

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №6
по дисциплине «Параллельные алгоритмы»
Тема: Умножение матриц.

Студент гр. 3388

Дубровин Д.Н.

Преподаватель

Татаринов Ю.С.

Санкт-Петербург

2025

Цель работы.

Изучить параллельные алгоритмы перемножения матриц и реализовать их на практике. Установить зависимость времени выполнения программы от числа процессов и размеров входным матриц. Построить сеть Петри для реализованного параллельного алгоритма.

Задание.

Выполнить задачу умножения двух квадратных матриц A и B размера $m \times m$, результат записать в матрицу C . Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Вариант 1: Ленточный алгоритм 1 (горизонтальные полосы).

Выполнение работы.

1. Описание алгоритма:

Вначале производится рассылка в процесс ранга K элементов K -й строки матрицы A и элементов K -й строки матрицы B . Элементы строки s , в которой будет содержаться соответствующая строка произведения AB результат, обнуляются. Затем запускается цикл (число итераций равно N), в ходе которого выполняются два действия: Основы параллельного программирования: алгоритмы умножения матриц 2 1) выполняется перемножение элементов строк матрицы A и матрицы B с одинаковыми номерами, и результаты добавляются к соответствующему элементу строки s ; 2) выполняется циклическая пересылка строк матрицы B в соседние процессы (направление пересылки может быть произвольным: как по возрастанию рангов процессов, так и по их убыванию). После завершения цикла в каждом процессе будет содержаться соответствующая строка произведения AB . Останется переслать эти строки главному процессу.

2. Построим сеть Петри.

