Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 2

по курсу «Разработка параллельных и распределенных программ»

Решение систем линейных алгебраических уравнений итерационными методами

Студент: Пишикина М.В.

Группа: ИУ9-51Б

Преподаватель: Царев А.С.

Москва 2024

Содержание

1 Постановка задачи

1) Написать программу, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double.

2) Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы: 1: векторы x и b дублируются в каждом MPI-процессе, 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A. (только для сдающих после срока) Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

3) Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Также параметр N разрешено подобрать таким образом, чтобы он нацело делился на на 1,2,4,8 и 16.

4) На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

2 Практическая реализация

```
from mpi4py import MPI
import numpy as np

N = 30000
```

```
epsilon = 1e-5
tau = 0.1 / (N)

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

def minimal_residual_method(A_local, b_local, x, counts, displacements):
    iteration = 150

    for i in range(iteration):
        # y_n = A_local * x - b_local : локальные значения
        y_local = A_local.dot(x) - b_local

        y = np.zeros(N)

        # норма невязки
        comm.Allgatherv(y_local, [y, counts, displacements, MPI.DOUBLE])
        residual_norm = np.linalg.norm(y_local)
        global_residual_norm = np.sqrt(comm.allreduce(residual_norm ** 2,
        ↪   op=MPI.SUM))

        if rank == 0:
            print(f"Iteration {i + 1}: residual norm = {global_residual_norm}")

        if residual_norm < epsilon:
            if rank == 0:
                print(f"Solution found after {i + 1} iterations with residual norm
                ↪   {residual_norm}.")
            #break

        # Ay_n = A_local * y
        Ay_local = A_local.dot(y)
```

```
        Ay = np.zeros(N)
        comm.Allgatherv(Ay_local, [Ay, counts, displacements, MPI.DOUBLE])

        # tau_n = (y, Ay) / (Ay, Ay)
        tau_numerator = np.dot(y, Ay)
        tau_denominator = np.dot(Ay, Ay)

        if tau_denominator < 1e-12:
            if rank == 0:
                print("Знаменатель для tau слишком мал")
            break

        tau = tau_numerator / tau_denominator

        # x_(n+1) = x_n - tau * y : Обновление решения
        x = x - tau * y

    return x, i + 1


def main():
    start_time = MPI.Wtime()

    # rank == 0 - главный процесс
    if rank == 0:
        A = np.full((N, N), 1.0)
        np.fill_diagonal(A, 2.0)
        b = np.full(N, N + 1, dtype=np.double)
        x0 = np.full(N, 0.1, dtype=np.double)
    else:
        A = None
        b = None
```

```
    x0 = None

# распространение от главного к остальным
x0 = comm.bcast(x0, root=0)
b = comm.bcast(b, root=0)

# Вычисление размеров локального массива
counts = np.full(size, N // size, dtype=int)
counts[:N % size] += 1
displacements = np.cumsum(counts) - counts

# Разбросать строки A по всем процессам
local_n = counts[rank]
A_local = np.zeros((local_n, N), dtype=np.float64)
send_counts = counts * N
send_displacements = displacements * N

comm.Scatterv([A, send_counts, send_displacements, MPI.DOUBLE],
 ↪  A_local, root=0)

b_local = np.copy(b)
x = np.copy(x0)

x, iterations = minimal_residual_method(A_local, b_local, x, counts,
 ↪  displacements)

end_time = MPI.Wtime()

if rank == 0:
    elapsed_time = end_time - start_time
    print(f"Время выполнения: {elapsed_time:.6f} секунд")
    print(f"Количество используемых процессоров: {size}")
    print(f"Финальный x: {x[:10]} ...")
```

```
        # print("Ожидаемый x: ", np.ones(N)[:10])  # Print expected result
        print("Решение верно:", np.allclose(x, np.ones(N), atol=epsilon))

if __name__ == "__main__":
    main()
```

3 Характеристика устройства

Устройство: MacBook Pro 2020

Операционная система: macOS Sonoma

Процессор: Intel Core i5

Характеристика процессора: 4-ядерный чип, частота 2 ГГц

Оперативная память: 16GB LPDDR4X

4 Время работы программы

Взято N = 30000 , чтобы протестировать работу программы более точно.

