхронная версия этой функции

```
    requests = comm.Isend([a, 1, MPI.INT], dest=numprocs-1, tag=0)
```

аргументы все те же самые, но добавляется переменная *requests* предопределенного типа данных MPI_Request, которая содержит информацию о том, завершилась ли реально передача сообщения или нет. Выход из этой операции происходит сразу. Аналогично с Recv

```
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)
```

здесь появляется дополнительный возвращаемый параметр *status*, которого у нас нет, но при желании мы могли бы завести переменную предопределенного типа данных MPI_Status, что мы обсуждали на первой лекции, она содержит информацию о полученном сообщении: от какого процесса, с каким идентификатором пришло это сообщение и количество элементов в нем. Асинхронный аналог

```
    requests =  comm.Irecv([b, 1, MPI.INT], source=1, tag=0)
```

Существенное отличие заключается в отсутствии аргумента *status*, так как выход из этой функции происходит сразу, то мы не знаем получено сообщение или нет, поэтому переменную *status* заполнять бессмысленно

Заменим теперь обычные Send, Recv их асинхронными аналогами

```
requests = [MPI_Request() for i in range(2)]

if rank == 0:
    requests[0] =   comm.Isend([a, 1, MPI.INT], dest=numprocs-1, tag=0)
    requests[1] =   comm.Irecv([b, 1, MPI.INT], source=1, tag=0)

elif rank == numrocs - 1 :
    requests[0] =   comm.Isend([a, 1, MPI.INT], dest=numprocs-2, tag=0)
    requests[1] =   comm.Irecv([b, 1, MPI.INT], source=0, tag=0)
else :
    requests[0] =   comm.Isend([a, 1, MPI.INT], dest=rank-1, tag=0)
    requests[1] =   comm.Irecv([b, 1, MPI.INT], source=rank+1, tag=0)

print('I, process with rank {}, got number {}'.format(rank, b))
```

Необходимо заранее выделить место в памяти под массив размера 2, так как на каждом MPI-процессе будет вызов двух асинхронных операций. После запуска на четырех процессах получим

```
    I, process with rank 2, got number 100
    I, process with rank 0, got number 100
    I, process with rank 1, got number 100
    I, process with rank 3, got number 100
```

При перезапуске строки будут меняться местами, значит все выполняется одновременно, но вывод неверный, мы ожидаем получить числа на единицу больше ранга соответствующего процесса, кроме третьего, он должен вывести 0.

Почему так произошло? Все процессы сначала вызвали команду Isend, чтобы отправить сообщение другому процессу, сразу из этой функции вышли, приступили к выполнению команды Irecv, чтобы записать полученное сообщение в область памяти, ассоциированную с массивом b, сразу произошел выход, после чего все процессы перешли к команде print. Мы видим, что вывод произошел до того, как посланные сообщения дошли до процессов.

Чтобы избежать подобного исхода, добавим блокирующую операцию перед print:

```
MPI.Request.Waitall(requests, statuses=None)
print('I, process with rank {}, got number {}'.format(rank, b))
```

Фактически она приостанавливает работу программы до тех пор, пока в *requests* не появится информация о том, что обмен сообщений произошел. Вместо MPI.Request.Waitall(requests, statuses=None) можно было написать MPI.Request.Wait(requests[1], statuses=None), программа приостановится до тех пор, пока не будет получена о том, что было реально получено сообщение. В любом случае, после запуска скрипта получим

```
    I, process with rank 2, got number 3
    I, process with rank 0, got number 1
    I, process with rank 1, got number 2
    I, process with rank 3, got number 0
```

Есть и другие команды по приостановке программы. На практике почти всегда выгоднее использовать асинхронные функции, так как на их фоне могут происходить вычислительные операции, что зачастую может понизить накладные расходы и повысить эффективность программной реализации.

Однако использование асинхронных операций требует большей внимательности, если в предыдущую программу добавить вычислительную задачу, например к a добавить 10, после отправки сообщения, то поставив эту операцию не в том месте

```
    a += 10

MPI.Request.Waitall(requests, statuses=None)
print('I, process with rank {}, got number {}'.format(rank, b))
```

получим

```
    I, process with rank 1, got number 12
    I, process with rank 0, got number 11
    I, process with rank 3, got number 0
    I, process with rank 2, got number 3
```

То есть второй и третий отработали правильно, а нулевой и первый нет, для того, что бы добавление 10 было в нужный момент, можно написать

```
    MPI.Request.Wai(requests[0], statuses=None)
    a += 10
```

```
MPI.Request.Waitall(requests, statuses=None)
print('I, process with rank {}, got number {}'.format(rank, b))
```

тогда вычисление будет производится уже гарантированно после отправки сообщения.

Фактически сейчас мы реализовали самостоятельно команду Sendrecv, которую использовали с 6-ой лекции. Внутри она реализована как раз с помощью использования асинхронных операций. Если какая-то команда уже реализована, то лучше использовать ее, так как она уже отлажена, никаких ошибок нет и, скорее всего, она работает лучше.

Использование асинхронных операций можно проводить совместно с блокирующими, например, на том же самом примере, в топологии типа линейка, мы можем написать

```
requests = [MPI_Request() for i in range(2)]

if rank == 0:
    comm.Recv([b, 1, MPI.INT], source=1, tag=0, status=None)
elif rank == numrocs - 1 :
    comm.Send([a, 1, MPI.INT], dest=numprocs-2, tag=0)
else :
    requests[0] =   comm.Isend([a, 1, MPI.INT], dest=rank-1, tag=0)
    requests[1] =   comm.Irecv([b, 1, MPI.INT], source=rank+1, tag=0)
    MPI.Request.Waitall(requests, statuses=None)

print('I, process with rank {}, got number {}'.format(rank, b))
```

На нулевом процессе не имеет смысла использовать асинхронный Irecv, он ничего не посылает, пусть ждет сообщения, аналогично, на последнем можно использовать обычный Send. На промежуточных процессах использование асинхронных операций приводит к тому, что обмен сообщения происходит одновременно между процессами.

Подытожим, начиная с 6-ой лекции активно использовались операции Sendrecv и Sendrecv_replace, которые позволяли реализовать обмен сообщениями со своими соседями, и, как говорилось, мы их используем, потому что у обычных Send и Recv есть свои особенности, на этой лекции мы их и рассмотрели. Те проблемы, которые возникают при их использовании можно решить с помощью асинхронных операций. При наличии встроенных в пакет функций, конечно, нужно использовать их.

Лекция 12. Отложенные запросы на взаимодействие

На этой лекции мы продолжим обсуждать особенности работы с асинхронными операциями и, в частности, обсудим новый тип асинхронных операций, который существенным образом позволит понизить накладные расходы, связанные с приемом-передачей сообщений между различными MPI-процессами

Применение асинхронных операций к одному из разобранных примеров

Начнем обсуждение с примера, который мы рассматривали на 6 лекции, когда впервые знакомились с виртуальными топологиями. Напомним, у нас есть декартова двухмерная сетка MPI-процессов

1 2	2 3	3 4	4 5
2 4	4 6	6 8	8 10

на каждом процессе есть вектор из двух элементов, наша цель реализовать сложение элементов, находящийся в каждой строке этой декартовой сетки, полученный результат сложения этих элементов в строке должен быть на каждом MPI-процессе. Листинг программы

```
1  from mpi4py import MPI
2  from numpy import array, int32
3
4  comm = MPI.COMM_WORLD
5  numprocs = comm.Get_size()
6
7  num_row = 2; num_col = 4
8
9  comm_cart = comm.Create_cart(dims=(num_row, num_col),
10                              periods=(True, True), reorder=True)
11
12 rank_cart = comm_cart.Get_rank()
13
```

```
14 neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
15
16 a = array([(rank_cart % num_col + 1 + i)*2**(rank_cart // num_col)
17               for i in range(2)], dtype=int32)
```

Сначала определяем коммуникатор MPI.COMM_WORLD, который содержит информацию о всех процессах, на которых запущена наша программа; каждый процесс по итогу вызова comm.Get_size() знает число процессов, участвующих в вычислениях. В 7-ой строке определяем число строк и столбцов виртуальной декартовой топологии, которую создаем с помощью функции Create_cart, она фактически является двухмерной сеткой. При этом соответствующую декартову топологию мы делаем периодической по каждому из направлений. Напомним, если у нас есть такая декартова двухмерная сетка, то в случае включения периодичности только по одному направлению она превращается в цилиндр, а в случае включения периодичности по обоим направлениям — в двухмерный тор, это обсуждалось в начале 6 лекции.

Также используем переопределение процессов, это нужно для того, чтобы процессы, которые физически могут быть раскиданы совершенно произвольным образом в нашей реальной физической коммуникационной среде, были перенумерованны системой таким образом, чтобы они и физически располагались рядом друг с другом в соответствии с этой топологией.

После этого, в 9-ой строке, определяем внутри этой топологии ранг процесса, который в случае вышеописанного переопределения, может отличаться от ранга процесса в нашем исходном коммуникаторе.

Дальше с помощью команды Shift вдоль первого направления определяем левого и правого соседа для текущего процесса, который выполняет эту программу. Напомним, если бы мы поставили direction=0, то определили бы соседей сверху и снизу. Затем произвольным образом определим значение массива из двух элементов, которые содержатся на каждом MPI-процессе.

Затем запоминаем текущее значение вектора a на каком-то MPI-процессе, организуем цикл

```
18 summa = a.copy()
19 for n in range(num_col-1) :
20     comm_cart.Sendrecv_replace([a, 2, MPI.INT],
21                     dest=neighbour_right, sendtag=0,
22                     source=neighbour_left, recvtag=MPI.ANY_TAG,
23                     status=None)
24     summa = summa + a
25
```

```
26 print('Process {} has summa={}'.format(rank_cart, summa))
```

каждый MPI-процесс передает значение вектора a своему правому соседу, а от левого получает соответствующее сообщение, записывая его на место вектора a. После того, как это сообщение принято, сумма увеличивается, поэтому, проделывая цикл по числу столбцов виртуальной декартовой топологии минус один, получим, что на каждом MPI-процессе будет содержаться результат суммирования всех векторов вдоль этого направления. Запустим скрипт на 8 процессах 'mpiexec -n 8 python script.py'

```
Process 0 has summa=[10 14]
Process 1 has summa=[10 14]
Process 2 has summa=[10 14]
Process 3 has summa=[10 14]
Process 4 has summa=[20 28]
Process 5 has summa=[20 28]
Process 6 has summa=[20 28]
Process 7 has summa=[20 28]
```

Таким образом, мы реализовали функцию Allreduce, которая складывала какие-то промежуточные вектора, а результат сложения этих векторов с различных MPI-процессов записывался на всех MPI-процессах? на которых запускалась эта команда.

Теперь можно сделать следующее, давайте команду Sendrecv_replace реализуем сами с помощью асинхронных Isend и Irecv

```
10 ...
11
12 requests = [MPI_Request() for i in range(2)]
13 a_recv = empty(2, dtype=int32)
14
15 summa = a.copy()
16 for n in range(num_col-1) :
17    requests[0] = comm.Isend([a, 2, MPI.INT], dest=neighbour_right, tag=0)
18    requests[1] = comm.Irecv([a_recv, 2, MPI.INT], source=neighbour_left,
    tag=0)
19    MPI.Request.Waitall(requests, statuses=None)
20    a = a_recv
21    summa = summa + a
22
23 print('Process {} has summa={}'.format(rank_cart, summa))
```

асинхронный Isend говорит, что нужно обратиться к области памяти, ассоциированной с массивом a, с начала этой области памяти взять два элемента и послать процессу, который в виртуальной декартовой топологии располагается справа от текущего процесса.

Выход из этой операции произойдет сразу, независимо ушло сообщение или нет. После этого вызывается асинхронная команда Irecv, которая говорит, что полученные сообщения должно быть записано в область памяти, ассоциированный с массивом a_recv, именно по этой причине, так как это промежуточный буфер, под него выделили место в памяти (строка 13).

Для выполнения последующих операций, нужно быть уверенным, что сообщения дошли, поэтому ставим Request.Waitall, а затем переопределяем переменную a, считаем сумму. Заметим, что в этом случае нужно ставить именно Request.Waitall, а не Request.Wait, так как сообщение от левого соседа уже может дойти до текущего процесса, но с него еще не было отправлено сообщение правому соседу. Таким образом, мы организовали MPI-функцию Sendrecv_replace с помощью асинхронных Isend и Irecv. Однако если есть готовая функция, зачем мы это делали?