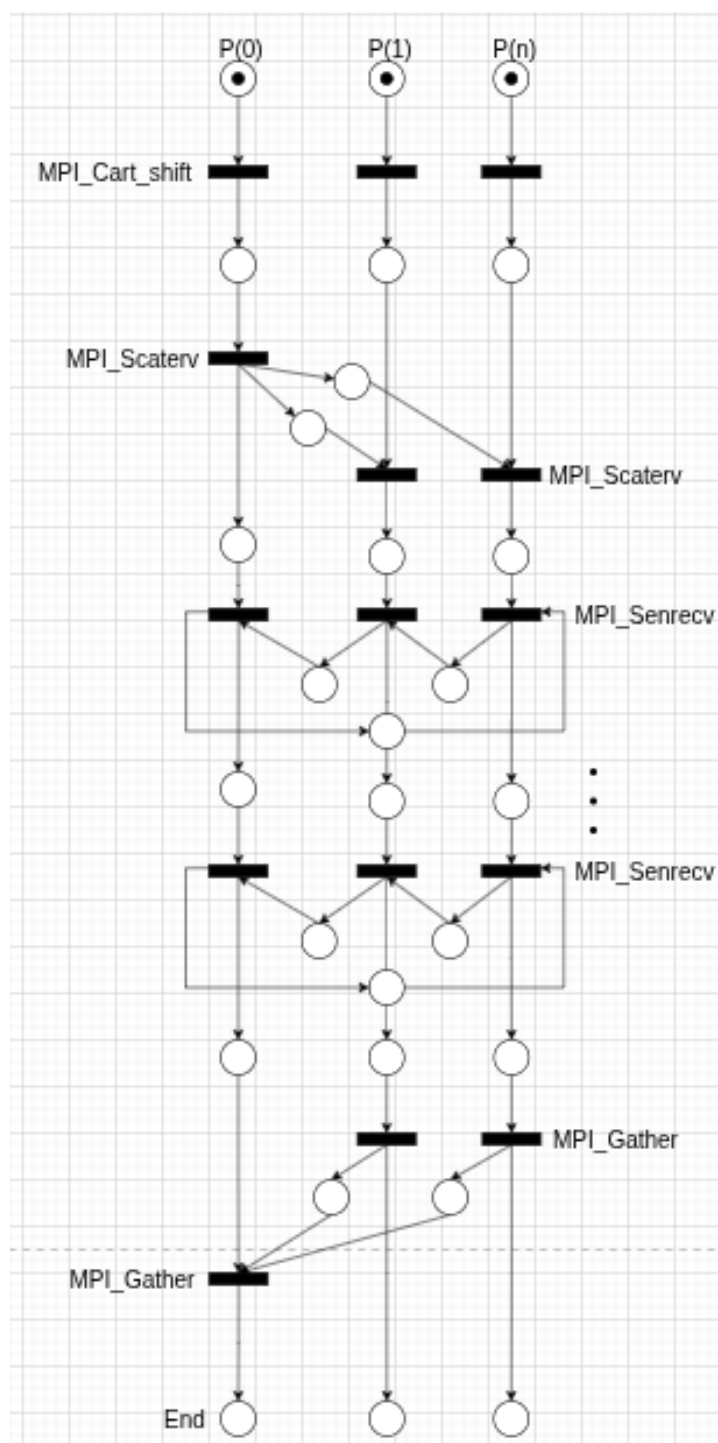


Рис.1 - сеть Петри

```

• > mpirun --oversubscribe -np 3 ./parallel 9
Matrix multiplication of 9x9 matrices using 3 processes
Matrix A:
  5  -7   3  -2   7  -5   0   7   6
  9 -10 -10   4   9  -7   0   7  -7
 -8   3   5  -1   8  -4  -8  -6  -9
 -1   4  10   7  -3  -8  -3   5   8
  3  -8   2  -4 -10   1  -6   5  -1
 -3  -8  -5  -2  -8   6  -9   2   1
 -3   4   5  -2  -8  -1  -2   7   6
 -2  -7  -2  -7   4   0   6  10  -1
 -4  -9  -9   5  -2  -7   8   5  -4

Matrix B:
  1   6   8  -8 -10  -1   5  -2  -1
 -9   4  -7   5  -8  -6  -8  -7   9
  1   9   6  10  -6   8 -10   7  -7
 -9   2   6  -5  -7  -1  -8  -7  -1
 -1   8   8   6  10  -1   9   2  -1
 -7   3   3  10  -7   1   5 -10   4
  0   9  -2   2  10 -10  -3  -5  -7
  7  -5  -6  -5   3  -9 -10  -2   9
 -2  -3   1   7  -2  -7  -1  -3  -6

Matrix C (result):
154  13  100  -36  116  -54  29  106  -87
156 -31  108 -342  196  -63  176  49  35
-25  44   34   93   61  216  39  172  33
-12 -10    2   20 -136  -26 -309  50  -51
153 -157  -28 -118 -100   98  -25  85  22
 60 -239  -49  -65  -70   92   79   3  75
 36 -99 -133   49 -102  -57 -228  -1  51
190 -33  -62  -25  269 -112   41  49  10
117 -109  -80 -247  263 -121  -21   1 -32

Num of proc: 3 | Execution time: 0.000158 seconds

```

Рис.2 — пример работы программы

3. Проведём запуск при разном количестве процессов и размерностях матриц.

Таблица 1- результаты выполнения программы при различном количестве процессов и размерностях матриц.

Размерность матриц	Последовательный алгоритм	2 процесса	5 процесса	10 процессов	25 процессов	50 процессов
10	0.000003	0.000031	0.000063	0.000159	-	-
50	0.000363	0.000242	0.000241	0.000314	0.002367	0.007391
100	0.001905	0.001491	0.001306	0.000878	0.002902	0.011872
500	0.234574	0.170902	0.093769	0.071876	0.082673	0.137202
1000	2.111620	1.412257	0.614803	0.623011	0.885051	0.844760

4. Построим графики времени выполнения и ускорения для 1-50 процессов при матрицах размерностью 10, 100, 1000.

