Последовательная реализация вычисления скалярного произведения векторов

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.

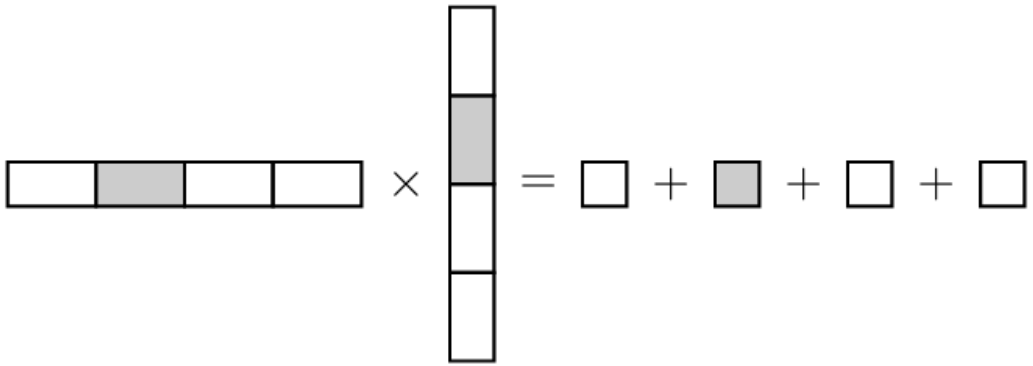

Рис. 2.1: Схема подсчёта скалярного произведения векторов

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

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

```
from numpy import arange

M = 20
a = arange(1, M+1)

ScalP = 0
for j in range(M):
    ScalP += a[j]*a[j]

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.

Параллельная реализация вычисления скалярного произведения векторов

Начнём программу так

```
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

```
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
```

На изображении представлен код, который распределяет задачи между процессами с использованием MPI.

Здесь используется функция divmod, которая делит число M на numprocs - 1 и возвращает два значения:
ave — это целая часть от деления.
res — это остаток от деления

rcounts = empty(numprocs, dtype=int32) — создаётся массив для хранения количества задач, которые будут отправлены каждому процессу.
displs = empty(numprocs, dtype=int32) — создаётся массив для хранения смещений (начальных индексов) частей исходного массива для каждого процесса.

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 рассылаем 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 log2(numprocs-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 сотт и проводит над ними операцию op=MPI .SUM. Результат записывается в переменнуюScalP 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 сот (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 про грамма 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
```