Параллельное Умножение Матрицы на Вектор с Использованием MPI

Распределенное умножение матрицы на вектор с помощью mpi4py для оптимизации вычислений

Введение

Тема: Умножение матрицы на вектор с помощью параллельного программирования на mpi4py.

Задача: Разделить строки матрицы между процессами, передать вектор всем процессам, выполнить умножение на каждом процессе и собрать результаты.

Цель: Повысить эффективность умножения матрицы на вектор за счет распределения работы между процессами.

Умножение Матрицы на Вектор

Математическое Определение

•Если матрица A имеет размер m×n, а вектор B — размерность n, результатом их умножения будет новый вектор C размерности m.

•Каждый элемент результирующего вектора ci вычисляется как

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j$$

$$A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_{1n} \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_1 + a_{12} \cdot b_2 + \dots + a_{1n} \cdot b_n \\ a_{21} \cdot b_1 + a_{22} \cdot b_2 + \dots + a_{2n} \cdot b_n \\ \dots \quad \dots \quad \dots \quad \dots \quad \dots \quad \dots \\ a_{m1} \cdot b_1 + a_{m2} \cdot b_2 + \dots + a_{mn} \cdot b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \dots \\ c_{1m} \end{pmatrix}$$

Пример умножения Матрицы на Вектор

- матрица A имеет размер 5×3,
- вектор B — размерность 3,
- Вектор C размерности 5, результат их умножения.

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \\ 68 \\ 86 \end{pmatrix}$$

Последовательная программа умножения матрицы на вектор

```python
import numpy as np

N = 5  # Количество строк матрицы
M = 3  # Количество столбцов матрицы (длина вектора)

matrix = np.array([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
    [7.0, 8.0, 9.0],
    [10.0, 11.0, 12.0],
    [13.0, 14.0, 15.0]
])

vector = np.array([1.0, 2.0, 3.0])

result = np.dot(matrix, vector)

print("Матрица (5x3):")
print(matrix)
print("\nВектор (длины 3):")
print(vector)
print("\nРезультат умножения матрицы на вектор:")
print(result)
```

Output:

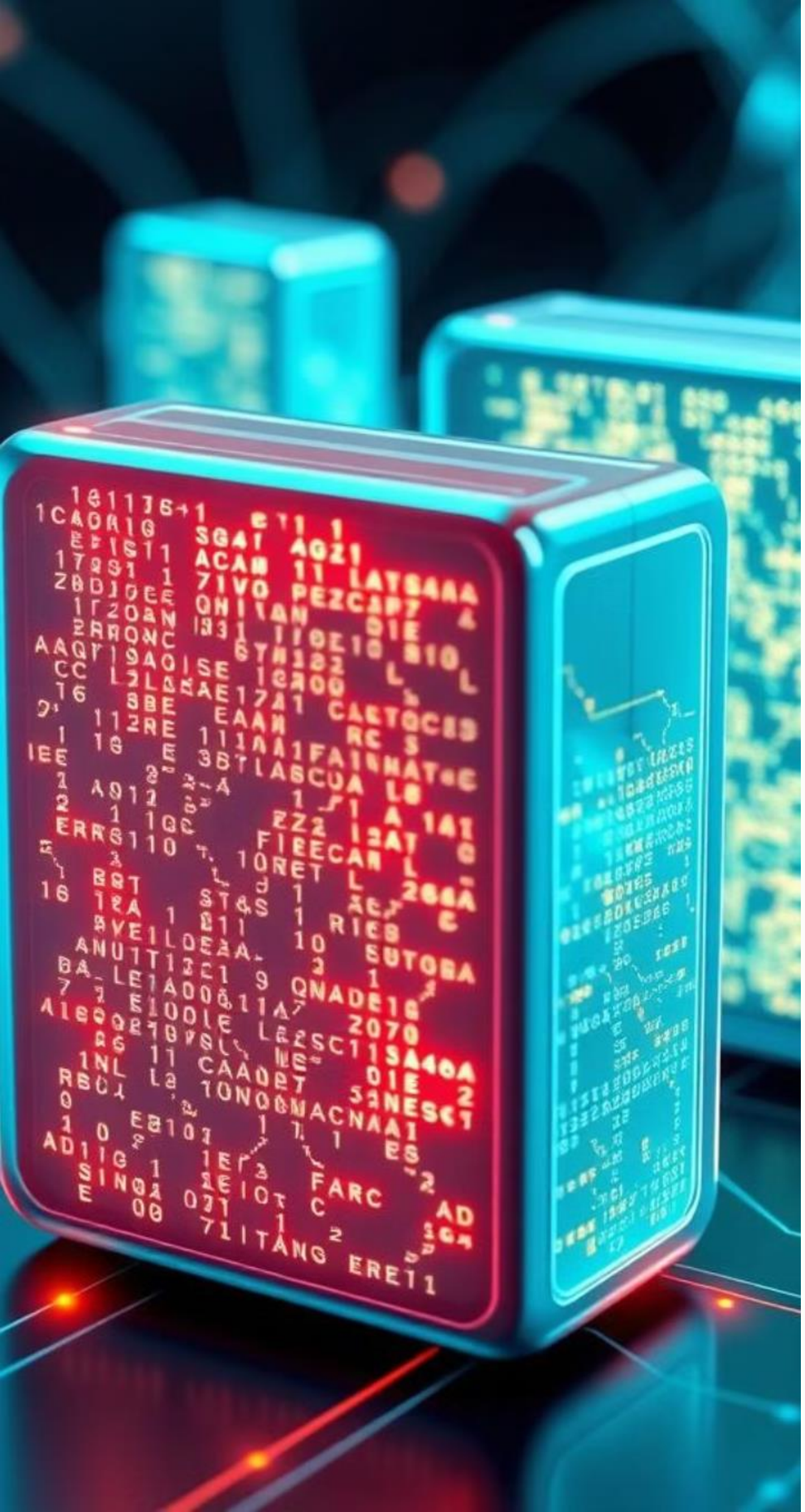
Output

```
Матрица (5x3):
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]
 [10. 11. 12.]
 [13. 14. 15.]]

Вектор (длины 3):
[1. 2. 3.]

Результат умножения матрицы на вектор:
[14. 32. 50. 68. 86.]

=== Code Execution Successful ===
```