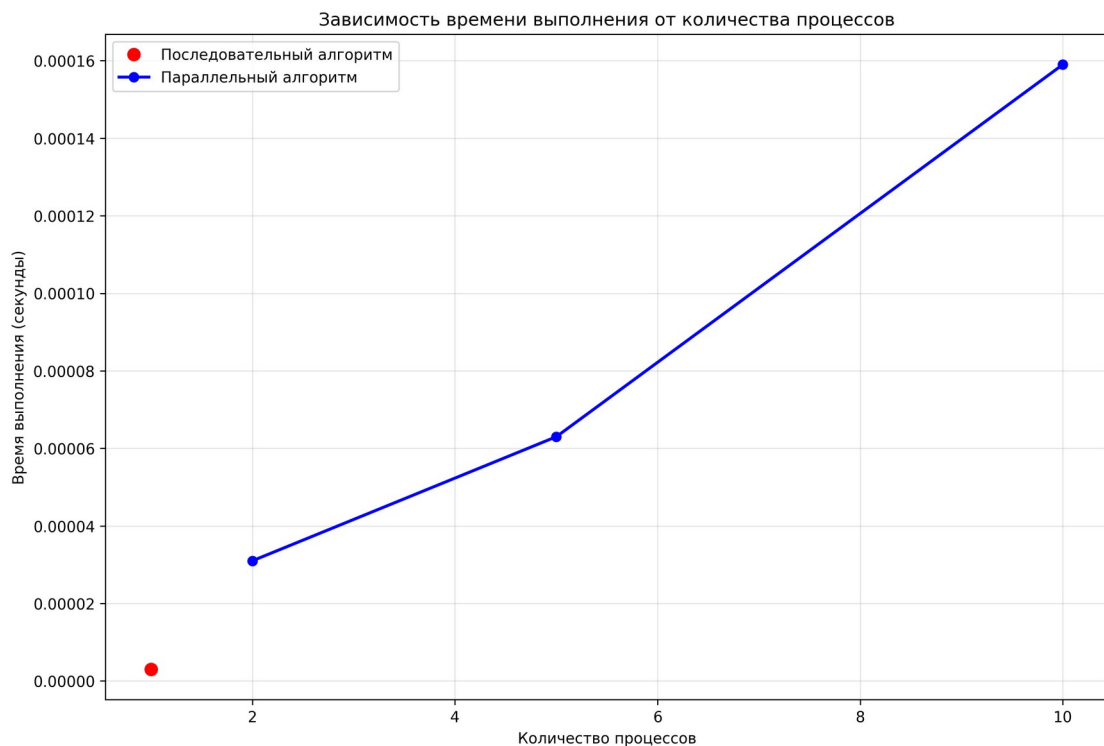


Рис.3 — матрица 10x10

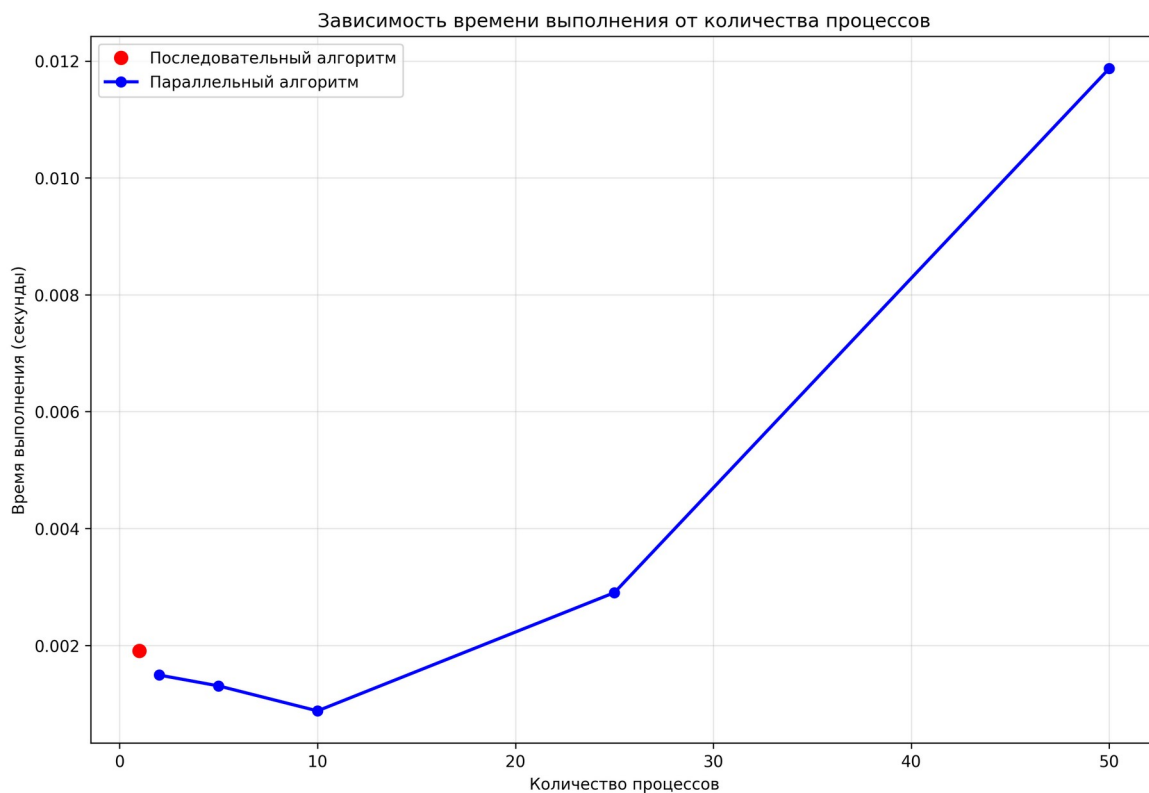


Рис.4 — матрица 100x100