Необходимость отложенных запросов на взаимодействие

Если нужно реализовать Sendrecv_replace только один раз, тогда рационально и конструктивно использовать не асинхронные операции, написанное вручную, а именно готовую реализацию Sendrecv_replace. Если число MPI-процессов и сетка, которая определяет нашу виртуальную декартову топологию, достаточно большие, то в цикле команды Isend и Irecv вызываются огромное число раз, идет инициализация посылки однотипных сообщений, которые берут информацию из одной и той же области памяти (вот в этой реализации это не совсем так, но имеется ввиду общий случай), вектора одних и тех же размеров, которые посылаются с текущего процесса другому процессу, одному и тому же. То есть какие-то операции выполняются однотипно и много-много раз. В алгоритмах решения задач в частных производных, например, операции выполняются в массивах куда большей размерности.

Здесь на помощь приходят так называемые отложенные запросы на взаимодействия. Идея в том, что вместо того, чтобы в цикле вызывать команду, например, Send или Recv, в частности, асинхронные, при которых идет инициализация посылки сообщений, взаимодействие с коммуникационным оборудованием и только потом непосредственно физическая посылка сообщения, лучше сначала описать параметры соответствующего сообщения, фактически инициализировать посылку сообщений, а саму физическую посылку вызывать тогда, когда нам нужно.

```
24 ...
25
26 requests = [MPI_Request() for i in range(2)]
27 a_recv = empty(2, dtype=int32)
```

```
28
29 requests[0] = comm_cart.Send_init([a, 2, MPI.INT], dest=neighbour_right,
     tag=0)
30 requests[1] = comm_cart.Recv_init([a_recv, 2, MPI.INT], source=
     neighbour_left, tag=0)
31
32 summa = a.copy()
```

Send_init и Recv_init это так называемые отложенные запросы на взаимодействие, никаких посылок и принятий сообщений здесь не происходит. Здесь подготавливаются соответствующие структуры данных, которые указывают параметры сообщения, которое будет передано (в случае Send_init) или получено (в случае Recv_init). В частности, для Send_init описывается, что сообщение, которое будет посылаться соседнему правому процессу, будет находиться в месте памяти, ассоциированным с массивом a, из этого места в памяти нужно всегда будет брать два элемента.

```
33 ...
34
35 for n in range(num_col-1) :
36 MPI.Prequest.Startall(requests)
37 MPI.Request.Waitall(requests, statuses=None)
38 a = a_recv
39 summa = summa + a
40
41 print('Process {} has summa={}'.format(rank_cart, summa))
```

Здесь появляется новая команда MPI.Prequest.Startall(requests), которая вызывает и осуществляет асинхронную посылку всех сообщений, ассоциированных идентификаторами, которые содержатся в requests, и их асинхронный прием. Если бы нам потребовалось запустить только посылку сообщений, следовало бы написать MPI.Prequest.Start(requests[0]). После этого, конечно, нужно вызвать команду Waitall.

Здесь есть один нюанс, связанный с работой самого python. Если запустить этот скрипт, то посчитанные суммы будут отличаться от того, что должно было получится. Это связано с ссылочной моделью python. Проследим, что делает python при запуске программы на первом процессе. При выполнении команды Startall нужно обратиться к области памяти, ассоциированной с массивом a, взять из этой области памяти 2 числа и послать правому соседу. Соответствующие данные с первого процесса ушли второму, а от нулевого процесса первому пришли два числа и были записаны в область памяти, ассоциированных с массивом a_recv.

Далее, команда a=a_recv, она означает, что теперь a ссылается не на объект, который был ассоциирован с тем местом памяти, откуда берет данные команда Send_init, а ссыла-

ется на a_recv, но на следующей итерации цикла команда Send_init будет обращаться к старому месту в памяти, где a не изменилось, аналогично и Rend_init. В итоге на всех итерациях будет посылаться одно и тоже сообщение. Чтобы этого избежать, нужно написать

```
...

for n in range(num_col-1) :
MPI.Prequest.Startall(requests)
MPI.Request.Waitall(requests, statuses=None)
a[:] = a_recv
summa = summa + a

print('Process {} has summa={}'.format(rank_cart, summa))
```

Двоеточие в квадратных скобках означает, что мы в область памяти, ассоциированной с массивом a записали массив a_recv, поэтому сейчас все будет работать правильно.

Отложенные запросы на взаимодействие позволяют снизить накладные расходы на взаимодействие между процессами по приему-передачи сообщений, потому что та часть действий, которая выполнялась бы в цикле многократно фактически ”вынеслась из цикла поэтому сделается один раз, а не много. Это имеет смысл только в том случае, когда однотипные сообщения, которые берут информацию из одной и той же области памяти (это важно) и высылают ее одному и тому же процессу, вызываются многократно в каких-то циклах. Если же прием-передача сообщения однократная, то от этих функций смысла никакого не будет, они только замедлят работу программы.

Пример 12.1, модификация примера 10.2

Вернемся к материалу 10-ой лекции, мы рассматривали задачи для уравнения в частных производных двухмерную по пространству. Напомним, что в 10 лекций (рекомендуется сначала обязательно ознакомится с той лекцией), пример 10.2, основной вычислительной частью был двойной цикл

```
for m in range(M) :

    for i in range (1 N_x_part_aux - 1) :
        for j in range (1 N_y_part_aux - 1) :
            u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
                tau*(eps*((u_part_aux[m, i+1, j] - 2*u_part_aux[m, i, j] +
    u_part_aux[m, i-1, j])/h_x**2 +
                          (u_part_aux[m, i, j+1] - 2*u_part_aux[m, i, j] +
    u_part_aux[m, i, j-1])/h_y**2) +
```

```
111                     u_part_aux[m, i, j]*((u_part_aux[m, i+1, j] -
    u_part_aux[m, i-1, j])/(2*h_x) +
112                                 (u_part_aux[m, i, j+1] - u_part_aux[m, i, j
    -1])/(2*h_y)) +
113                     u_part_aux[m, i, j**3])
114
```

Для того, чтобы какой-нибудь MPI-процесс, ответственный за вычисление набора сеточных значений внутри области, отмеченной зеленым цветом (рис. 12.1) на следующем временном слое, он должен был обладать информацией о сеточных значений которые выделены синей рамкой (рис. 12.1) на текущем временном слое. По вот этой информации в двойном цикле происходит вычисление сеточных значений на следующем временном слое в узлах внутри зеленой рамки(рис. 12.1) .

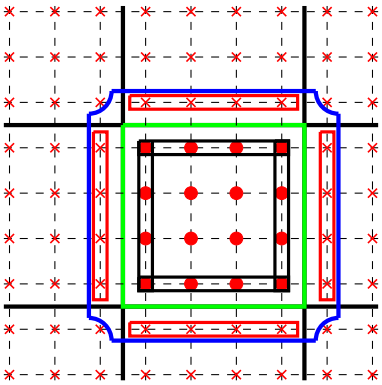


Рис. 12.1: Иллюстрация узлов сетки

Для того, чтобы продолжить вычисления на очередном временном слое MPI-процесс должен получить от своих соседей в двухмерной виртуальной топологии типа сетки информацию о сеточных значениях выделенных красной рамкой (рис. 12.1). Кроме того, чтобы остальные MPI-процессы могли продолжить работу он должен передать сеточные значения, узлы которых выделены черной рамкой (рис. 12.1). Эти действия по приему и передаче сообщений своим соседям были организованы таким образом

```
115     if my_col > 0 :
116         comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, 1, 1:], N_y_part, MPI.
    DOUBLE],
117                            dest=my_row*num_col + (my_col-1), sendtag=0,
118                            recvbuf=[u_part_aux[m+1, 0, 1:], N_y_part, MPI.DOUBLE
    ],
119                            source=my_row*num_col + (my_col-1), recvtag=MPI.
    ANY_TAG, status=None)
120
```

```
121    if my_col < num_col-1 :
122        comm_cart.Sendrecv(sendbuf=[u_part_aux[m+1, N_x_part_aux-2, 1:],
       N_y_part, MPI.DOUBLE],
123                           dest=my_row*num_col + (my_col+1), sendtag=0,
124                           recvbuf=[u_part_aux[m+1, N_x_part_aux-1, 1:], N_y_part
       , MPI.DOUBLE],
125                           source=my_row*num_col + (my_col+1), recvtag=MPI.
       ANY_TAG, status=None)
126
127    if my_row > 0 :
128        temp_array_send = u_part_aux[m+1, 1:N_x_part+1, 1].copy()
129        temp_array_recv = empty(N_x_part, dtype=float64)
130        comm_cart.Sendrecv(sendbuf=[temp_array_send, N_x_part, MPI.DOUBLE],
131                           dest=(my_row-1)*num_col + my_col, sendtag=0,
132                           recvbuf=[temp_array_recv, N_x_part, MPI.DOUBLE],
133                           source=(my_row-1)*num_col + my_col, recvtag=MPI.
       ANY_TAG, status=None)
134    u_part_aux[m+1, 1:N_x_part+1, 0] = temp_array_recv


135
136    if my_row > 0 :
137        temp_array_send = u_part_aux[m+1, 1:N_x_part+1, N_y_part_aux-2].copy
       ()
138        temp_array_recv = empty(N_x_part, dtype=float64)
139        comm_cart.Sendrecv(sendbuf=[temp_array_send, N_x_part, MPI.DOUBLE],
140                           dest=(my_row+1)*num_col + my_col, sendtag=0,
141                           recvbuf=[temp_array_recv, N_x_part, MPI.DOUBLE],
142                           source=(my_row+1)*num_col + my_col, recvtag=MPI.
       ANY_TAG, status=None)
143    u_part_aux[m+1, 1:N_x_part+1, N_y_part_aux-1] = temp_array_recv
```

Напомним, что в этом примере топология была виртуальной в виде декартовой сетки, в которой периодичность по каждому направлению была выключена, поэтому у некоторых процессов может не быть, например, левого соседа или нижнего. Поэтому здесь вышеописанные условия и стоят.

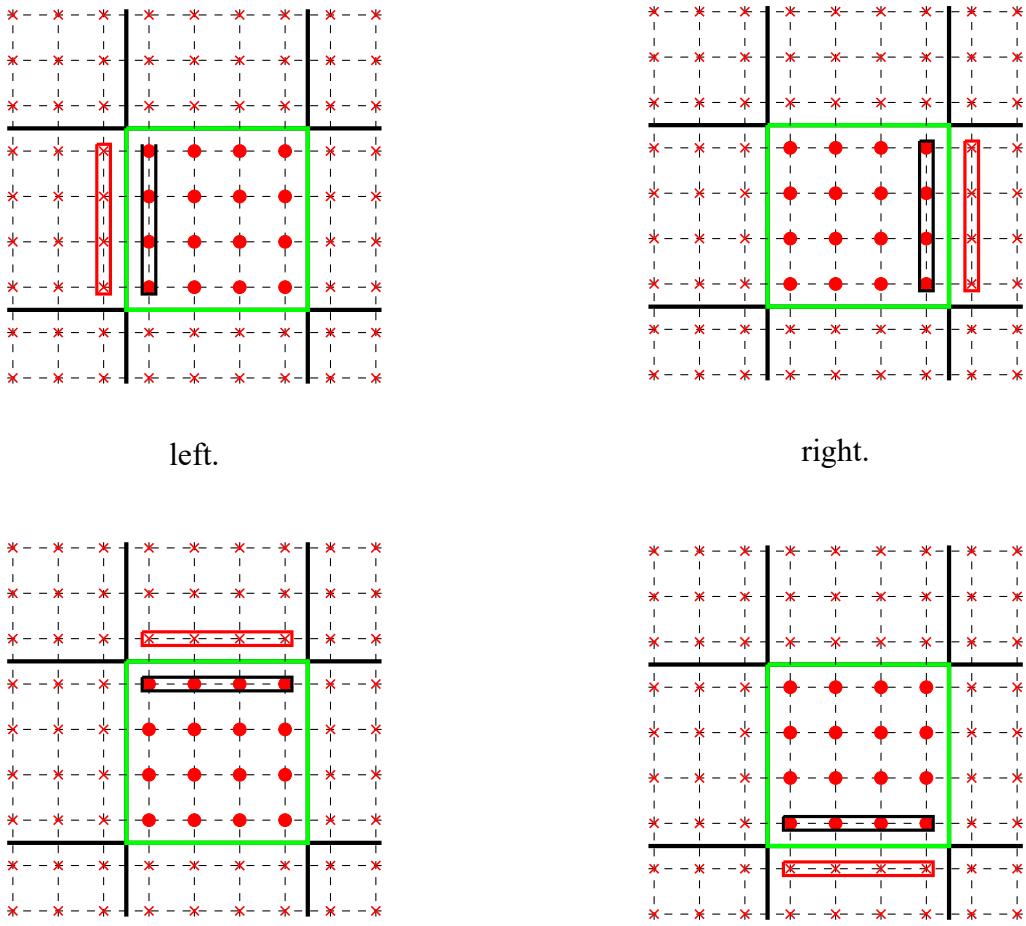
Здесь реализованны функции Sendrecv, которые высылают и получают сообщения. Все это происходит в большом цикле по числу временных слоев. Возникает мысль переписать Sendrecv вручную с помощью асинхронных операций, при этом инициализировать отложенные запросы на взаимодействия. Это возможно, поскольку на каждой итерации большого цикла эти действия выполняются по работе с массивами одинаковых размеров, сообщения посылаются всегда одному и тому же процессу.

