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.

Студентка гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2022

Цель работы.

Написать программы, реализующую последовательный и ленточный алгоритмы перемножения матриц. Провести анализ эффективности, сравнить теоретические и экспериментальные оценки.

Задание.

Вариант 1.

Выполнить задачу умножения двух квадратных матриц A и B размера m × 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. Ленточный алгоритм (горизонтальные полосы).

Выполнение работы.

В ходе программ – создаются две матрицы и инициализируются случайным образом в диапазоне значений 0-149.

Разработанный программный код см. в Приложении А.

1. Последовательный алгоритм:

Производится прямое перемножение матриц – пошагово строка на столбец. Алгоритм ведёт расчёт через три вложенных цикла.

Результаты работы программы на 1 процессе. (Последовательный алгоритм.)

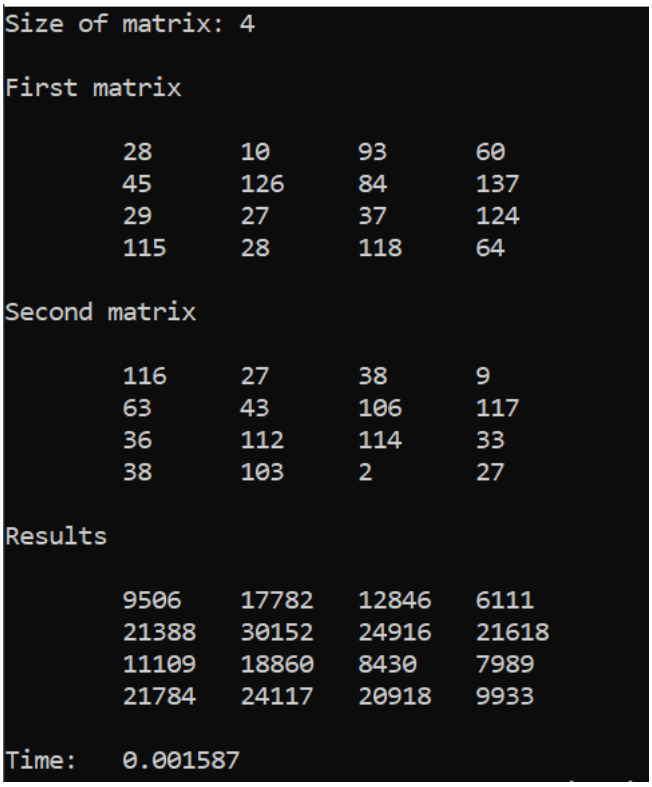

Рис. 1 - Результаты работы последовательного алгоритма с размером матрицы 4.