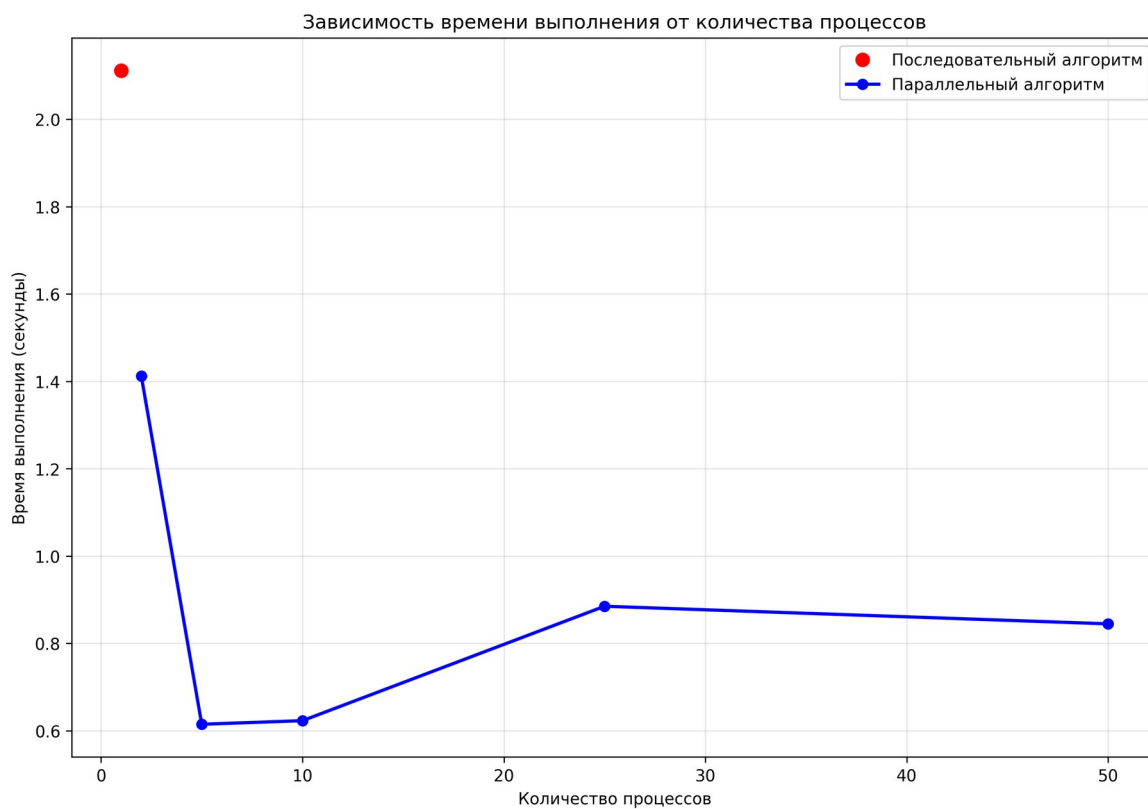


Рис.5 — матрица 1000x1000

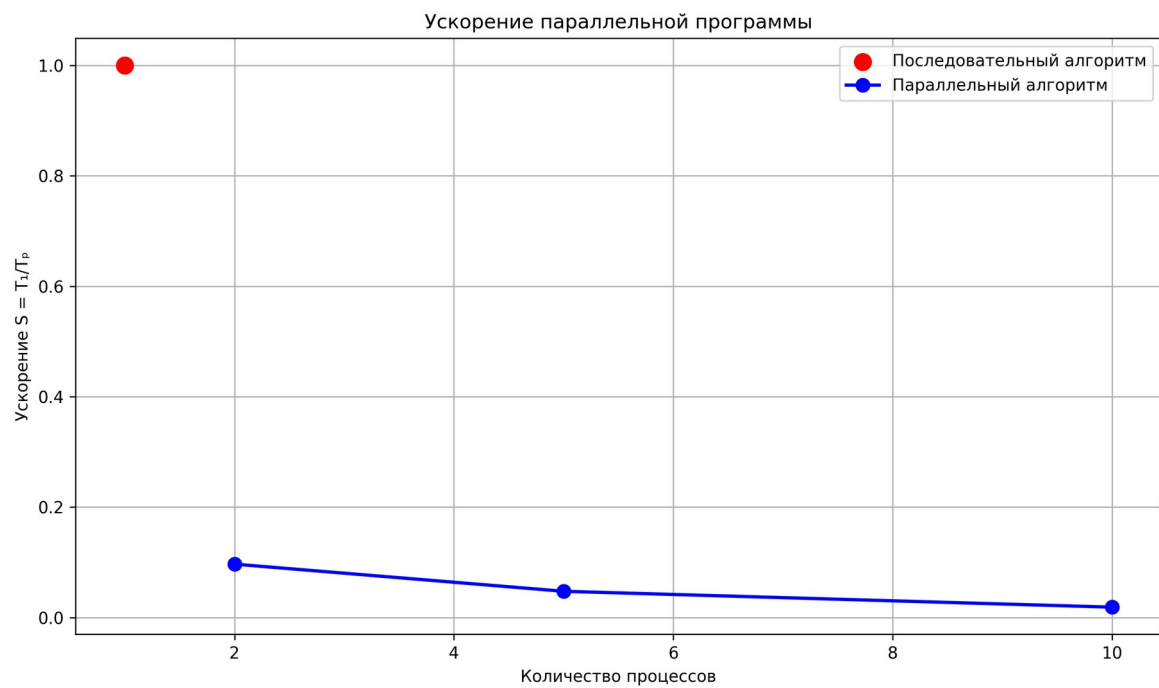


Рис.6 — матрица 10x10