Выделим место в памяти под следующие массивы.

```
    temp_array_N_y_left_send = empty(N_y_part, dtype=float64)
```

```
temp_array_N_y_left_recv = empty(N_y_part, dtype=float64)
temp_array_N_y_right_send = empty(N_y_part, dtype=float64)
temp_array_N_y_right_recv = empty(N_y_part, dtype=float64)
temp_array_N_x_top_send = empty(N_x_part, dtype=float64)
temp_array_N_x_top_recv = empty(N_x_part, dtype=float64)
temp_array_N_x_bottom_send = empty(N_x_part, dtype=float64)
temp_array_N_x_bottom_recv = empty(N_x_part, dtype=float64)
```

Например, temp_array_N_y_left_send будет посылаться левому соседу, аналогичный смысл имеют остальные массивы (рис. 12.2).



left. right.

top. bottom.

Рис. 12.2: Иллюстрация приема-передачи информации соседних MPI-процессов.

Так как произвольном случае сообщение будет 8, то и массив *requests* будет состоять из 8-ми элементов

```
requests = [MPI.Requests() for i in range(8)]
```

Далее инициализируем отложенный запрос на взаимодействия

```
if my_col > 0 :
    requests[0] = comm_cart.Send_init([temp_array_N_y_left_send, N_y_part,
   MPI.DOUBLE], dest=my_row*num_col + (my_col-1), tag=0)
    requests[1] = comm_cart.Recv_init([temp_array_N_y_left_send, N_y_part,
   MPI.DOUBLE], source=my_row*num_col + (my_col-1), tag=MPI.ANY_TAG)
```

requests[0] означает, что процессу, который в виртуальной топологии располагается слева от текущего, нужно будет послать число элементов N_y_part с типом данных MPI.DOUBLE, которые возьмутся из области памяти, ассоциированных с массивом temp_array_N_y_left_send. Аналогично requests[1] будет принимать сообщения. Поскольку инициализировать эти отложенные запросы на взаимодействия можно не для произвольного процесса из двухмерной виртуальной декартовой топологии, а только для тех, у кого есть сосед слева, то ставим условие, что координата соответствующего процесса (номер столбца) строго больше нуля. Аналогичным образом инициализируем отложенные запросы на взаимодействия с другими соседями

```
if my_col < num_col -1 :
    requests[2] = comm_cart.Send_init([temp_array_N_y_right_send, N_y_part
  , MPI.DOUBLE], dest=my_row*num_col + (my_col+1), tag=0)
    requests[3] = comm_cart.Recv_init([temp_array_N_y_right_send, N_y_part
  , MPI.DOUBLE], source=my_row*num_col + (my_col+1), tag=MPI.ANY_TAG)
if my_row > 0 :
    requests[4] = comm_cart.Send_init([temp_array_N_x_top_send, N_x_part,
  MPI.DOUBLE], dest=(my_row-1)*num_col + my_col, tag=0)
    requests[5] = comm_cart.Recv_init([temp_array_N_x_top_send, N_x_part,
  MPI.DOUBLE], source=(my_row-1)*num_col + my_col, tag=MPI.ANY_TAG)

if my_row > 0 :
    requests[6] = comm_cart.Send_init([temp_array_N_x_bottom_send,
  N_x_part, MPI.DOUBLE], dest=(my_row+1)*num_col + my_col, tag=0)
    requests[7] = comm_cart.Recv_init([temp_array_N_x_bottom_send,
  N_x_part, MPI.DOUBLE], source=(my_row+1)*num_col + my_col, tag=MPI.
  ANY_TAG)
```

После цикла, который является самой вычислительно емкой частью программы

```
104 for m in range(M) :
105
106     for i in range (1 N_x_part_aux - 1) :
107         for j in range (1 N_y_part_aux - 1) :
108             u_part_aux[m+1, i, j] = u_part_aux[m, i, j] + \
```

```
109                 tau*(eps*((u_part_aux[m, i+1, j] - 2*u_part_aux[m, i, j] +
    u_part_aux[m, i-1, j])/h_x**2 +
110                          (u_part_aux[m, i, j+1] - 2*u_part_aux[m, i, j] +
    u_part_aux[m, i, j-1])/h_y**2) +
111                     u_part_aux[m, i, j]*((u_part_aux[m, i+1, j] -
    u_part_aux[m, i-1, j])/(2*h_x) +
112                          (u_part_aux[m, i, j+1] - u_part_aux[m, i, j-1])
    /(2*h_y)) +
113                     u_part_aux[m, i, j**3])
114
```

необходимо обменяться сообщениями, для этого делаем

```
104 if my_col > 0 :
105     temp_array_N_y_left_send[:] = u_part_aux[m+1, 1, 1:N_y_part+1]
106     MPI.Prequest.Startall([requests[0], requests[1]])
107     MPI.Request.Waitall([requests[0], requests[1]],statuses=None)
108     u_part_aux[m+1, 0, 1:N_y_part+1] = temp_array_N_y_left_recv
109 if my_col < num_col -1 :
110     temp_array_N_y_right_send[:] = u_part_aux[m+1, N_x_part_aux-2, 1:
    N_y_part+1]
111     MPI.Prequest.Startall([requests[2], requests[3]])
112     MPI.Request.Waitall([requests[2], requests[3]],statuses=None)
113     u_part_aux[m+1, N_x_part_aux-1, 1:N_y_part+1] =
    temp_array_N_y_right_recv
114 if my_row > 0 :
115     temp_array_N_y_top_send[:] = u_part_aux[m+1, 1:N_x_part+1, 1]
116     MPI.Prequest.Startall([requests[4], requests[5]])
117     MPI.Request.Waitall([requests[5], requests[5]],statuses=None)
118     u_part_aux[m+1, 1:N_x_part+1, 0] = temp_array_N_y_top_recv

119 if my_row > 0 :
120     temp_array_N_y_bottom_send[:] = u_part_aux[m+1, 1:N_x_part+1,
    N_y_part_aux-2]
121     MPI.Prequest.Startall([requests[6], requests[7]])
122     MPI.Request.Waitall([requests[6], requests[7]],statuses=None)
123     u_part_aux[m+1, 1:N_x_part+1, N_y_part_aux-1] =
    temp_array_N_y_bottom_recv
124
```

К чему приведут эти изменения? Давайте посмотрим, как и на 10-ой лекции, на эффективность программы. Для тестирования использовался суперкомпьютер Ломоносов-2 суперкомпьютерного комплекса МГУ (НИВЦ МГУ имени М.В.Ломоносова)

Раздел test (max время: 15 минут; max узлов на задачу: 15)

Тип узлов :

CPU: 1x Intel Xeon E5-2697 v3 2.60GHz
CPU-ядер: 14
ОЗУ: 64GB (4.5 GB на ядро)

Тестовый набор параметров 1 :

$N_x = 200$, $N_y = 200$, $M = 4000$ (размер массива, содержащего сеточное решение : $N_x \times N_y \times M \times 8$байт $= 1.19$ GB)

Время работы последовательной версии программы на 1 ядре : 791.2 с.

График времени работы расчетной части с 10-ой лекции, пример 10.2, и новая реализация.

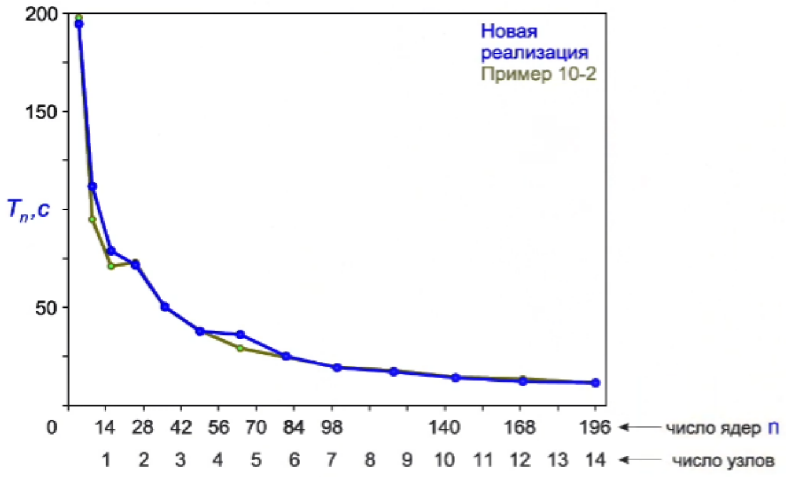


Рис. 12.3: График времени работы расчетной части

Здесь есть, конечно, какие-то отличия, но нас интересуют что происходит при большом числе ядер или узлов. По крайней мере зрительно кривые наложились друг на друга. Но если посмотреть на соответствующие числа более пристально, окажется что при 14 узлах время около 12-14 секунд и что время в новой реализации примерно на пол секунды меньше. Кажется, что при таком масштабе это совсем немного, но, как мы видели на предыдущем слайде, время работы на одном ядре примерно 800 секунд, поэтому если мы производим расчет на 200 ядрах, то само время счета естественно уменьшилась в 200 раз, то есть до 4 секунд. Общее время 12 секунд, значит 8 секунд приходится на взаимодействие между процессами. Получается, что время на прием-передачу сообщений сократилось на полсекунды от 8 секунд, то есть порядка 5-10%, не плохой результат.

Приведем также график ускорения

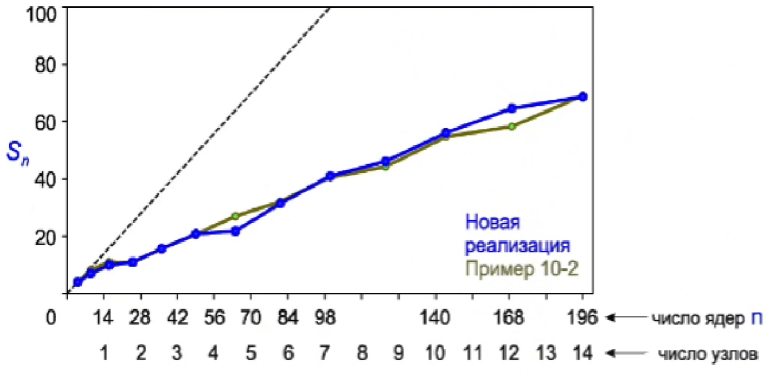


Рис. 12.4: График ускорения параллельных программ

Видим, что увеличение числа ядер приводит к повышению ускорения вычислений.

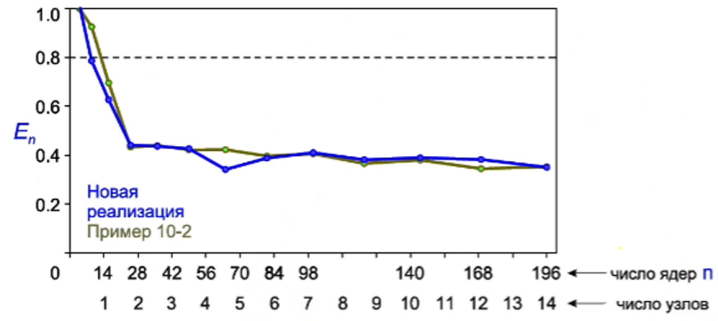


Рис. 12.5: График эффективности параллельных программ

Считается, что программа является эффективной, если ее эффективность на уровне от 0.8 и выше. У нас получилось примерно на уровне 0.4, но эта задача учебная, она изначально реализована не лучшим образом в плане вычислений.

Пример 12.2, модификация примера 6.2

Рассмотрим также пример 6.2 из 6-ой лекции, здесь отложенные запросы на взаимодействие тоже могут принести пользу. В этом примере был реализован метод сопряженных градиентов для решения переопределенных систем линейных алгебраических уравнений.