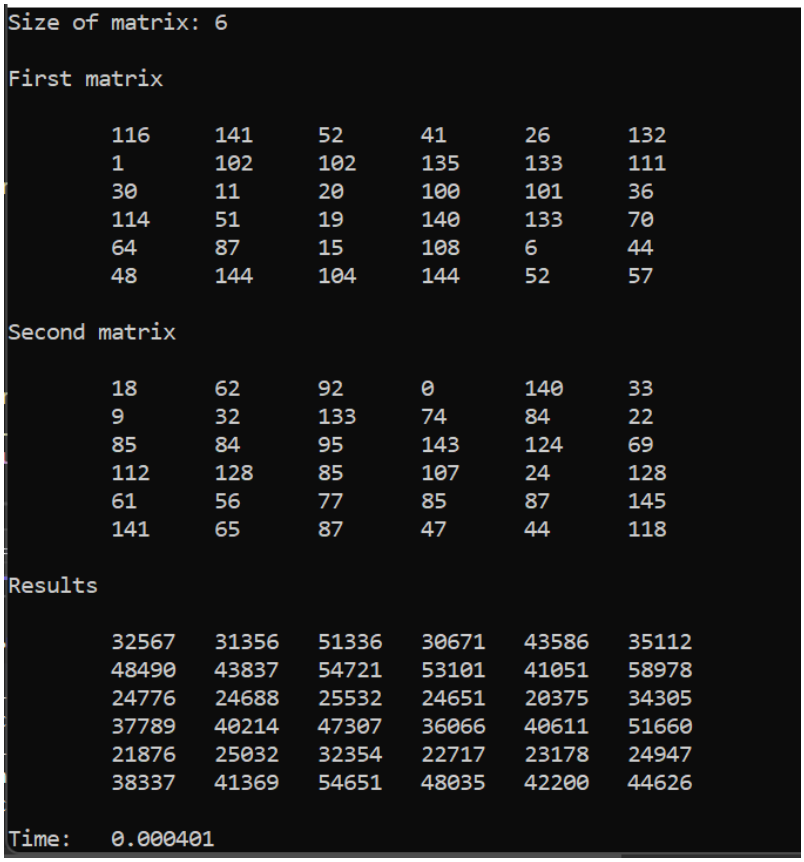

Рис. 2 - Результаты работы последовательного алгоритма с размером матрицы 6.

2. Параллельный алгоритм - Ленточный алгоритм 1 (горизонтальные полосы).

Реализация алгоритма основана на разделении исходной матрицы по блокам-линиям таким образом, что строки обеих матриц, в зависимости от количества процессов, извлекаются процессами для дальнейших преобразований – в цикле происходит перемножение элементов исходных матриц с одинаковым индексом, и полученное значение прибавляется к элементу строки-результата. После чего производится циклическая пересылка строк второй матрицы в процессы-соседей. Таким образом в каждом блоке расчётов получается результирующая строка – для итоговой матрицы. То есть по окончанию цикла каждый процесс имеет результат – строку умножения матриц. Все получившиеся строки склеиваются в нулевом процессе.

Написаны три вспомогательные функции: печати матрицы на экран *matrixprint(),* извлечения линии из матриц *lineOf*() и «сбор» линий в матрицу - *lineTo().*

В нулевом процессе создаются и инициализируются случайными числами две матрицы, которые нужно перемножить. Для результата также создаётся и инициализируется нулями итоговая матрица. С учётом количества задействованных процессов вычисляется длина линии – для разделения элементов матрицы между ними. Вызывается вышеупомянутый метод *lineOf*(), с помощью которого такие линии извлекаются из обеих матриц и записываются в переменные *first_line* и *second_line*. Затем на процессах определяется «декартова топология» - с помощью функции *MPI_Cart_create()*. Это помогает разделить процессы на подгруппы для последующего (кольцевого) обмена. После этого воспользуемся коллективными операциями *MPI_Bcast()* и *MPI_Scatter()* для передачи выделенных линий (их размеров, размеров матриц) следующим процессам данной подгруппы.

Запускается главный цикл, внутри которого и производится умножение, в результате которого получаем часть результатов исковой матрицы. Значения суммируются с результатами из других блоков вычислений. Функция

MPI_Sendrecv_replace() производит обмен «линиями» - внутри созданной выше декартовой топологии (со сдвигом *MPI_Cart_shift()*). После цикла, результирующие строки передаются нулевому процессу (функция *MPI_Gather())* и там данные преобразуются в итоговую матрицу (функция *lineTo*).

Результаты работы программы на 1..n процессах. (Параллельный алгоритм.)

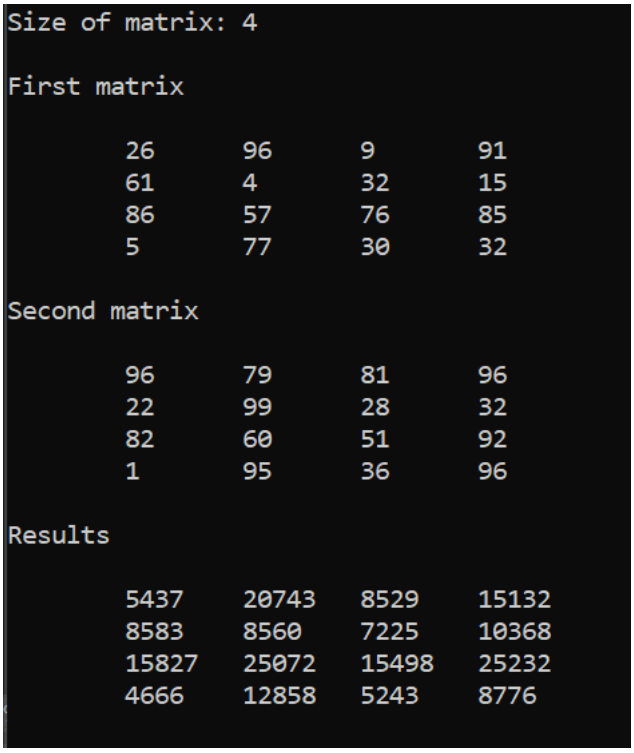

Рис. 3 - Результаты работы программы на 2х процессорах с размером матрицы 4.