Параллельная программа умножения матрицы на вектор

Общая Структура Решения

1. Чтение данных

Матрица и вектор загружаются из файлов.

2. Распределение данных

Строки матрицы распределяются между процессами, а вектор передается полностью всем процессам.

3. Локальное умножение

Каждый процесс умножает свою часть матрицы на вектор.

4. Сбор результатов

Частичные результаты от каждого процесса собираются в процесс 0 для формирования окончательного результата.

Описание Входных Данных

Файл in.dat

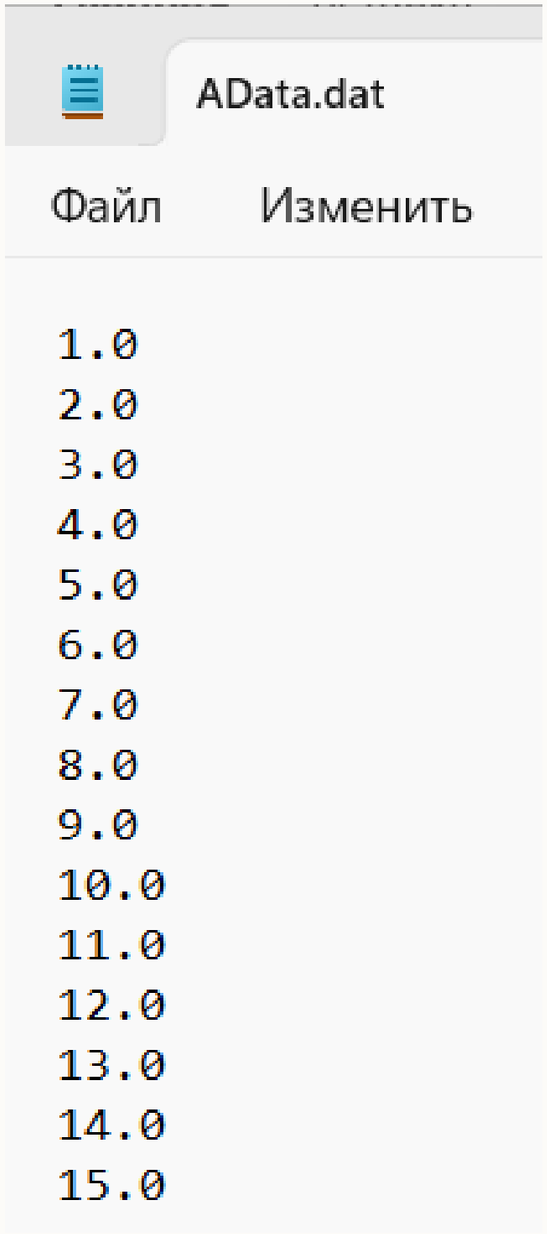
Содержит размеры матрицы: о Первое число — количество строк матрицы N. о Второе число — количество столбцов матрицы M.