Предполагается, что была создана виртуальная двухмерная топология типа двухмерный тор, число строк и столбцов этой виртуальной топологии было зафиксировано, но по каждому из направлений была включена периодичность. Напомним, что происходит в этой программе. Определяли соседей этой виртуальной топологии справа, слева, сверху, снизу

```
16      def conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
17                                    N, comm_cart, comm_row, comm_col) :
18
19      neighbour_up, neighbour_down = comm_cart.Shift(direction=0, disp=1)
20      neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
21
22      ScalP_temp = empty(1, dtype=float64)
23
24      s = 1
25
26      p_part = zeros(N_part, dtype=float64)
27
28      while s<= N :
29
```

```
95
96      s = s + 1
97
98      return x_part
99
```

Далее в цикле у нас реализовывались следующие формулы метода сопряженных градиентов

$r^{(s)}$, $p^{(s)}$, $q^{(s)}$ — вспомогательные вектора,
$p^{(0)} = 0$,
$x^{(1)}$ — начальное приближение,

$$r^{(s)} = \begin{cases} A^T\left(A\,x^{(s)} - b\right), \ s = 1, \\ r^{(s-1)} - q^{(s-1)} \big/ \left(p^{(s-1)}, q^{(s-1)}\right), \end{cases}$$

$$p^{(s)} = p^{(s-1)} + r^{(s)} \big/ \left(r^{(s)}, r^{(s)}\right),$$
$$q^{(s)} = A^T\left(A\,p^{(s)}\right),$$
$$x^{(s+1)} = x^{(s)} - p^{(s)} \big/ \left(p^{(s)}, q^{(s)}\right),$$

$x^{(N)}$ — решение СЛАУ

Например, формулы для $r^{(s)}$ реализованы следующим образом

```
30  if s == 1 :
31      Ax_part_temp = dot(A_part, x_part)
```

```
32      Ax_part = Ax_part_temp.copy()
33      for n in range(num_col-1) :
34          comm_cart.Sendrecv_replace([Ax_part_temp, M_part, MPI.DOUBLE],
35                                     dest=neighbour_right, sendtag=0,
36                                     source=neighbour_left, recvtag=MPI.
    ANY_TAG,
37                                     status=None)
38          Ax_part = Ax_part + Ax_part_temp
39      b_part = Ax_part - b_part
40      r_part_temp = dot(A_part.T, b_part)
41      r_part = r_part_temp.copy()
42      for m in range(num_row-1) :
43          comm_cart.Sendrecv_replace([r_part_temp, N_part, MPI.DOUBLE],
44                                     dest=neighbour_down, sendtag=0,
45                                     source=neighbour_up, recvtag=MPI.
    ANY_TAG,
46                                     status=None)
47      r_part = r_part + r_part_temp
48  else :
49      ScalP_temp[0] = dot(p_part, q_part)
50      ScalP = ScalP_temp.copy()
51      for n in range(num_col-1) :
52          comm_cart.Sendrecv_replace([ScalP_temp, 1, MPI.DOUBLE],
53                                     dest=neighbour_right, sendtag=0,
54                                     source=neighbour_left, recvtag=MPI.
    ANY_TAG,
55                                     status=None)
56          ScalP = ScalP + ScalP_temp
57      r_part = r_part - q_part/ScalP
58
```

Команды Sendrecv_replace можем реализовать с помощью асинхронных операций, используя отложенные запросы на взаимодействия, так как они многократно вызываются в цикле. Напомним, что наиболее вычислительно емкие операции здесь это произведение матрицы на вектор с помощью встроенной функции dot из пакета numpy и скалярное произведение векторов.

Вычисление скалярного произведения:

$$p^{(s)} = p^{(s-1)} + r^{(s)} / \left(r^{(s)}, r^{(s)}\right).$$

```
59      ScalP_temp[0] = dot(r_part, r_part)
60      ScalP = ScalP_temp.copy()
61      for n in range (num_col-1) :
62          comm_cart.Sendrecv_replace([ScalP_temp, 1, MPI.DOUBLE],
63                                     dest=neighbour_right, sendtag=0,
```

```
64                                      source=neighbour_left, recvtag=MPI.
    ANY_TAG,
65                                      status=None)
66
67          ScalP = ScalP + ScalP_temp
68      p_part = p_part + r_part/ScalP
```

Операция

$$q^{(s)} = A^T\left(A\,p^{(s)}\right)$$

```
69      Ap_part_temp = dot(A_part, p_part)
70      Ap_part = Ap_part_temp.copy()
71      for n in range(num_col-1) :
72          comm_cart.Sendrecv_replace([Ap_part_temp, M_part, MPI.DOUBLE],
73                                      dest=neighbour_right, sendtag=0,
74                                      source=neighbour_left, recvtag=MPI.
    ANY_TAG,
75                                      status=None)
76          Ap_part = Ap_part + Ap_part_temp
77      q_part_temp = dot(A_part.T, Ap_part)
78      q_part = q_part_temp.copy()
79      for n in range(num_col-1) :
80          comm_cart.Sendrecv_replace([Ap_part_temp, N_part, MPI.DOUBLE],
81                                      dest=neighbour_right, sendtag=0,
82                                      source=neighbour_left, recvtag=MPI.
    ANY_TAG,
83                                      status=None)
84          q_part = q_part + q_part_temp
```

Операция

$$x^{(s+1)} = x^{(s)} - p^{(s)}/\left(p^{(s)}, q^{(s)}\right)$$

```
69      Ap_part_temp = dot(p_part, q_part)
70      ScalP = ScalP_temp.copy()
71      for n in range(num_col-1) :
72          comm_cart.Sendrecv_replace([Ap_part_temp, 1, MPI.DOUBLE],
73                                      dest=neighbour_right, sendtag=0,
74                                      source=neighbour_left, recvtag=MPI.
    ANY_TAG,
75                                      status=None)
76      ScalP = ScalP + ScalP_temp
77      x_part = x_part - p_part/ScalP
```

Опять же команды Sendrecv_replace реализуем, используя отложенные запросы на взаимодействие. Итак, выделим место в памяти под вспомогательные массивы, как обычно

```
def conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
                              N, comm_cart, comm_row, comm_col) :

    neighbour_up, neighbour_down = comm_cart.Shift(direction=0, disp=1)
    neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
    Ax_part_temp = empty(M_part, dtype=float64)
    Ax_part_temp_recv = empty(M_part, dtype=float64)
    r_part_temp = empty(N_part, dtype=float64)
    r_part_temp_recv = empty(N_part, dtype=float64)
    ScalP_part_temp = empty(1, dtype=float64)
    ScalP_part_temp_recv = empty(1, dtype=float64)

    requests = [MPI.Requests() for i in range(6)]

    requests[0] = comm_cart.Send_init([Ax_part_temp, M_part, MPI.DOUBLE],
  dest=neighbour_right, tag=0)

    requests[1] = comm_cart.Recv_init([Ax_part_temp_recv, M_part, MPI.
  DOUBLE], source=neighbour_left, tag=MPI.ANY_TAG)

    requests[2] = comm_cart.Send_init([r_part_temp, N_part, MPI.DOUBLE],
  dest=neighbour_down, tag=0)

    requests[3] = comm_cart.Recv_init([r_part_temp_recv, N_part, MPI.
  DOUBLE], source=neighbour_up, tag=MPI.ANY_TAG)

    requests[4] = comm_cart.Send_init([ScalP_part_temp, 1, MPI.DOUBLE],
  dest=neighbour_right, tag=0)

    requests[5] = comm_cart.Recv_init([ScalP_part_temp_recv, 1, MPI.DOUBLE
  ], source=neighbour_left, tag=MPI.ANY_TAG)

    s = 1

    p_part = zeros(N_part, dtype=float64)

    while s<= N
```

Отлично, мы условно вынесли за цикл эти операции, теперь посмотрим, что делается внутри цикла while.

```
if s == 1 :
    Ax_part_temp = dot(A_part, x_part)
    Ax_part = Ax_part_temp.copy()
    for n in range(num_col-1) :
        MPI.Prequest.Startall([requests[0], requests[1]])
```

```
            MPI.Request.Waitall([requests[0], requests[1]],statuses=None)
            Ax_part_temp[:] = Ax_part_temp.recv
            Ax_part = Ax_part + Ax_part_temp
        b_part = Ax_part - b_part
        r_part_temp = dot(A_part.T, b_part)
        r_part = r_part_temp.copy()
        for m in range(num_row-1) :
            MPI.Prequest.Startall([requests[2], requests[3]])
            MPI.Request.Waitall([requests[2], requests[3]],statuses=None)
            r_part_temp[:] = r_part_temp.recv
            r_part = r_part + r_part_temp
    else :
        ScalP_temp[0] = dot(p_part, q_part)
        ScalP = ScalP_temp.copy()
        for n in range(num_col-1) :
            MPI.Prequest.Startall([requests[4], requests[5]])
            MPI.Request.Waitall([requests[4], requests[5]],statuses=None)
            ScalP_temp[0] = ScalP_temp_recv
            ScalP = ScalP + ScalP_temp
        r_part = r_part - q_part/ScalP
```

Изменения в кусочках программы, для нахождения $p^{(s)}$, $q^{(s)}$ и $x^{(s+1)}$ абсолютно аналогичны, на них останавливаться не будем. Хорошим тоном считается, что после того, как отложенные запросы на взаимодействия больше запускать не нужно, все request очистить

```
    for i in range(len(requests)) :
        MPI.Request.Free(requests[i])

    return x_part
```

В принципе, это все изменения. Так же как и в примере 12.1, эта программа работает чуть быстрее, чем та реализация, приведенная в примере 6.2. Здесь есть некоторые интересные нюансы, связанные с эффективностью. Для тестирования использовался суперкомпьютер Ломоносов-2 суперкомпьютерного комплекса МГУ (НИВЦ МГУ имени М.В.Ломоносова)

Раздел test (max время: 15 минут; max узлов на задачу: 15)

Тип узлов :

CPU: 1x Intel Xeon E5-2697 v3 2.60GHz
CPU-ядер: 14
ОЗУ: 64GB (4.5 GB на ядро)

Тестовый набор параметров 1 :

$N = 70000$, $M = 90000$ (размер массива, содержащего сеточное решение : $N \times M \times$ 8байт $= 46.9$ GB)

Время работы последовательной версии программы на 1 узле : 734 с.

Подчеркиваем, что в отличии от примера 12.1, где время работы последовательной версии программы было на 1 ядре, в этом примере рассматривается именно на 1 узел. Это связано с тем, что в предыдущем примере все вычисления делались на python, и если в его настройках при запуске программы не отключено требование запуска только на одном ядре, то при выполнении программы, когда запускается функция dot из пакета numpy, она определяет, что запущена на многоядерном процессоре и автоматически включается многопоточная реализация, которая задействует все ядра. Мы уже обсуждали на 4-ой лекции, что для того, чтобы получить время, эквивалентное времени работы на 1 ядре, нужно итоговое время умножить на 14, хотя это и достаточно грубая оценка.

Проведем вычисления таким образом, чтоб один MPI-процесс запускается на одном ядре и посмотрим на график зависимости времени работы программы от числа ядер.

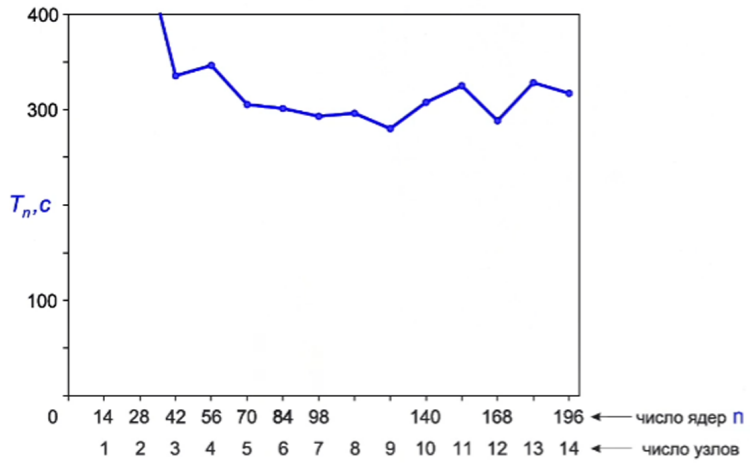


Рис. 12.6: График зависимости времени работы программы от числа ядер

Видим что-то чрезвычайно странное, время работы округлим до 800 секунд, получается какое бы число ядер мы не используем, время счета примерно все время одинаково, порядка 300-350 секунд, ускорение всего лишь примерно в два раза. Таким образом, график ускорения выглядит вот так