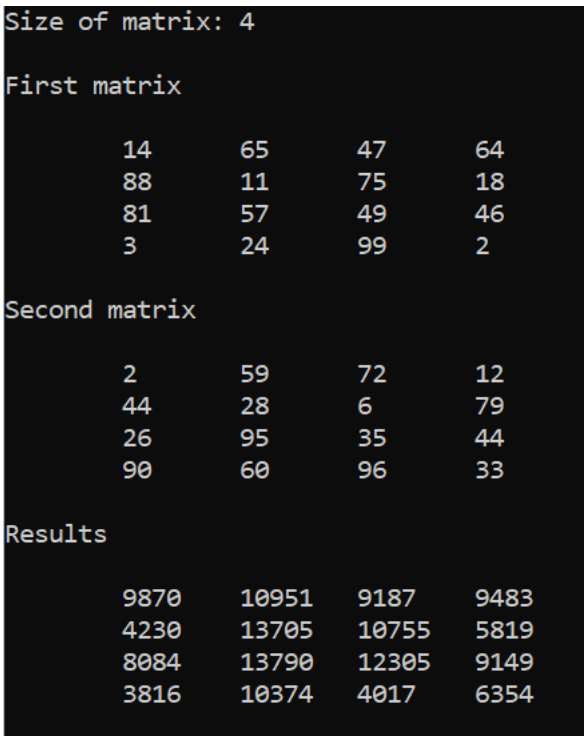

Рис. 4 - Результаты работы программы на 5 процессорах с размером матрицы 4.