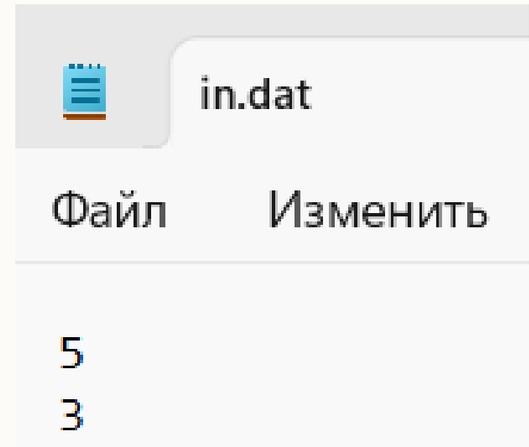
Файл AData.dat

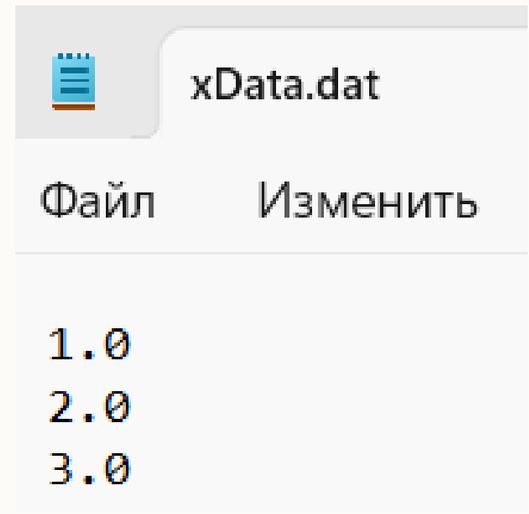
Содержит элементы матрицы размером N x M, записанные построчно.

Файл xData.dat

Содержит элементы вектора длины M.







Основные Этапы Программы

Шаг 1 Чтение размеров матрицы и вектора.

Шаг 2 Передача размеров другим процессам.

Шаг 3 Определение количества строк, обрабатываемых каждым процессом.

Шаг 4 Чтение строк матрицы и их распределение между процессами.

Шаг 5 Передача полного вектора каждому процессу.

Шаг 6 Локальное умножение части матрицы на вектор.

Шаг 7 Сбор частичных результатов и объединение их в итоговый вектор.

Код - Инициализация и Передача Размеров

```
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)
    M = array(0, dtype=int32)

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

- Описание: Процесс 0 читает размеры матрицы и передает их всем процессам.

Код - Распределение Строк Матрицы

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

    for k in range(1, numprocs):
        rcounts[k] = ave + 1 if k <= res else ave
        displs[k] = displs[k - 1] + rcounts[k - 1]
else:
    rcounts = empty(numprocs, dtype=int32)
    displs = empty(numprocs, dtype=int32)

comm.Bcast(rcounts, root=0)
comm.Bcast(displs, root=0)
```

• Описание: Каждый процесс получает информацию о количестве строк, которые он будет обрабатывать, и смещения для подматриц.

Код - Передача Строк Матрицы

```
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], M), dtype=float64)
        for j in range(rcounts[k]):
            for i in range(M):
                A_part[j, i] = float64(f2.readline())
        comm.Send([A_part, rcounts[k] * M, MPI.DOUBLE], dest=k,
tag=0)
    f2.close()
    A_part = empty((rcounts[rank], M), dtype=float64)
else:
    A_part = empty((rcounts[rank], M), dtype=float64)
    comm.Recv([A_part, rcounts[rank] * M, MPI.DOUBLE], source=0)
```

- Описание: Строки матрицы передаются процессам с помощью Send И Recv

Код - Передача Вектора и Умножение

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

comm.Bcast(x, root=0)

# Локальное умножение
b_temp = dot(A_part, x)
```

• Описание: Вектор передается всем процессам, и каждый процесс выполняет локальное умножение.

Код - Сбор Результатов

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

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

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

• Описание: Частичные результаты собираются в процессе 0 с помощью Gatherv, где они объединяются в итоговый результат.

Результат

```
PS C:\Users\Admin> mpiexec -n 6 "C:\Program Files\Python312\python.exe" MxV.py
[14. 32. 50. 68. 86.]
PS C:\Users\Admin>
```

Результаты вычислений параллельной программы соогласуются с результатами вычислений последовательной программы умножения матрицы на вектор

Выводы

В коде использовались функции `Bcast`, `Scatter` и `Gatherv` для передачи данных, распределения строк матрицы между процессами и сбора частичных результатов. Эти функции обеспечили эффективное распределение данных и агрегацию результатов. В качестве альтернатив можно использовать `Scatterv`, если данные неравномерно распределены, или `Allreduce`, если требуется, чтобы все процессы получили итоговый результат. Также полезной является функция `Reduce_scatter`, которая одновременно выполняет агрегацию и распределение, что может сократить накладные расходы при передаче данных. Выбор конкретных MPI функций зависит от особенностей задачи и структуры данных, и правильная комбинация функций позволяет достичь баланса между производительностью и сложностью реализации.