Рис.7 — матрица 100x100

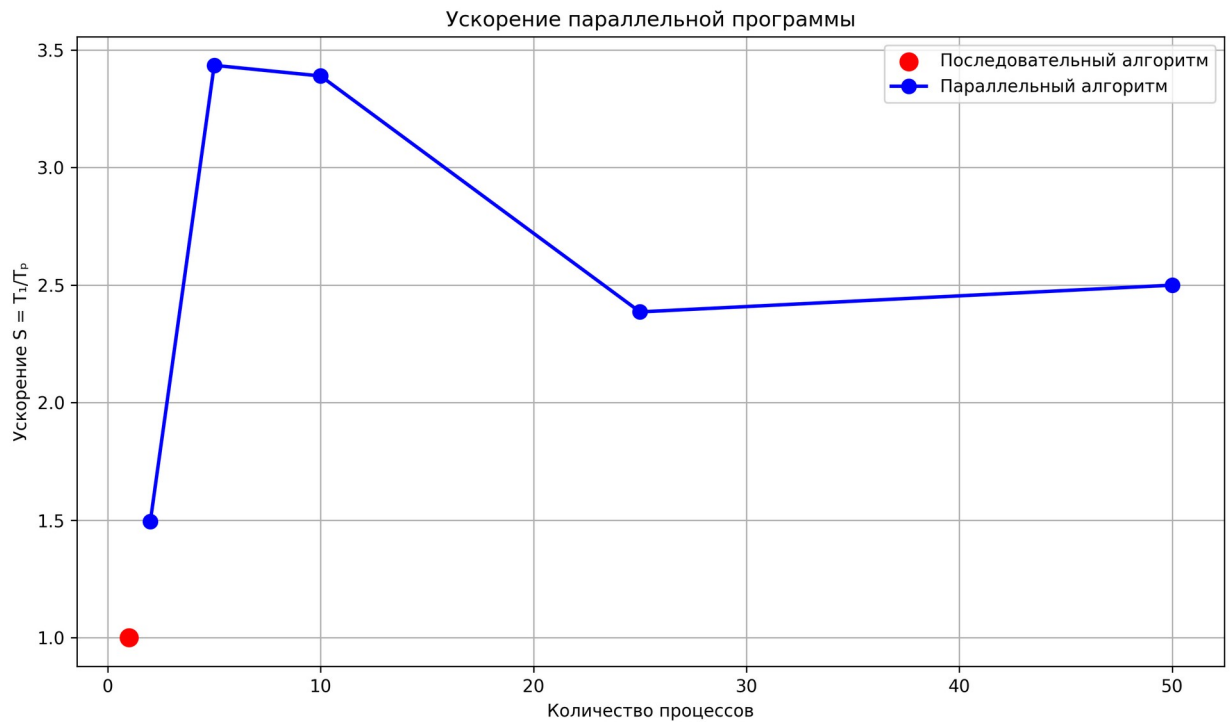


Рис.7 — матрица 1000x1000

Теоретический расчёт.