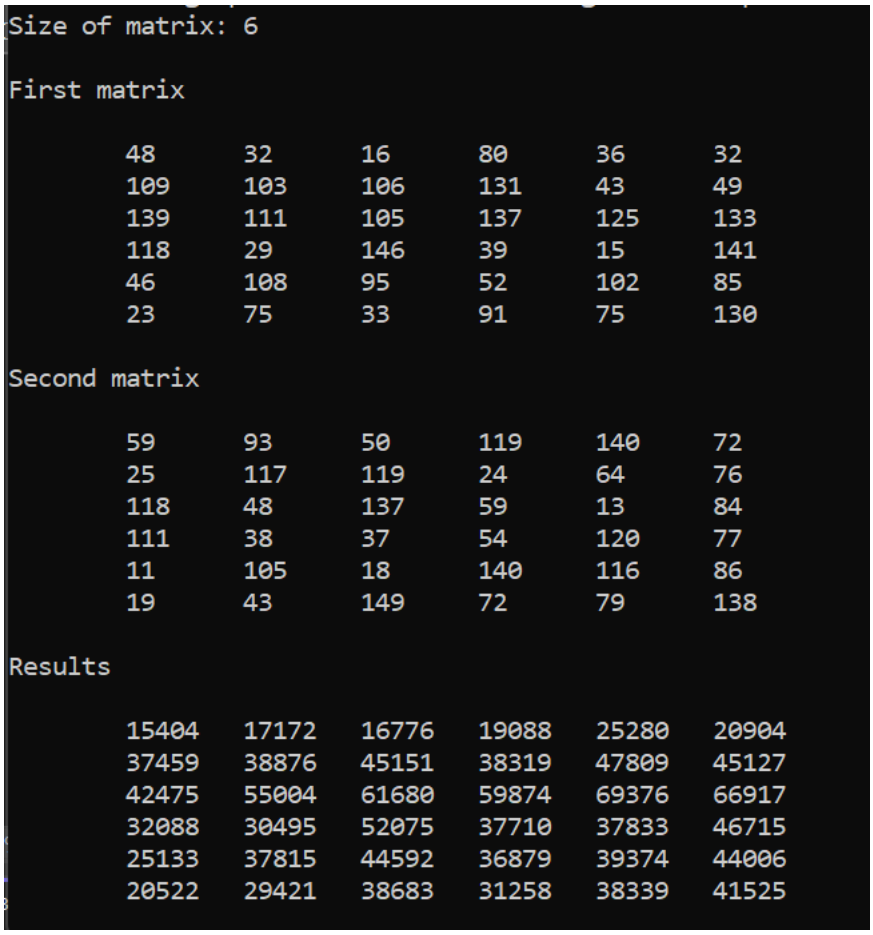

Рис. 5 - Результаты работы программы на 2 процессорах с размером матрицы 6.

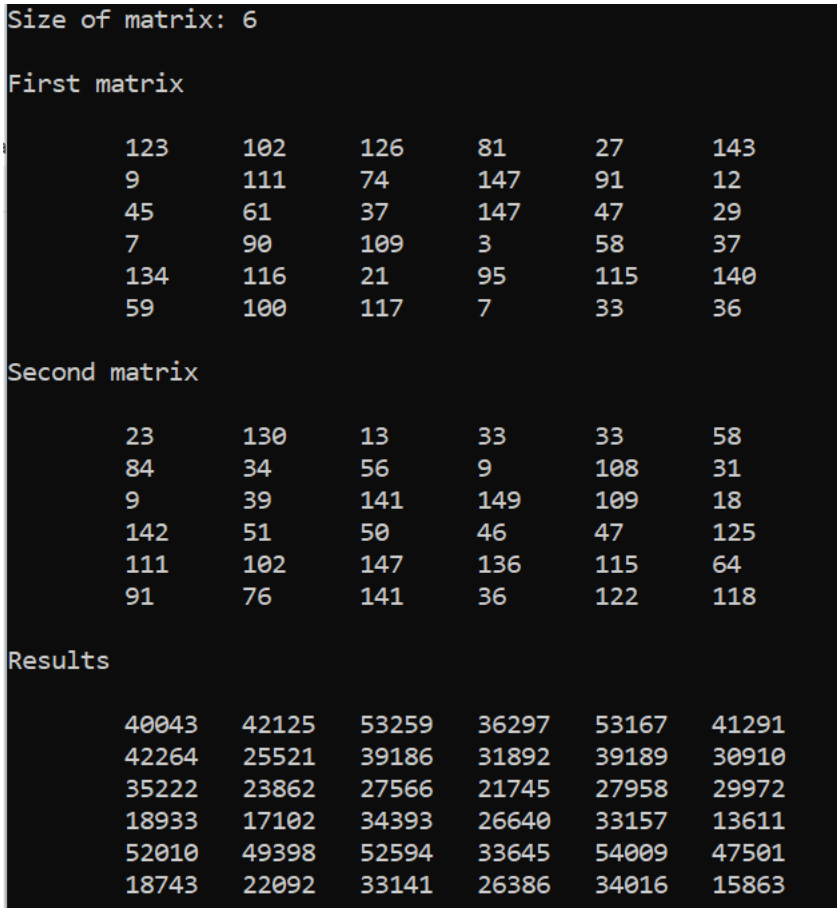

Рис. 6 - Результаты работы программы на 5 процессорах с размером матрицы 6.

Анализ эффективности выбранного алгоритма.

Теоретическая оценка.

Реализация последовательного алгоритма полагает n^3 операций. Следовательно, время выполнения программы напрямую зависит только от параметров размера перемножаемых матриц.

Реализация параллельного алгоритма предполагает перемножения элементов линий матриц на каждом шаге. Так как длина линии рассчитывается как n/r (r – количество процессов), то сложность алгоритма будет равна n^3/r.

Ускорение расчёта относительно параллельного и последовательному алгоритму равно $a = n^3/(n^3/r) = 1/r$.

Результаты вычислительных экспериментов.