Время работы при 1 процессоре: 28.731112s

Время работы при 2 процессорах: 25.263491s

Время работы при 4 процессорах: 24.8330473s

Время работы при 8 процессорах: 26.5138931s

Время работы при 16 процессорах: 27.484934s

Время работы при 32 процессорах: 29.4598234s

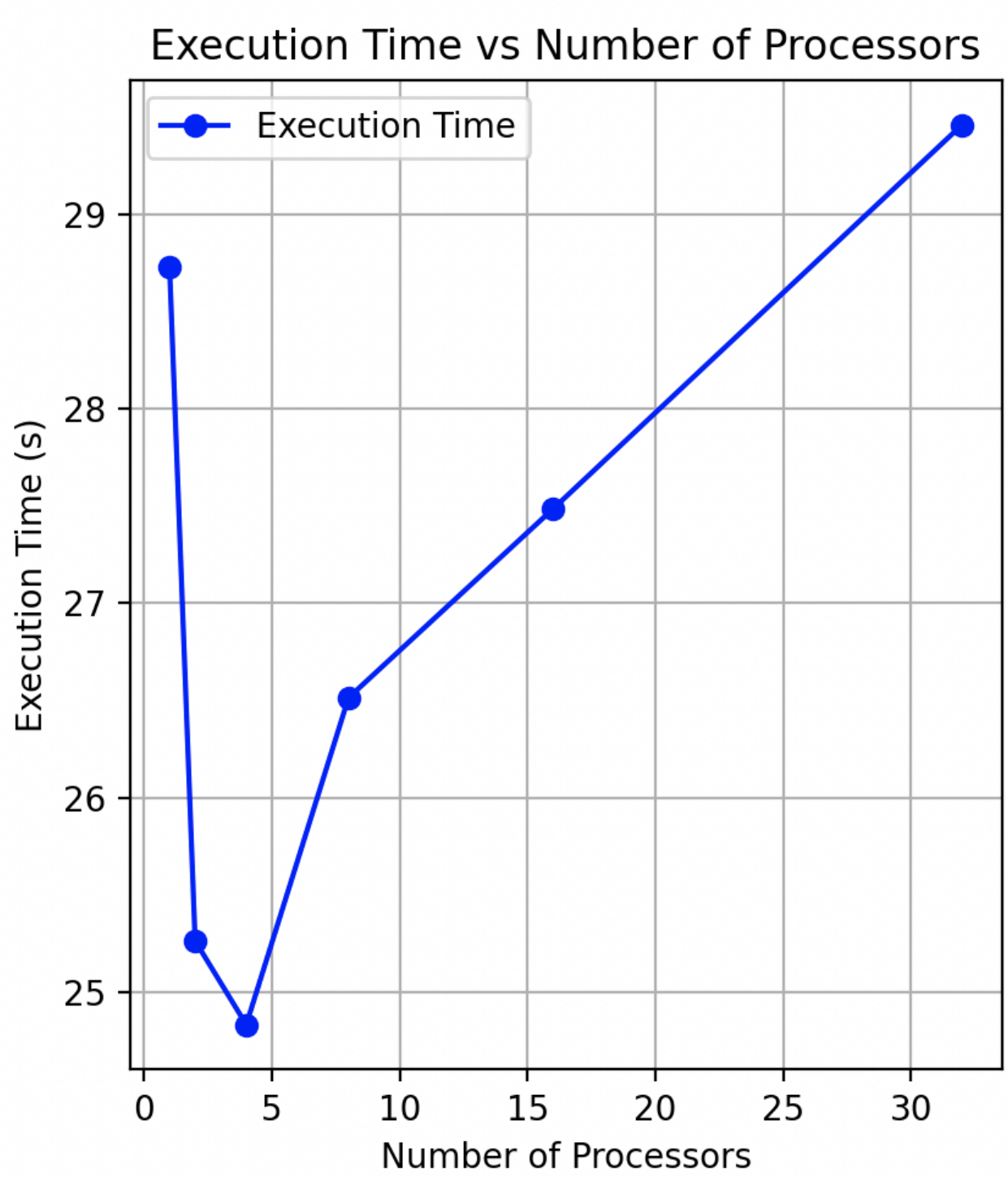

Рис. 1 — График: Зависимость времени выполнения от количества процессоров