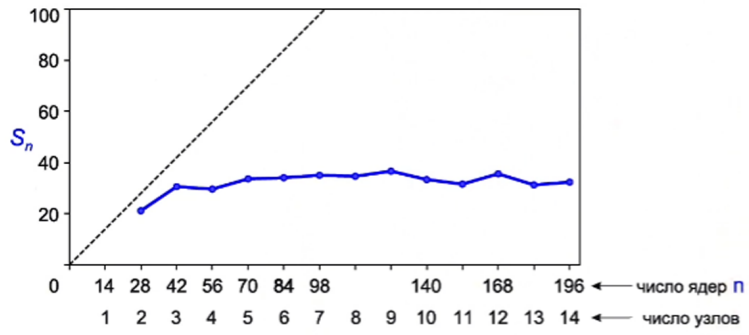


Рис. 12.7: График ускорения

А эффективность только падает

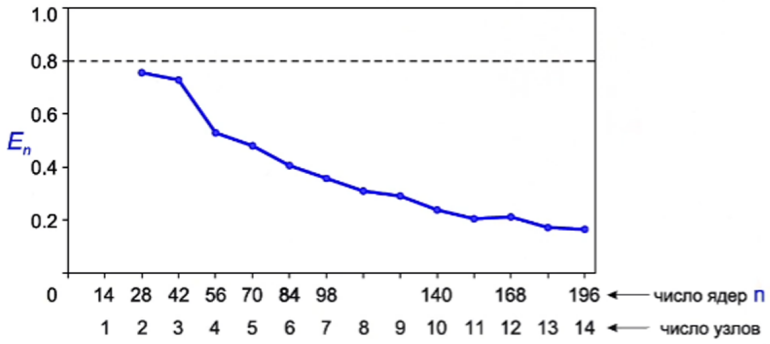


Рис. 12.8: График эффективности

Очень странный результат, в чем причина? Есть некоторые особенности запуска на реальных многопроцессорных системах, еще раз подчеркнем, что если не указать в настройках python, что один MPI-процесс запускается только на одном ядре, то есть фактически не отключили многопоточность функции dot, то происходит следующее; каждый MPI-процесс привязывается к одному ядру, на этом ядре выполняется соответствующий python-код, когда запускается функцию dot, она видит, что запущена на многоядерном процессоре и включает многопоточную реализацию. Так делают все MPI-процессы, запущеные на этом узле, возникает много конфликтов, за счет чего время работы фактически не ускоряется. Можно сделать следующие изменения. Мы можем не просто порождать какое-то число MPI-процессов? а можем указать, что один MPI-процесс привязывается и запускается не на одном ядре, а именно на одном узле. Это приводит к тому, что один MPI-процесс запускается на одном узле, python вроде бы работает последовательно, но все вычисления (99,(9)%) проводятся с помощью функции dot. Вызывается функция dot, python

видит, что запущена на многоядерной системе, запускается многопоточная реализация, которая использует все эти ядра. В результате, если мы сделаем так, у нас получается вот такой график

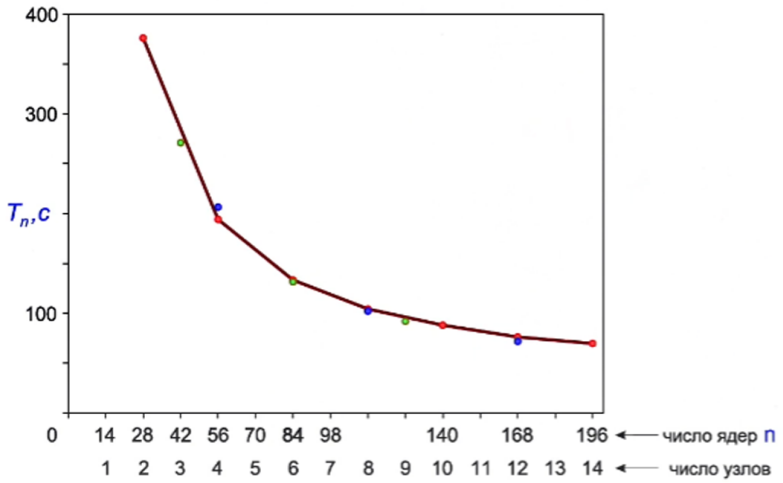


Рис. 12.9: График зависимости времени работы программы от числа ядер

Точками отмечены результаты при использовании различных топологий: зеленые точки - число строк 4, синие - число строк 3, красные - число строк 2. И даже вне зависимости от определения оптимальной топологии получается достаточно хороший результат, время работы весьма быстро уменьшается с увеличением числа узлов. График ускорения

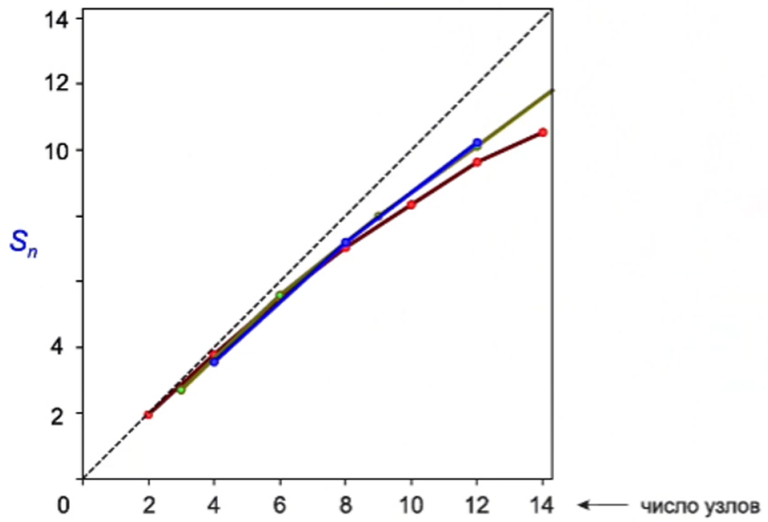


Рис. 12.10: График ускорения

Видим, что во всех случаях, что ускорение близко к линейному, при этом наилучшей является топология, при которой число строк равно 4, в этом случае, число строк и столбцов примерно совпадают, на 4-ой лекции обсуждалось, что это наиболее оптимальный вариант. График эффективности

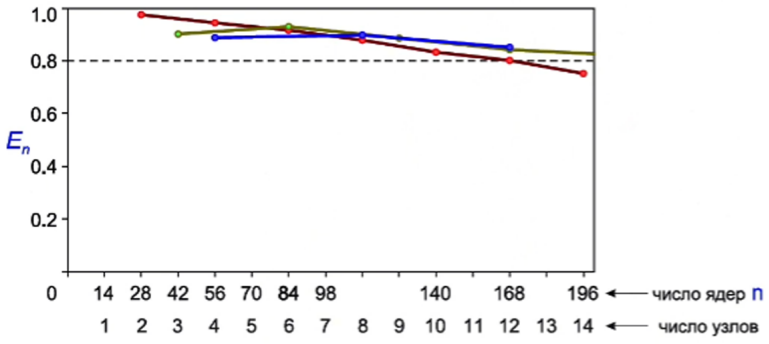


Рис. 12.11: График эффективности

Здесь эффективность лежит как раз в пределах от 0.8 до 1 (кроме красной линии, на 14 узлах получается 2 строки, 7 столбцов — не самая удачная топология), то есть можно заключить, что программная реализация действительно эффективна.

Лекция 13. Технологии гибридного параллельного программирования

На этой лекции будут обсуждены гибридные технологии параллельного программирования. На данный момент при написании всех программ мы абстрагировались от того, какими техническими особенностями обладают многопроцессорные системы, которые мы использовали для вычислений.

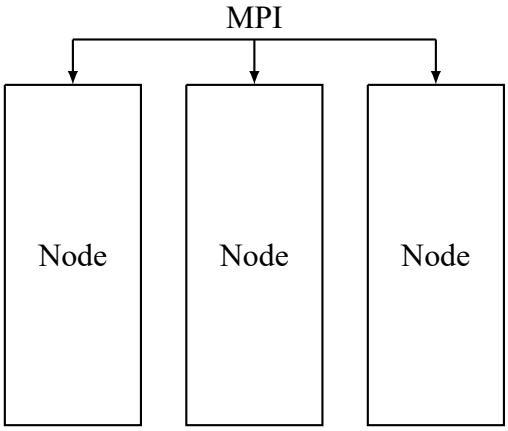


Рис. 13.1

Предполагалось только, что каждый вычислительный процесс запускается на своем вычислительном узле, и эти вычислительные узлы или процессы взаимодействуют друг с другом посредством передачи сообщений с помощью технологии MPI (Рис. 13.1). Мы предполагали, что по-крайней мере каждый вычислительный узел содержит центральный процессор и оперативную память. Таким образом, у нас есть система с распределенной памятью (Рис. 13.2).

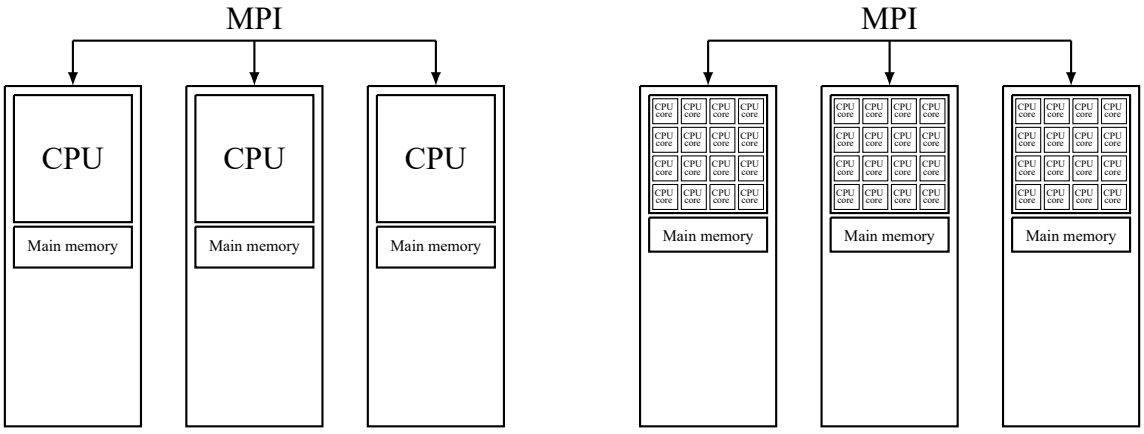


Рис. 13.2

В реальной жизни все гораздо сложнее, в нынешние времена каждый центральный

процессор это скорее всего многоядерная система, которая состоит из относительно большого числа вычислительных ядер (Рис. 13.2). Таким образом, каждый вычислительный узел мы можем локально рассматривать как вычислительную систему с общей памятью.

Более того, каждый вычислительный узел может содержать еще и графическую видеокарту, которая состоит из достаточно большого числа графических процессоров и содержит общую видеопамять. Таким образом, локально видеокарту мы можем воспринимать тоже как многопроцессорную систему с общей памятью (Рис. 13.3).

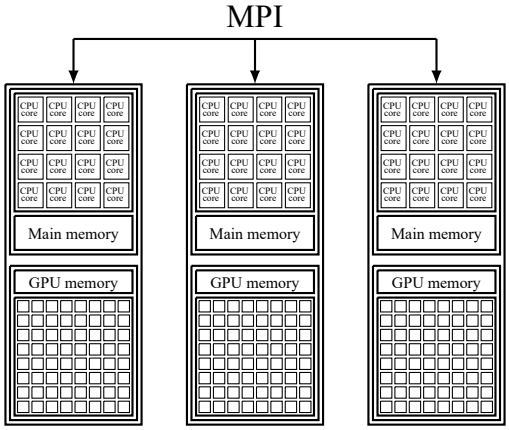


Рис. 13.3: Иллюстрация узлов сетки

То есть каждый узел содержит многоядерный процессор с общей оперативный памятью и видеокарту, которая тоже является многопроцессорной системой с общей памятью. Примером такой системы является суперкомпьютер «Ломоносов-2».

Рис. 13.4: Суперкомпьютер «Ломоносов-2» (2020),
НИВЦ МГУ имени М.В. Ломоносова

Здесь мы видим одну из многих стоек суперкомпьютера «Ломоносов-2», которая содержит большое число плат, которые представляют вышеописанные системы.

Конфигурация суперкомпьютера «Ломоносов-2» суперкомпьютерного комплекса МГУ (НИВЦ МГУ имени М.В. Ломоносова)

Основной раздел compute: 1504 узла типа K40

Узел K40:

CPU: 1x Intel Xeon E5-2697 v3 2.60GHz
CPU-ядер: 14
ОЗУ: 64GB (4.5 GB на ядро)

GPU: 1x Tesla K40s Объем памяти на GPU: 11.56 GB

Возникает вопрос, как все это эффективно использовать? Для этого, кроме технологии MPI, на помощь приходят следующие технологии

- MPI ≡ Message Passing Interface — технология параллельного программирования для систем с распределенной памятью.
- OpenMP ≡ Open Multi-Processing — технология параллельного программирования для систем с общей памятью, которая используется для написания параллельных программ на многоядерных процессорах.
- CUDA ≡ Compute Unified Device Architecture — технология параллельного программирования с использованием видеокарт Nvidia (системы с общей памятью). Эта технология позволяет писать достаточно эффективные программные реализации параллельных алгоритмов, которые при вычислениях задействуют графические процессоры.

Если вам доступен один вычислительный узел, который содержит многоядерный процессор и видеокарту, то можно ограничиться либо технологии CUDA, либо технологией OpenMP, задействуя для вычислений все ядра многоядерного процессора. Если таких узлов много, то необходимо организовать взаимодействие между этими узлами, это можно сделать с помощью технологии MPI.

Когда мы тестировали некоторые программы, которые реализовывали в этом курсе, мы на соответствующих графиках отмечали зависимость, например, времени работы многопроцессорной программы от количества ядер, либо от количество узлов, то есть в

принципе можно использовать технологию MPI в случае наличия многоядерного процессора, в этом случае каждый MPI-процесс обычно привязывается к своему вычислительному ядру центрального процессора.