Для получения экспериментальных данных модифицируем программу так, чтобы выводились также нужные данные: время работы программы. Для этого добавим в программу следующие строчки:

```
time_start = MPI_Wtime();
...
printf("\nTime:\t%f\n", MPI_Wtime() - time_start);
```

Таблица 1. Результаты вычислительных экспериментов

Размерность матриц (m)	Последовательный алгоритм	2 процесса		4 процесса		8 процессов	
	время	время	ускорение	время	ускорение	время	ускорение
10	0,000297	0,0003	0,99	0,001357	4,569023569	0,006371	21,45117845
50	0,066134	0,002471	26,76406313	0,002036	0,03078598	0,0102	0,154232316
100	0,243578	0,01032	23,60251938	0,016509	0,067777057	0,013475	0,055321088
200	1,016475	0,07689	13,21985954	0,43741	0,43032047	0,059823	0,05885339
500	3,246078	1,12996	2,872737088	1,000934	0,308351802	1,17623	0,362354201
1000	7,435669	3,25586	2,283780322	5,34747	0,71916461	7,21859	0,97080572
10000	∞	∞	-	∞	-	∞	-

	16 процессов		32 процесса		64 процесса	
	время	ускорение	время	ускорение	время	ускорение
	0,001936	0,153409091	0,1021223	0,002908278	0,459291	0,000646649
	0,015362	4,305038406	0,070876	0,933094418	0,161307	0,409988407
	0,761332	0,319936637	0,08597	2,833290683	0,339182	0,718133627
	1,225674	0,829319215	0,5314088	1,912792938	0,796195	1,276665892
	2,996532	1,08327827	3,1085531	1,044240808	4,774276	0,679910001
	8,046376	0,924101608	8,5324478	0,871457895	10,09075	0,736879534
	∞	-	∞	-	∞	-

В качестве ускорения были взяты средние значения ускорения по столбцам.

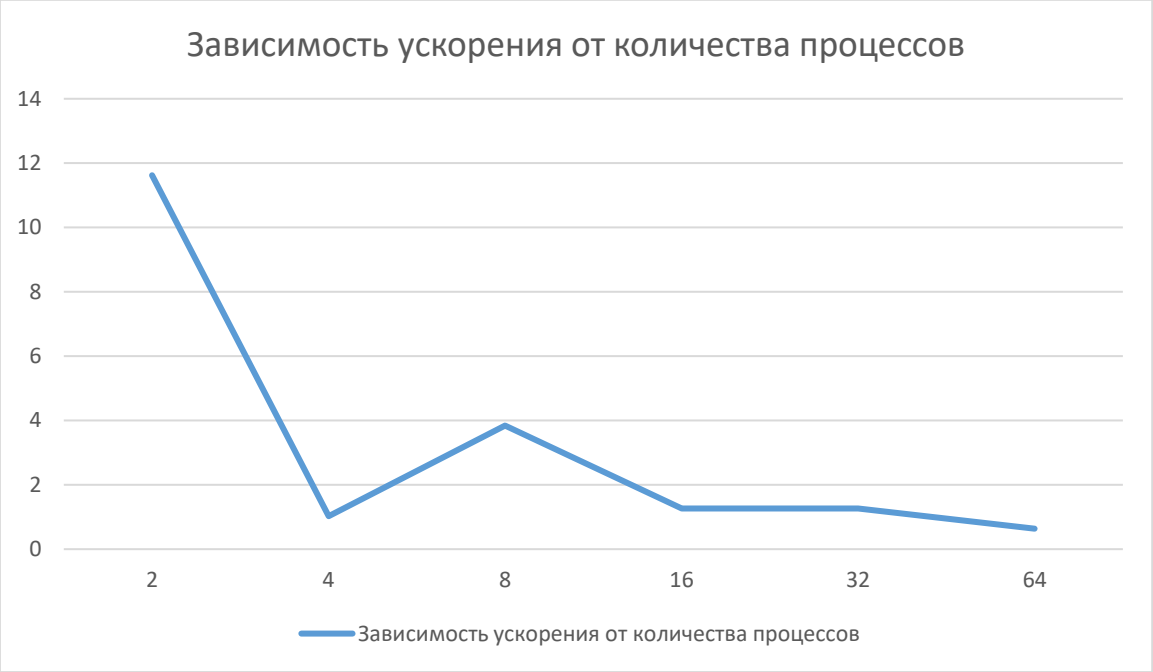

Рис. 7 — Зависимость ускорения от количества процессов

Исходя из графика, можно сделать вывод, что слишком большое число процессов приводит к падению эффективности параллельного алгоритма относительно последовательного.