Воспользуемся законом Амдала, разделим алгоритм на последовательную и параллельную часть.

$$Speedup_{parallel}(f,n) = \frac{1}{(1-f) + \frac{f}{n}}$$

Рис. 1. Закон Амдала

$Speed up$ – относительное ускорение;
 f – часть кода, которая может быть распараллелена; n – число параллельных процессоров

Рис.8 — закон Амдала

Будем рассматривать асимптотику работы с матрицами, так как она лучше всего поддаётся анализу.

- Последовательная часть $(1 - f)$: заполнение матриц случайными числами. Сложность $O(N^2)$.

- Параллельная часть (f): тройной вложенный цикл умножения матриц. Сложность $O(N^3)$.

Доля параллельного кода:

$$f = N^3 / (N^2 + N^3) = N / (N + 1)$$

Таким образом чем больше размерность матрицы N, тем ближе f к 1 (идеальному параллелизму)

Найдем S для 10 процессов на матрице 100x100:

$$S(10) = 9.01$$

Данный прогноз далёк от реальности, так как не учитывает затраты MPI на распараллеливание программы.

Анализ результатов.

Проведенный анализ демонстрирует, что уже для матрицы размером 100×100 параллельный алгоритм показывает существенное преимущество над последовательной реализацией. Максимальное ускорение достигается при использовании до 10 вычислительных процессов. Дальнейшее увеличение количества процессов приводит к снижению производительности, что объясняется ростом накладных расходов на синхронизацию потоков, организацию межпроцессного взаимодействия и многократное выделение памяти.

Разработанный программный код см. в приложении А.

Выводы.

В ходе работы были реализованы параллельный (ленточный) и последовательный алгоритмы для переумножения матриц. Для оптимизации параллельного алгоритма были задействованы виртуальные топологии. Был проведён анализ скорости выполнения программы от количества запущенных процессов, который показал эффективность параллельной реализации на больших матрицах.

Приложение А

Исходный код программы

Название файла: parallel.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define MAX_VISIBLE_SIZE 10

void init_matrix(int *matrix, int rows, int cols);
void print_matrix(int *matrix, int rows, int cols, char *name);
void panic(char *message, MPI_Comm comm);

int main(int argc, char **argv)
{
    int rank, size, N;
    int *A = NULL, *B = NULL, *C = NULL;
    int *A_local, *B_local, *C_local, *B_temp;

    MPI_Comm cart_comm;
    int dims[1];
    int periods[1];
    int source_rank, dest_rank;

    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    dims[0] = world_size;
    periods[0] = 1;

    MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, 1, &cart_comm);

    MPI_Comm_rank(cart_comm, &rank);
    MPI_Comm_size(cart_comm, &size);

    if (argc != 2) {
```

```

        if (rank == 0) printf("Usage: mpirun -np <num_processes>
./parallel <matrix_size>\n");
        MPI_Finalize();
        return 1;
    }

    N = atoi(argv[1]);
    if (N <= 0) {
        panic("Matrix size must be positive.\n", cart_comm);
    }
    if (N % size != 0) {
        panic("Matrix size must be divisible by number of processes.\n",
cart_comm);
    }

    int rows_per_proc = N / size;

    if (rank == 0) {
        A = (int*)malloc(N * N * sizeof(int));
        B = (int*)malloc(N * N * sizeof(int));
        C = (int*)malloc(N * N * sizeof(int));

        srand(time(NULL));
        init_matrix(A, N, N);
        init_matrix(B, N, N);

        printf("Matrix multiplication of %dx%d matrices using %d
processes\n", N, N, size);
        if (N <= MAX_VISIBLE_SIZE) {
            print_matrix(A, N, N, "A");
            print_matrix(B, N, N, "B");
        }
    }