Минус такого подхода в том, что MPI — это технология программирования для систем с распределенной памятью, поэтому если вы задействуете технологию MPI для написания параллельной версии какого-нибудь алгоритма и используйте для этого свой персональный компьютер, на котором один центральный процесс, но многоядерный, вы хоть и будете получать ускорение, но по сравнению с технологией OpenMP технология MPI будет в каком-то смысле тяжеловесной. Будут организовываться отдельные MPI-процессы, каждому из них будет привязываться свое ядро и выделяться какая-то часть оперативной памяти. Потом будет системно организовываться взаимодействие между этими процессами, которые на самом деле выполняются локально на каком-то многопроцессорном устройстве с общей памятью, то есть накладных расходов будет больше по сравнению с технологией OpenMP, которая порождает много потоков, каждый из которых привязывается к своему вычислительному ядру, но потоки между собой фактически не взаимодействует, потому что они берут данные из общей памяти.

На этой лекции обзорно обсудим, как в рамках тех знаний, что мы изучили на этом курсе, используя особенности python и некоторые его пакеты, как можно достаточно просто использовать гибридные технологии параллельного программирования, как в программе учесть то, что у нас на каждом вычислительном узле есть многоядерный процессор и есть видеокарта.

Преимущество и особенности функции dot

Обсуждение начнем с простого примера

```
from numpy import empty

f1 = open('in.dat', 'r')
N = int(f1.readline())
M = int(f1.readline())
f1.close()

A = empty((M,N)); x = empty(N); b = empty(M);

f2 = open('AData.dat'. 'r')
for j in range(M) :
    for i in range(N) :
        A[j,i] = float(f2.readline())
f2.close()
```

```
f3 = open('xData.dat'. 'r')
for i in range(N) :
    x[i] = float(f3.readline())
f3.close()

for j in range(M) :
    b[j] = 0
    for i in range(N) :
        b[j] = b[j] + A[j,i]*x[i]

print(b)
```

С нее мы начинали все обсуждение еще на первой лекции. Эта программа перемножает матрицу A на вектор x. В учебных целях на первой лекции мы из файла «in.dat» считали число столбцов матрицы N, число строк M, выделили под матрицу A, вектор x и вектор b места в памяти.

Мы обсуждали, что такая программная реализация неоптимальная, потому что python делает все вычисления достаточно медленно, и так как мы используем пакет numpy, то грех не заменить двойной цикл на более простую в плане программной реализации функцию dot. Преимущество этой функции в том, что большинство функций из пакета numpy реализованы на C и Fortran, поэтому даже их однопроцессорные реализации будут работать гораздо быстрее, чем те, которые мы напишем на python.

У этой функции есть еще одно значительное преимущество. Программа на Python всегда выполняется только на одном ядре, многопоточность на нем в принципе не реализуема, но если какая-либо функция написана по C или Fortran, то в соответствующем коде может уже быть реализована многопоточность. Когда вызывается соответствующая расчетная часть, система видит, что функция запущена на многоядерном процессоре и запускает многопоточную реализацию перемножения матрицы на вектор, написанную с помощью технологии OpenMP.

Пример функции, использующей технологию OpenMP

Рассмотрим параллельный алгоритм решения СЛАУ $Ax = b$. Такую задачу мы разбирали много раз

- Пример 3-1 (Лекция 3). Распараллелены абсолютно все вычисления (в рамках коммуникатора comm = MPI.COMM_WORLD)

- Пример 3-2 (Лекция 3). Распараллелена только часть вычислений с целью уменьшения накладных расходов, связанных с приемом-передачей информации между вычислительными узлами (в рамках коммуникатора comm = MPI.COMM_WORLD)
- Пример 6-2 (Лекция 6). Распараллелены абсолютно все вычисления(в рамках коммуникатора с декартовой топологией comm_cart типа двумерного тора)
- Пример 12-2 (Лекция 12). Усовершенствован Пример 6-2 с помощью использования асинхронных операций.

На примере 3-2 покажем, как встроить технологии OpenMP. Главной функцией являлся метод сопряженных градиентов, который как раз и осуществлял поиск вектора x.

```
9  def conjugate_gradient_method(A_part, b_part, x, N, comm) :
10
11     p = empty(N, dtype=float64)
12     r = empty(N, dtype=float64)
13     q = empty(N, dtype=float64)
14
15     s = 1
16
17     p[:] = zeros(N, dtype=float64)
18
19     while s <= N :
20
21     if s == 1 :
22         r_temp = dot(A_part.T, dot(A_part, x) - b_part)
23         comm.Allreduce([r_temp, N, MPI.DOUBLE],
24            [r, N, MPI.DOUBLE], op=MPI.SUM)
25     else :
26         r = r - q / dot(p, q)
27
28     p = p + r / dot(r, r)
29     q_temp = dot(A_part.T, dot(A_part, p))
30     comm.Allreduce([q_temp, N, MPI.DOUBLE], [q, N, MPI.DOUBLE], op=MPI.SUM
   )
31     x = x - p / dot(p, q)
32     s = s + 1
33
34 return x
```

На вход подается блок матрицы A, часть вектора b соответствующих размеров, и вектор x длины N, который из себя представляет начальное приближение для итерационного метода. По итогу работы этой программы возвращается вектор x, который является решением системы $Ax = b$.

Самая вычислительно емкая часть это перемножение матрицы и транспонированной матрицы на вектор. Что нужно сделать, чтобы применить технологию OpenMP? Каждый MPI-процесс должен запускаться не на отдельном ядре, а на отдельном узле. Когда программа будет работать, MPI-процесс изначально будет запускаться на каком-то только одном ядре этого узла, но как только будет осуществляться вызов функций dot, эти функции будут видеть, что они запущены на многоядерных процессорах и будет запускаться их параллельная реализация, написанная с помощью технологии OpenMP. То есть переписывать программу не нужно.

Для тестирования использовался суперкомпьютер Ломоносов-2 суперкомпьютерного комплекса МГУ (НИВЦ МГУ имени М.В.Ломоносова)

Раздел test (max время: 0.25 часов; max узлов на задачу: 15) Раздел compute (max время: 48 часов; max узлов на задачу: 50)

Тип узлов :

CPU: 1x Intel Xeon E5-2697 v3 2.60GHz
CPU-ядер: 14
ОЗУ: 64GB (4.5 GB на ядро)
GPU: 1x Tesla K40s
Объем памяти на GPU: 11.56 GB

Тестовый набор параметров :

$N = 70000$, $M = 90000$, $s_{\text{lim}} = 400$
(размер массива, содержащего сеточное решение : $N \times M \times 8$байт $= 46.9$ GB)

Время работы последовательной версии программы на 1 узле : 734 с.

На 4-ой лекции, чтобы продемонстрировать, что программа работает, мы подбирали специфические параметры N и M — число строк было в тысячи раз больше числа неизвестных. Это было связано с тем, что каждый MPI-процесс запускался на отдельном вычислительном ядре. На практике чаще всего мы решаем переопределенные СЛАУ, у которых число строк не сильно больше числа неизвестных.

Для тестирования таких сложных приложений использовался суперкомпьютер «ломоносов-2» со следующими параметрами запуска

```
module load slurm
module load anaconda3/3.7.0 mpich/3.4.2-gcc
source activate mypyenv
```

Запуск задачи с помощью скрипт файла:

```
#!/bin/bash - -login

#SBATCH - -partition=test
#SBATCH - -time=0-00:15:00
#SBATCH - -nodes=15
#SBATCH - -ntasks-per-node=14


srun - -mpi=pmi2 python ./test.py
```

При запуске этого скрипта выделялось 15 узлов $\times$ 14 ядер $=$ 210 ядер, порождалось 210 MPI-процессов, каждый из которых запускался на соответствующем вычислительном ядре. Если хотим запускать 1 MPI-процесс только на одном узле, записываем ntasks-per-node=1. Теперь запрашивается 15 узлов, но на каждом из них будет запускаться только 1 MPI-процесс.

Посмотрим на графики, запустив пример 3.2 и 3.1

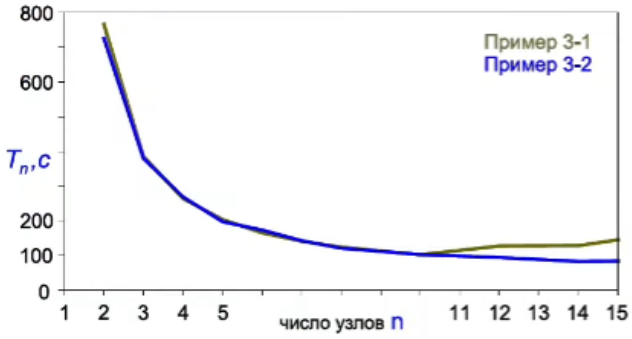


Рис. 13.5: График зависимости времени работы программы от числа ядер

Несмотря на то, что часть операций в примере 3.2 была не распараллелена, этот пример является более эффективным. На основе этого графика можно построить график ускорения

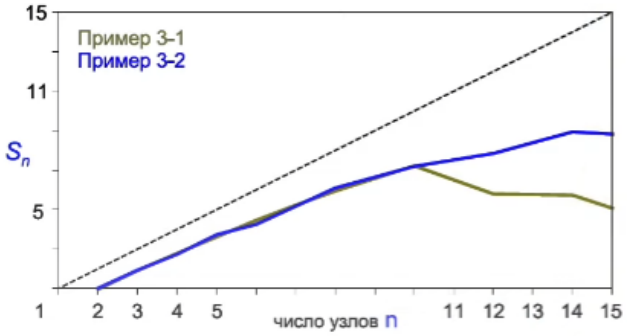


Рис. 13.6: График ускорения

Пунктиром отображен идеальный случай линейного ускорения. Примерно до 10-ти узлов, ускорение почти параллельно линейному случаю, оно не совпадает по той причине, что был выделен отдельно один управляющий процесс. После 10-ти узлов для примера 3.2 ускорение продолжает расти до 14-ти узлов, а для примера 3.1 начинает падать. Далее можно построить график эффективности

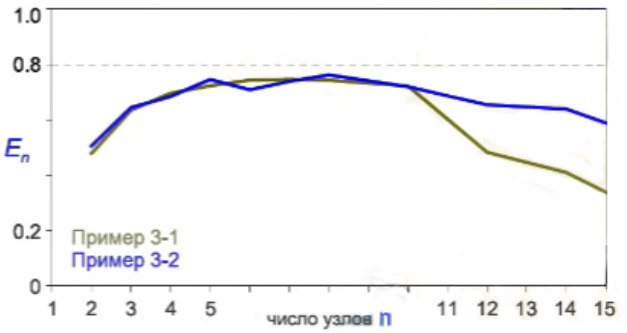


Рис. 13.7: График эффективности

Чем ближе график к единице, тем программная реализация эффективней. По этому графику можно судить о масштабируемости, то есть о способности программы сохранять эффективность с увеличением числа вычислительных узлов. Здесь видим, что для примера 3.2 эффективность сохраняется в рамках такого количества узлов.

Перестроим эти графики, увеличив число узлов до 64, а время работы уменьшив до 300. Для примеров 3.1 и 3.2 не имеет смысла использовать больше узлов, так как видно, что уже при прежнем масштабе их эффективность падает. Добавим на эти графики пример 12.2, который используют виртуальные топологии, которые попытались отобразить на реальную физическую топологию суперкомпьютера «ломоносов-2», то есть пример

уже заведомо более эффективный и запускался вплоть до 64-х узлов. Так же на графиках отобразим ту же реализация программы из примера 12.2, но с неудачным выбором топологии, при которой соседние узлы в реальном физической топологии расположены достаточно хаотично, поэтому время на коммуникации становится больше.