Построим теоретическую оценку, исходя из полученной формулы и снизим результат на порядок для правильного сопоставления данных (экспериментальные данные – в секундах).

Получим: = n^3*p/r, где p – понижение порядка.

Сравнение экспериментальных и теоретических оценок времени выполнения задачи на 2-х процессах.

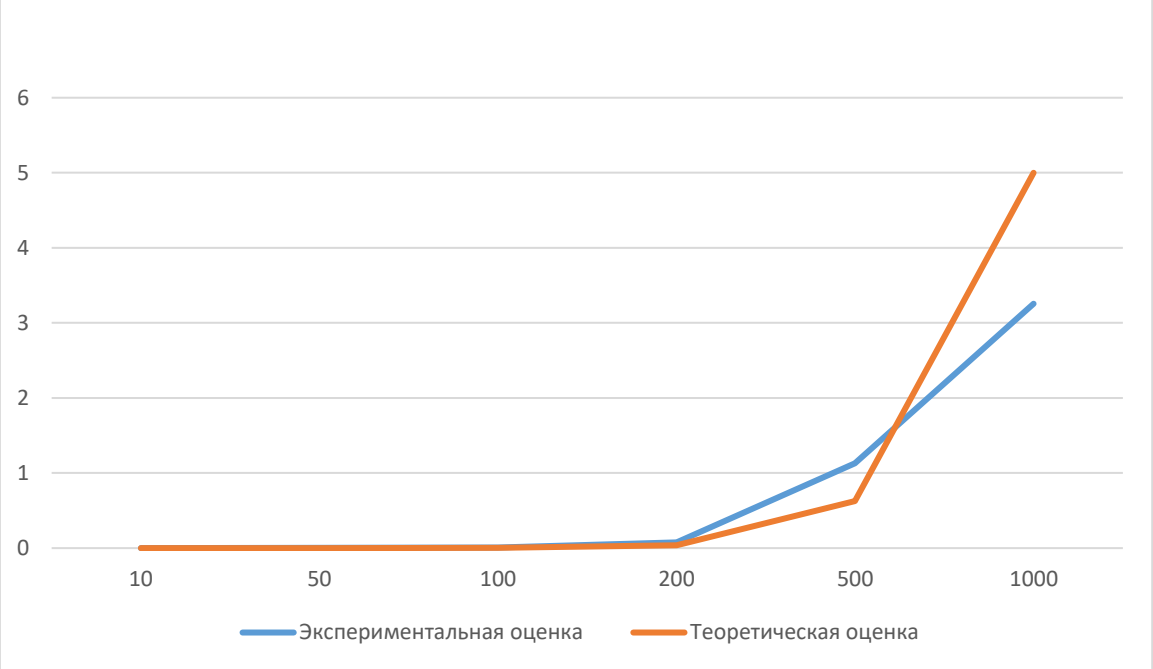

Рис. 8 — Сравнение экспериментальных и теоретических оценок.

По графику видно, что экспериментальные и теоретические данные очень близки, но на значении размерности матрицы = 1000 теоретически ожидалось показание лучше полученного.

Сети Петри:

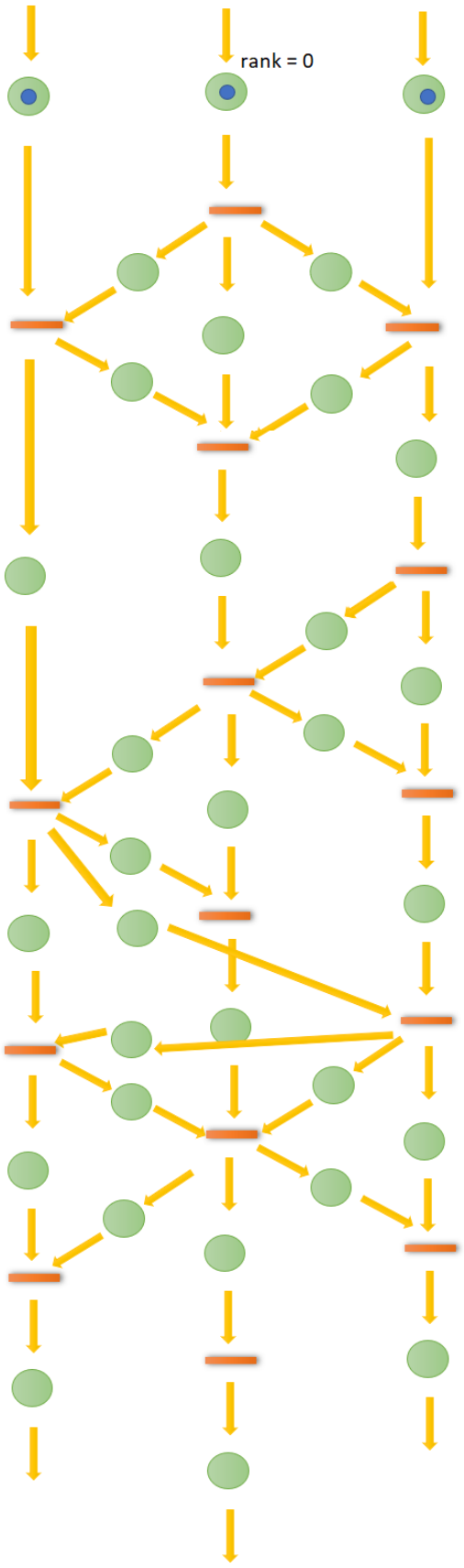

Рис. 7 — Сети Петри.

Выводы.

В ходе лабораторной работы написаны программы, реализующие последовательный и параллельный алгоритмы перемножения матриц. Проведён анализ эффективности на основе сравнения теоретических и экспериментальных оценках. В результате анализа можно заключить, что при работе с матрицами малых размерностей лучше использовать последовательный алгоритм, а с большими матрицами параллельный алгоритм справляется намного быстрее, при условии, что задействовано не очень много процессов (в противном случае эффективность теряет свои показатели).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Последовательный алгоритм:

```
#include <stdio.h>
#include <mpi.h>
#include <cstdlib>
#include <ctime>
#define MAX_LENGTH 1000
#define SIZE 6

int main(int argc, char* argv[]) {
	MPI_Init(&argc, &argv);
	srand(time(0));
	printf("Size of matrix: %d\n", SIZE);
	int first_matrix[SIZE][SIZE];
	int second_matrix[SIZE][SIZE];
	int result[SIZE][SIZE];;

	printf("\nFirst matrix\n\n");
	for (int i = 0; i < SIZE; i++) {
		for (int j = 0; j < SIZE; j++) {
			first_matrix[i][j] = rand() % 150;
			printf("\t%d", first_matrix[i][j]);
		}
		printf("\n");
	}
	printf("\nSecond matrix\n\n");
	for (int i = 0; i < SIZE; i++) {
		for (int j = 0; j < SIZE; j++) {
			second_matrix[i][j] = rand() % 150;
			printf("\t%d", second_matrix[i][j]);
			result[i][j] = 0;
		}
		printf("\n");
	}
	double time = MPI_Wtime();
	for (int i = 0; i < SIZE; i++) {
		for (int j = 0; j < SIZE; j++) {
			for (int k = 0; k < SIZE; k++) {
				result[i][j] += second_matrix[k][j] * first_matrix[i][k];
			}
		}
	}
	printf("\nResults\n\n");
```

```
	for (int i = 0; i < SIZE; i++) {
		for (int j = 0; j < SIZE; j++) {
			printf("\t%d", result[i][j]);
		}
		printf("\n");
	}
	printf("\nTime:\t%f", MPI_Wtime()- time);
	MPI_Finalize();
	return 0;
}
```