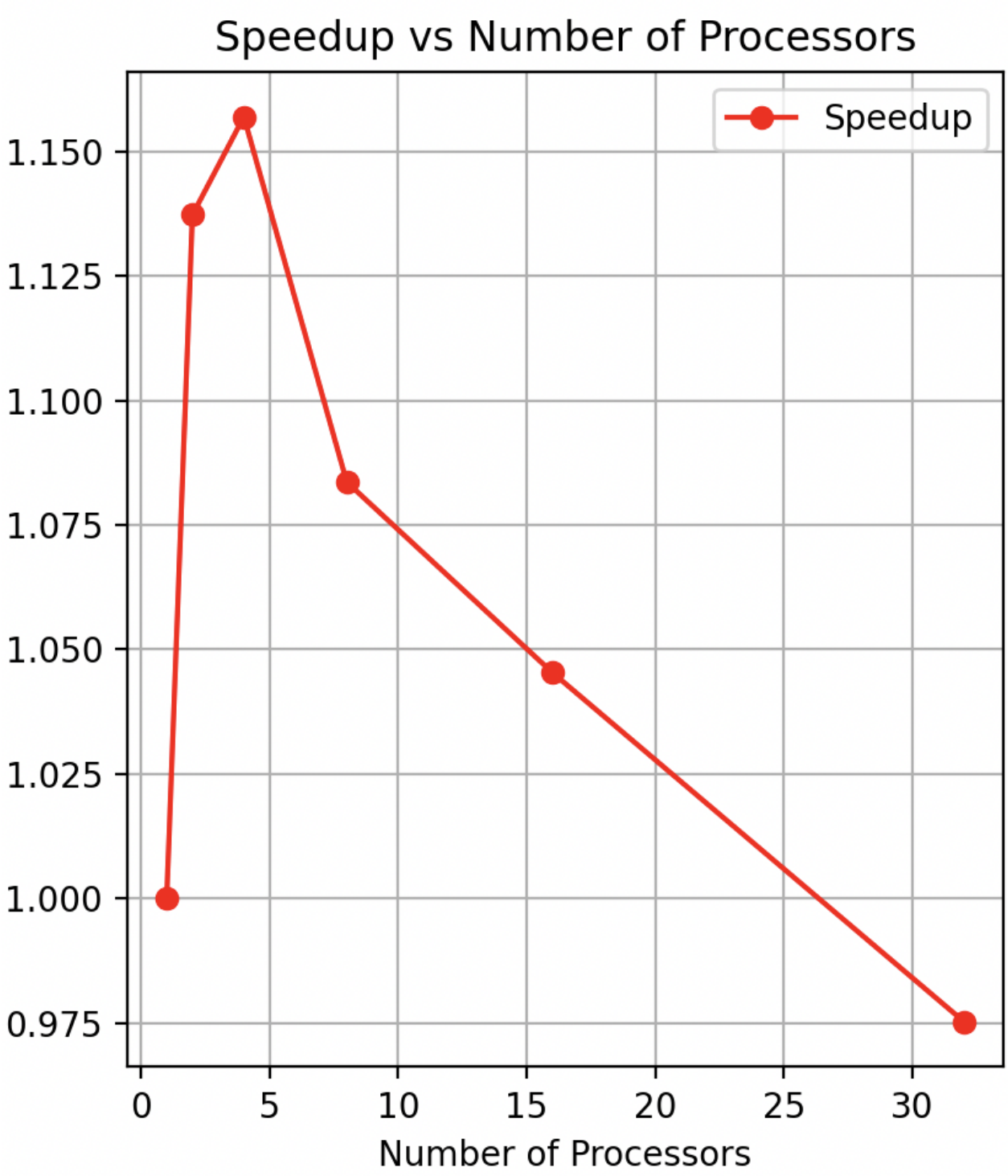

Рис. 2 — График ускорения

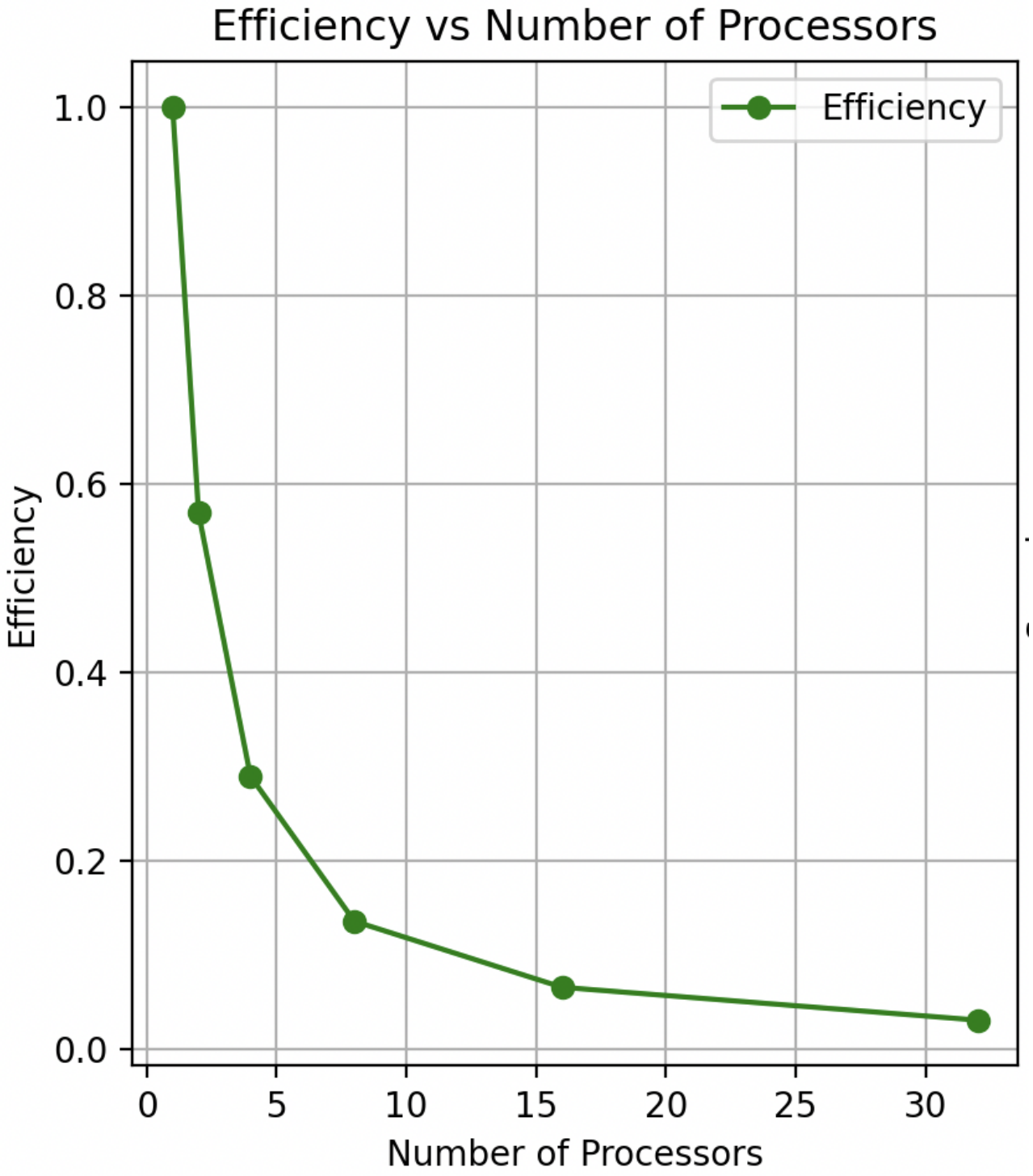

Рис. 3 — График эффективности

5 Вывод

По графикам видно, что время выполнения программы уменьшается с увеличением числа процессоров, что подтверждает применимость метода распарал-

леливания с использованием MPI для данной задачи. Однако при использовании 8, 16 и 32 процессоров время выполнения начинает увеличиваться. Это связано с тем, что ноутбук, на котором проводилось тестирование, имеет только 4 физических процессора. Соответственно, при увеличении количества процессов сверх этого значения эффективность снижается из-за накладных расходов на управление виртуальными процессами и обмен данными между ними.