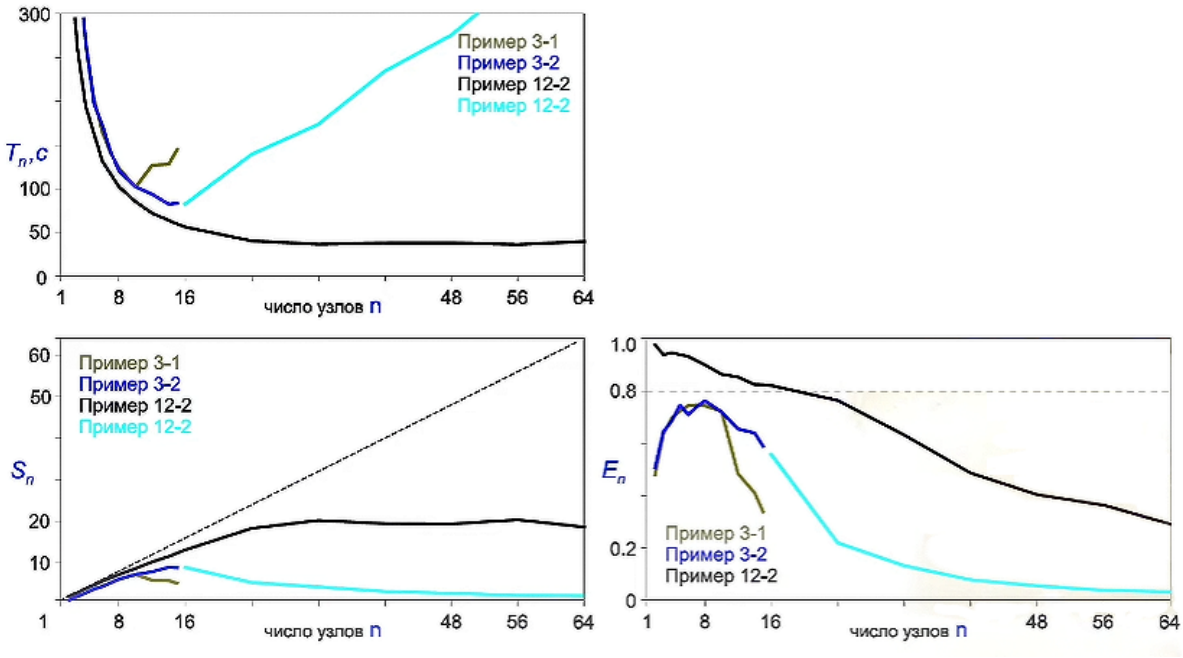


Рис. 13.8

Если привязывать один MPI-процесс к одному ядру, то графики будут гораздо хуже, если привязывать к одному вычислительному узлу — лучше, потому что в нашей программной реализации, когда MPI-процесс доходит до функций dot, то автоматически запускается их многопоточная версия, написанная на C, либо на Fortran.

Пример функции, использующей технологию CUDA

Снова вернемся к примеру перемножения матрицы на вектор с использованием функции dot из пакета numpy. Как с помощью технологии CUDA это реализовать? Есть несколько популярных пакетов, например, «numba» и «cupy». Пакет numba позволяет писать программы на низком уровне, то есть в таком же стиле, как мы писали программы под MPI.

В рамках нашего курса рассмотрение будет обзорным, поэтому будут использоваться пакетные решения. Подгрузим пакет cupy целиком

```
from numpy import empty
imort cupy as cp
```

Далее необходимо ввести терминологию, используемую при написании програм на видеокартах. Хост — центральный процессор и его оперативная память. Девайс — видеокарта, то есть графические процессоры и соответствующая видеопамять. После считывания данных

```

f1 = open('in.dat', 'r')
N = int(f1.readline())
M = int(f1.readline())
f1.close()

A = empty((M,N)); x = empty(N); b = empty(M);

f2 = open('AData.dat'. 'r')
for j in range(M) :
for i in range(N) :
A[j,i] = float(f2.readline())
f2.close()

f3 = open('xData.dat'. 'r')
for i in range(N) :
x[i] = float(f3.readline())
f3.close()
```

Они хранятся в общей оперативной памяти. Для вычислений на видеокарте необходимо перенести данные из общей оперативной памяти в видеопамять. Для этого в рамках пакета cupy есть простые функции

```

A_d = cp.asarray(A)
x_d = cp.asarray(x)
```

Она фактически по системной шине передает матрицу A и вектор x из оперативной памяти нашей системы в видеопамять, в результате создаются объекты, которые расположены на видеокарте, поэтому здесь принято ставить постфиксы «d» от слова device.

Теперь осталось вызвать функцию dot, но не из пакета numpy, а из пакета cupy, для этого нужно написать cp.dot:

```

b_d = cp.dot(A_d, x_d)
b = cp.asnumpy(b_d)

print(b)
```

Внутри эта функция как раз использует технологии CUDA. Результирующий вектор хранится на видеопамяти, чтобы его вывести, нужно сначала перенести его в оперативную память с помощью команды «cp.asnumpy». Перенести вектор b на оперативную память можно и следующим образом

```
24
25 b = cp.dot(A_d, x_d),get()
26
27
28 print(b)
```

Теперь видоизменим, используя технологии CUDA, пример 3.2, здесь только метод сопряженных градиентов, но полная версия программы выложена на портале teach-in в разделе «Материалы к лекциям».

```
9  def conjugate_gradient_method(A_part, b_part, x, N, comm) :
10     import cupy as cp
11
12     A_part_d = cp.asarray(A_part)
13
14     p = empty(N, dtype=float64)
15     r = empty(N, dtype=float64)
16     q = empty(N, dtype=float64)
17
18     s = 1
19     p[:] = zeros(N, dtype=float64)
20
21     while s <= N :
22
23     if s == 1 :
24         x_d = cp.asarray(x); b_part_d = cp.asarray(b_part)
25         r_temp = cp.dot(A_part_d.T, cp.dot(A_part_d, x_d) - b_part_d).get
   ()
26         comm.Allreduce([r_temp, N, MPI.DOUBLE], [r, N, MPI.DOUBLE], op=MPI
   .SUM)
27     else :
28         r = r - q / dot(p, q)
29
30     p = p + r / dot(r, r)
31     p_d = cp.asarray(p)
32     q_temp = cp.dot(A_part_d.T, cp.dot(A_part_d, p_d)).get()
33     comm.Allreduce([q_temp, N, MPI.DOUBLE], [q, N, MPI.DOUBLE], op=MPI.SUM
   )
34     x = x - p / dot(p, q)
35     s = s + 1
36
37 return x
```

Обратим внимание на некоторые особенности. Во-первых, хоть мы и считаем r_temp в 25-ой строке с помощью CUDA, нам необходимо эти временные векторы сложить, чтобы результат оказался на каждом из MPI-процессов, поэтому его нужно перенести в оперативную память хоста. Это является узким местом технологии CUDA, по возможности нужно избегать приема-передачи данных с хоста на девайс и наоборот. Однако здесь это необходимо. Во-вторых, скалярное произведение считается достаточно легко, поэтому на центральном процессоре это может занять меньше времени, чем посылка сообщений на видеокарту. Кроме того, так как при их счете используется функция dot из пакета numpy, то здесь также участвует технология OpenMP.

В заключение рассмотрим пример 12.2, он более громоздок, но и более эффективен. Так же встроим технологию CUDA

```
def conjugate_gradient_method(A_part, b_part, x_part, N_part, M_part,
                              N, comm_cart, comm_row, comm_col) :

A_part_d = cp.asarray(A_part)

neighbour_up, neighbour_down = comm_cart.Shift(direction=0, disp=1)
neighbour_left, neighbour_right = comm_cart.Shift(direction=1, disp=1)
Ax_part_temp = empty(M_part, dtype=float64)
Ax_part_temp_recv = empty(M_part, dtype=float64)
r_part_temp = empty(N_part, dtype=float64)
r_part_temp_recv = empty(N_part, dtype=float64)
ScalP_part_temp = empty(1, dtype=float64)
ScalP_part_temp_recv = empty(1, dtype=float64)

requests = [MPI.Requests() for i in range(6)]

requests[0] = comm_cart.Send_init([Ax_part_temp, M_part, MPI.DOUBLE], dest
  =neighbour_right, tag=0)

requests[1] = comm_cart.Recv_init([Ax_part_temp_recv, M_part, MPI.DOUBLE],
   source=neighbour_left, tag=MPI.ANY_TAG)

requests[2] = comm_cart.Send_init([r_part_temp, N_part, MPI.DOUBLE], dest=
  neighbour_down, tag=0)

requests[3] = comm_cart.Recv_init([r_part_temp_recv, N_part, MPI.DOUBLE],
  source=neighbour_up, tag=MPI.ANY_TAG)

requests[4] = comm_cart.Send_init([ScalP_part_temp, 1, MPI.DOUBLE], dest=
  neighbour_right, tag=0)

requests[5] = comm_cart.Recv_init([ScalP_part_temp_recv, 1, MPI.DOUBLE],
  source=neighbour_left, tag=MPI.ANY_TAG)
```

```
s = 1

p_part = zeros(N_part, dtype=float64)

while s<= N
```

Далее, как и в предыдущем примере, перемножения матрицы на вектор выполняем с помощью видеокарт, а скалярные произведения будем реализовывать без их применения

```
if s == 1 :
    x_part_d = cp.asarray(x_part)
    Ax_part_temp = cp.dot(A_part_d, x_part_d).get()
    Ax_part = Ax_part_temp.copy()
    for n in range(num_col-1) :
        MPI.Prequest.Startall([requests[0], requests[1]])
        MPI.Request.Waitall([requests[0], requests[1]],statuses=None)
        Ax_part_temp[:] = Ax_part_temp.recv
        Ax_part = Ax_part + Ax_part_temp
    b_part = Ax_part - b_part
    b_part_d = cp.asarray(b_part)
    r_part_temp = cp.dot(A_part_d.T, b_part_d).get()
    r_part = r_part_temp.copy()
    for m in range(num_row-1) :
        MPI.Prequest.Startall([requests[2], requests[3]])
        MPI.Request.Waitall([requests[2], requests[3]],statuses=None)
        r_part_temp[:] = r_part_temp.recv
        r_part = r_part + r_part_temp
    else :
        ScalP_temp[0] = dot(p_part, q_part)
        ScalP = ScalP_temp.copy()
        for n in range(num_col-1) :
            MPI.Prequest.Startall([requests[4], requests[5]])
            MPI.Request.Waitall([requests[4], requests[5]],statuses=None)
            ScalP_temp[0] = ScalP_temp_recv
            ScalP = ScalP + ScalP_temp
    r_part = r_part - q_part/ScalP
```

Изменения в кусочках программы, для нахождения $p^{(s)}$, $q^{(s)}$ и $x^{(s+1)}$ абсолютно аналогичны. Обратим внимание, что были подобраны достаточно хорошие примеры, которые с помощью этого пакета легко реализуются, потому что все вычисление выполнялись функцией dot, аналог которой есть в пакете cupy. Так же красиво сделать для примеров, которые рассматривались с 7-ой по 10-ую лекции, не получится. Если мы хотим реализовать вычисления с помощью CUDA, нам нужно реализовывать вычислительную часть в виде отдельной функции с помощью либо пакета cupy, либо numba.

Посмотрим на эффективность программной реализации. На одном и том же графике отобразим примеры 12.2, использующий гибридные технологии MPI+OpenMP и пример 13.2, реализованный технологиями MPI, OpenMP и CUDA.

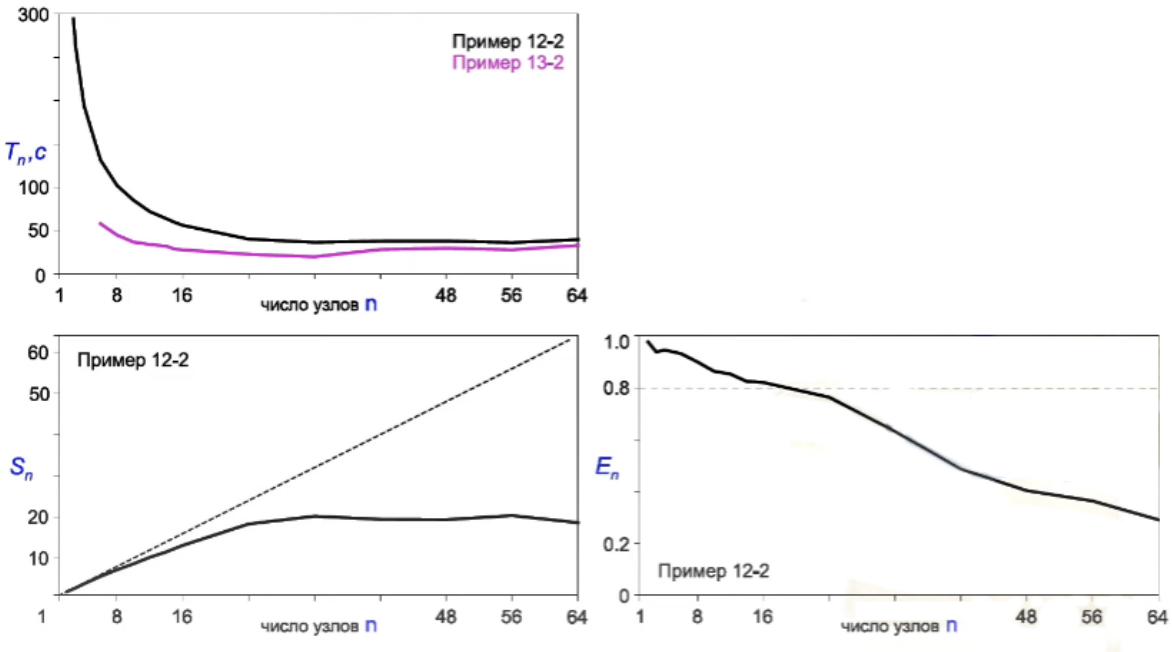


Рис. 13.9

В нашем тестовом примере, матрица примерно 50 ГБ, на одном узле 64 Гб, она влезает, но видеопамять примерно 12 Гб, этого не хватает чтобы полностью ее передать, а для построения графиков ускорения и эффективности необходимо определить время работы программы на одном узле. Поэтому эти графики и не построены.По той же причине график зависимости времени от числа узлов начинается с 6. Можно сделать качественные оценки, мы видим, что использование видеокарты уменьшило время работы мы примерно в 2-2.5 раза, то есть это уже успех. Касательно графиков эффективности, она была бы похуже, так как на нее влияют всегда в первую очередь накладные расходы, а в случае CUDA это прием-передача данных с хоста на девайс и наоборот. Поскольку соответствующие пересылки присутствуют, график эффективности не может быть выше, чем в примере 12.2.