    A_local = (int*)malloc(rows_per_proc * N * sizeof(int));
    B_local = (int*)malloc(rows_per_proc * N * sizeof(int));
    B_temp = (int*)malloc(rows_per_proc * N * sizeof(int));
    C_local = (int*)calloc(rows_per_proc * N, sizeof(int));

    double start_time = MPI_Wtime();

```

```

        MPI_Scatter(A, rows_per_proc * N, MPI_INT, A_local, rows_per_proc *
N, MPI_INT, 0, cart_comm);
        MPI_Scatter(B, rows_per_proc * N, MPI_INT, B_local, rows_per_proc *
N, MPI_INT, 0, cart_comm);

MPI_Cart_shift(cart_comm, 0, 1, &source_rank, &dest_rank);

/* Ленточный алгоритм */
for (int iter = 0; iter < size; iter++) {
    for (int i = 0; i < rows_per_proc; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < rows_per_proc; k++) {
                int current_b_block = (rank - iter + size) % size;
                int col_A = current_b_block * rows_per_proc + k;
                sum += A_local[i * N + col_A] * B_local[k * N + j];
            }
            C_local[i * N + j] += sum;
        }
    }

    if (iter < size - 1) {
        for (int i = 0; i < rows_per_proc * N; i++) {
            B_temp[i] = B_local[i];
        }

        MPI_Sendrecv(B_temp, rows_per_proc * N, MPI_INT, dest_rank,
0,
                        B_local, rows_per_proc * N, MPI_INT,
source_rank, 0,
                        cart_comm, MPI_STATUS_IGNORE);
    }
}

MPI_Gather(C_local, rows_per_proc * N, MPI_INT,
        C, rows_per_proc * N, MPI_INT, 0, cart_comm);

double end_time = MPI_Wtime();

if (rank == 0) {
    if (N <= MAX_VISIBLE_SIZE) {

```

```

        print_matrix(C, N, N, "C (result)");
    }
    printf("Num of proc: %d | Execution time: %.6f seconds\n",
        size, end_time - start_time);

    free(A);
    free(B);
    free(C);
}

free(A_local);
free(B_local);
free(B_temp);
free(C_local);

MPI_Comm_free(&cart_comm);

MPI_Finalize();
return 0;
}

void init_matrix(int *matrix, int rows, int cols)
{
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = (rand() % 21) - 10;
    }
}

void print_matrix(int *matrix, int rows, int cols, char *name)
{
    printf("Matrix %s:\n", name);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%4d ", matrix[i * cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

void panic(char *message, MPI_Comm comm)

```

```

{
    int rank;
    MPI_Comm_rank(comm, &rank);
    if (rank == 0) {
        printf("Error: %s", message);
    }

    MPI_Abort(comm, 1);
}

```

Название файла: serial.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_VISIBLE_SIZE 10

void init_matrix(int *matrix, int rows, int cols);
void print_matrix(int *matrix, int rows, int cols, char *name);

int main(int argc, char *argv[]) {
    int N;
    if (argc == 2) {
        N = atoi(argv[1]);
    } else {
        printf("Usage: %s <matrix_size>\n", argv[0]);
        return 1;
    }

    if (N <= 0) {
        printf("Error: Matrix size must be positive.\n");
        return 1;
    }

    int *A = (int*)malloc(N * N * sizeof(int));
    int *B = (int*)malloc(N * N * sizeof(int));
    int *C = (int*)malloc(N * N * sizeof(int));

    if (!A || !B || !C) {
        printf("Memory allocation failed!\n");
        return 1;
    }
}

```



```

    }

    srand(time(NULL));
    init_matrix(A, N, N);
    init_matrix(B, N, N);

    printf("Sequential matrix multiplication of %dx%d matrices\n", N,
N);

    if (N <= MAX_VISIBLE_SIZE) {
        print_matrix(A, N, N, "A");
        print_matrix(B, N, N, "B");
    }

    clock_t start = clock();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    if (N <= MAX_VISIBLE_SIZE) {
        print_matrix(C, N, N, "C (result)");
    }

    printf("Time elapsed: %.6f seconds\n", time_spent);

    free(A);
    free(B);
    free(C);

    return 0;
}

```

```

void init_matrix(int *matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = (rand() % 21) - 10;
    }
}

void print_matrix(int *matrix, int rows, int cols, char *name) {
    printf("Matrix %s:\n", name);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%4d ", matrix[i * cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

```