Итак, на этой лекции мы познакомились с некоторыми подходами к использованию гибридных технологий параллельного программирования, относительно подробно обсудили, как оптимизировать вычисления и ускорить их, в том случае, если нам известно, что каждый вычислительный узел представляет из себя достаточно сложную структуру, содержащего и многоядерный процессор, и современную видеокарту. Еще раз обратим ваше внимание, что в подобранных примерах самая вычислительно емкая часть реализо-

вывалась с помощью тех функций пакета numpy, у которых есть аналоги, написанные с использованием технологий CUDA и OpenMP. На практике же следует действовать следующим образом: если все самые вычислительно емкие части программы имеют реализации в виде каких-то функций в пакете, например, cupy или numba, следует использовать их, в противном случае нужно задуматься над тем, чтобы освоить технологию OpenMP или CUDA, чтобы написать самими соответствующие расчетные части, которые вы будете вызывать ваших программах.

Лекция 14. Причины плохой масштабируемости параллельных программ.

На финальной лекции по курсу параллельные вычисления мы подведем итоги, и обсудим типовые ошибки. Основными цельями данного курса было ознакомление со способами ускорения громоздких вычислительных задач, с помощью разбиения на небольшие позадачи и использования нескольких процессов для их вычисления. Мы ожидаем, что при вычислении на N узлах мы получим соответсвующее ускорение в N раз, но это идеализированный случай, который почти не реализуем на практике. И на этой лекции мы обсудим основные причины плохой масштабируемости параллельных программ.

Поэлементная пересылка данных вместо блочной. Последовательный обмен сообщениями вместо одновременного.

В случае поэлементной пересылки сообщений причиной плохой масштабируемости является латентность коммуникационной среды. Это связано с тем, что для отправки сообщения внутри среды требуется некоторое время для его подготовки, поэтому по возможности необходимо передавать блоки наибольшей размерности. Чтобы разобрать типовую ошибку последовательного обмена сообщений вместо одновременного рассмотрим следующий пример с 1 Лекции:

```
if rank == 0:
    for k in range(1, numprocs):
        comm.Recv([buf, source=k, tag=0, status=None])
        ...
        ...
        ...
else:
    comm.Send(buf, dest=0, tag=0)
```

В этом примере отправка сообщений с процессоров может быть произвольным, и например последний процессор может быть готов отправить сообщение раньше первого, но соответствующее сообщение с него не уходит, потому что нулевой процесс ожидает получения сообщения с первого процесса. И в результате часть процессоров может простаивать, и это приводит к дополнительным накладным расходам. Решение этой проблемы выглядит следующим образом:

```
if rank == 0:
    for k in range(1, numprocs):
        comm.Probe(source=MPI.ANY_SOURCE, tag=0, status=status)
        source = status.Get_source()
        comm.Recv(buf, source=source, tag=0, status=None)
        ...
else:
    comm.Send(buf, dest=0, tag=0)
```

Мы добавили метод $Probe$, прозандировали первое сообщение, заполнили структуру статус из нее получили информацию о номере процесса. И далее получили сообщение не от процессора с номером k, а от процессора готового передать сообшение. Рассмотрим следующий пример:

```
if rank == 0:
    comm.Recv(buf, source=1, tag=0, status=None)
if rank in range(1, numprocs - 1):
    comm.Sendrecv(sendbuf,  dest=rank-1, sendtag=0,
        recvbuf, source=rank+1, recvtag=0, status=None)
if rank == numprocs-1:
    comm.Send(buf, dest=numprocs-2, tag=0)
```

В этом примере у нас есть топология процессов типа линейки. Каждому процессу требуется передать информацию с данными на процесс с номером на единицу меньшим, и принять информацию с процесса с номером на еденицу больше. В этом случае все передается одновременно и задержек связанных с приемом передачей сообщений не возникает. Рассмотрим следующий пример, где процессоры также образуют линейку, но они передают данные и правому и левому соседу:

```
if rank == 0:
    comm.Sendrecv(sendbuf,  dest=rank+1, sendtag=0,
        recvbuf, source=rank+1, recvtag=0, status=None)
if rank in range(1, numprocs - 1):
    comm.Sendrecv(sendbuf_1,  dest=rank-1, sendtag=0,
        recvbuf_1, source=rank-1, recvtag=0, status=None)
    comm.Sendrecv(sendbuf_2,  dest=rank+1, sendtag=0,
        recvbuf_2, source=rank+1, recvtag=0, status=None)
if rank == numprocs-1:
    comm.Sendrecv(sendbuf,  dest=rank-1, sendtag=0,
        recvbuf, source=rank-1, recvtag=0, status=None)
```

В случае этой реализации возникает дедлок, предопследний ждет отправкий сообщения с последнего и так далее. И передача сообщения происходит постепенно вне зависимоти

от готовности процессов. Решается данная проблема следующим образом:

```
if rank == 0:
    comm.Sendrecv(sendbuf,  dest=rank+1, sendtag=0,
        recvbuf, source=rank+1, recvtag=0, status=None)
if rank in range(1, numprocs - 1):
    comm.Sendrecv(sendbuf_1,  dest=rank-1, sendtag=0,
        recvbuf_1, source=rank+1, recvtag=0, status=None)
    comm.Sendrecv(sendbuf_2,  dest=rank+1, sendtag=0,
        recvbuf_2, source=rank-1, recvtag=0, status=None)
if rank == numprocs-1:
    comm.Sendrecv(sendbuf,  dest=rank-1, sendtag=0,
        recvbuf, source=rank-1, recvtag=0, status=None)
```

В данной реализации, мы изменили порядок приема и передачи на промежуточных процессорах, и дедлока не возникает. Простое дополнительное задание: усовершенствовать примеры 8-2 и 9-2, построить графики $T(n)$, $S(n)$ и $E(n)$ (времени работы, ускорения и эффективности). Ещё один пример с 10 лекции (54:46):

```
if rank > 0:
    comm.Sendrecv(sendbuf_1,  dest=rank-1, sendtag=0,
        recvbuf_1, source=rank-1, recvtag=0, status=None)
if rank < numprocs - 1:
    comm.Sendrecv(sendbuf_2,  dest=rank+1, sendtag=0,
        recvbuf_2, source=rank+1, recvtag=0, status=None)
```

В этом коде, также происходит обмен с двумя соседями, и присутствует аналогичная ошибка. И аналогичная ситуация возникает в 10 лекции (1:38:11) при делении двухмерным образом. В 12 лекции мы пользовались операциями на отложенное взаимодействие для двумерной декартовой топологии:

```
if my_col > 0:
    MPI.Prequest.Startall([requests[0], requests[1]])
    MPI.Prequest.Waitall([requests[0], requests[1]], statuses=None)
if my_col < num_col-1:
    MPI.Prequest.Startall([requests[2], requests[3]])
    MPI.Prequest.Waitall([requests[2], requests[3]], statuses=None)
if my_row > 0:
    MPI.Prequest.Startall([requests[4], requests[5]])
    MPI.Prequest.Waitall([requests[4], requests[5]], statuses=None)
if my_row < num_row-1:
    MPI.Prequest.Startall([requests[6], requests[7]])
    MPI.Prequest.Waitall([requests[6], requests[7]], statuses=None)
```

Здесь четные запросы отвечали отправки сообщений, а нечетные приему от левого. Это улучшенная реализация прошлого примера, и в этом случае также выстраиваются в цепочку отправки сообщений с левой границы на правую и с верхней на нижнюю. Можно с помощью if реорганизовать блоки, будет работать еще быстрее. Однако можно воспользоваться также тем, что эти сообщения являются асинхронными и получить более органичное решение:

```
    if my_col > 0:
        MPI.Prequest.Startall([requests[0], requests[1]])

    if my_col < num_col-1:
        MPI.Prequest.Startall([requests[2], requests[3]])

    if my_row > 0:
        MPI.Prequest.Startall([requests[4], requests[5]])

    if my_row < num_row-1:
        MPI.Prequest.Startall([requests[6], requests[7]])

    MPI.Request.Waitall(requests, status=None)
```

В этом случае происходят асинхронные передачи и приемы сообщений, а затем мы убеждаемся, что все сообщения получены. Однако в этом случае мы получаем новую проблему приема и передачи сообщений, а именно пересылку большого количества данных. Потому что обмен данными идет одновременно, и вдоль строк, и вдоль столбцов. Может произойти перегрузка коммуникационной сети. Поэтому в случае асинхронных операций нужно быть аккуратными и внимательными к физическим возможностям коммуникационной сети.

Разделение этапов счёта и пересылок.

На примере 12 лекции также рассмотрим проблему разделения этапов счета и пересылок.

```
    if my_col > 0:
        MPI.Prequest.Startall([requests[0], requests[1]])

    if my_col < num_col-1:
        MPI.Prequest.Startall([requests[2], requests[3]])

    if my_row > 0:
        MPI.Prequest.Startall([requests[4], requests[5]])
```

```
    if my_row < num_row-1:
        MPI.Prequest.Startall([requests[6], requests[7]])

    MPI.Request.Waitall(requests, status=None)
```

Область в которой ведуться рассчеты одним процессом является прямоугольником. И для полных рассчетов сеточных значений в этом прямоугольники требуются значения, которые мы получаем путем пересылки с соседних прямоугольников. Однако для узлов которые не находятся на границе внешние сеточные значения нам не требуются. И пока идет пересылка данных с соседних процессов можно выполнить рассчеты во внутренних узлах, а затем рассчитать значения в узлах на границе области рассчетов.

```
    if my_col > 0:
    MPI.Prequest.Startall([requests[0], requests[1]])

    if my_col < num_col-1:
    MPI.Prequest.Startall([requests[2], requests[3]])

    if my_row > 0:
    MPI.Prequest.Startall([requests[4], requests[5]])

    if my_row < num_row-1:
    MPI.Prequest.Startall([requests[6], requests[7]])

    # Start of calculations in internal nodes
    ...
    ...
    ...
    # End of calculations in internal nodes

    MPI.Request.Waitall(requests, status=None)
```

Простое дополнительное задание: усовершенствовать пример 12-1, построить графики $T(n)$, $S(n)$ и $E(n)$. Эффективность в данном случае должна быть близка к еденице, до проявления эффектов связанных с $1/\sqrt{N}$ и $1/N$.

Наиболее распространённые причины плохой масштабируемости параллельных программ.

Примеры выше демонстрируют причины плохой масштабируемости программ связанные с приемом и передачей сообщений между процессорами, фактически относящиеся к нашей программной реализаций. Нужно обратить внимание на другие примеры, связанные с параметрами запуска.

1. Плохое соответсвие топологии вычислений и сетевой топологии. Причина плохой масштабируемости: процессоры сильно связанной программы привязаны к ”далеко отстоящим”по сетевой топологии вычислительным узлам (Лекция 13 43:37). Много раз упоминаемая причина связанная с проекцией нашей программной реализации на реальную физическую топологию.
2. Недостаточная пропускная спсобность системной шины для работы с GPU. Причина плохой масштабируемости: объём вычислений на GPU уменьшается квадратично, а объём передаваемых на GPU данных – линейно(Лекция 13 1:04:21 ; 1:24:50).
3. Работа не через общую память в рамках одного узла. Причина плохой масштабируемости: при компиляции не указан компонент, ответственный за работу через общую память, и процессы внутри одного узла общаются через сеть (Лекция 9 1:14:56).

Общие рекомендации по написанию параллельных программ.

Совет по порядку ознакомления с технологиями паралленльного программирования:

1. Использование коммуникаций между процессорами в рамках коммуникатора $comm \equiv MPI.COMM_WORLD$.
   (a) Операции типа ”точка-точка”.
   (b) Операции коллективного взаимодействия.
2. Использование дополнительных коммуникаторов в т.ч. виртуальных топологий.
3. Использование асинхронных операций.
4. Использование для рассчетной части $C/C++/Fortran$.
5. Использование гибридных технологий параллельного программирования $MPI+$.

(a) $+ OpenMP$.

(b) $+ CUDA(Python : Cupy, Numba)$.

6. Использование профайлеров для определения узких мест программных реализаций.

ФИЗИЧЕСКИЙ
ФАКУЛЬТЕТ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in

ЛЕКЦИИ УЧЕНЫХ МГУ