Параллельный алгоритм:

```
#include <stdio.h>
#include <mpi.h>
#include <cstdlib>
#include <ctime>
#define MAX_LENGTH 1000
#define SIZE 4

void matrixprint(int** matr, int size) {
	for (int x = 0; x < size; x++) {
		for (int y = 0; y < size; y++) {
			printf("\t%d", matr[x][y]);
		}
		printf("\n");
	}
}

void lineOf(int** matrix, int h, int w, int* line) {
	int count = 0;
	for (int x = 0; x < h; x++) {
		for (int y = 0; y < w; y++) {
			line[count++] = matrix[x][y];
		}
	}
}

void lineTo(int** matrix, int size, int* line) {
	for (int x = 0; x < size; x++) {
		for (int y = 0; y < size; y++) {
			matrix[x][y] = line[x * size + y];
		}
	}
}
```

```
int main(int argc, char* argv[]) {
	int number, rank, source, rank_dest;
	MPI_Status status;
	MPI_Comm new_comm;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &number);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	srand(time(0));
	double time = MPI_Wtime();
	int** first_matrix, ** second_matrix, ** result{}, * first_line{}, * second_line{},
		* result_line{}, * first_buffer, * second_buffer, * tmp, paragraph, length;
	if (rank == 0) {
		printf("Size of matrix %d\n", SIZE);
		paragraph = (number- SIZE % number) % number;
		first_matrix = (int**)calloc(paragraph + SIZE, sizeof(int*));
		second_matrix = (int**)calloc(paragraph + SIZE, sizeof(int*));
		result = (int**)calloc(SIZE, sizeof(int*));
		int new_size = paragraph + SIZE;
		first_line = new int[new_size];
		second_line = new int[new_size];
		result_line = new int[new_size * SIZE];
		for (int i = 0; i < new_size; i++) {
			first_matrix[i] = new int[SIZE];
			second_matrix[i] = new int[SIZE];
			result[i] = new int[SIZE];
		}
		length = round(new_size / number);
		for (int x = 0; x < SIZE; x++)
			for (int y = 0; y < SIZE; y++) {
				first_matrix[x][y] = rand() % 150;
				second_matrix[x][y] = rand() % 150;
				result[x][y] = 0;
			}
		printf("\n\nFirst matrix\n");
		matrixprint(first_matrix, SIZE);
		printf("\n\nSecond matrix\n");
		matrixprint(second_matrix, SIZE);
		lineOf(first_matrix, new_size, SIZE, first_line);
		lineOf(second_matrix, new_size, SIZE, second_line);
	}
	first_buffer = new int[length * SIZE];
	second_buffer = new int[length * SIZE];
	tmp = new int[length * SIZE];
	int dims[1] = { number };
	int periods[] = { 1, 0 };
	MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, 1, &new_comm);
	MPI_Comm_rank(new_comm, &rank);
```

```
	int int_size = SIZE;
	MPI_Bcast(&int_size, 1, MPI_INT, 0, new_comm);
	MPI_Bcast(&length, 1, MPI_INT, 0, new_comm);
	MPI_Scatter(first_line, length * SIZE, MPI_INT, first_buffer, length * SIZE, MPI_INT, 0,
new_comm);
	MPI_Scatter(second_line, length * SIZE, MPI_INT, second_buffer, length * SIZE, MPI_INT, 0,
new_comm);
	for (int procNumber = 0; procNumber < number; procNumber++) {
		MPI_Cart_shift(new_comm, 0, 1, &source, &rank_dest);
		int p = ((rank + number- procNumber) % number) * length;
		for (int x = 0; x < length; x++) { for (int y = 0; y < SIZE; y++)
				for (int z = p; z < p + length; z++)
					tmp[y + x * SIZE] += first_buffer[z + SIZE * x] * second_buffer[y
+ SIZE * (z- p)];

		}
		MPI_Sendrecv_replace(second_buffer, length * SIZE, MPI_INT, rank_dest, 0, source,
0, new_comm, &status);
	}
	MPI_Gather(tmp, length * SIZE, MPI_INT, result_line, length * SIZE, MPI_INT, 0, new_comm);
	if (rank == 0) {
		lineTo(result, SIZE, result_line);
		printf("\n\nResults\n");
		matrixprint(result, SIZE);
		printf("\n\nTime\t%f", MPI_Wtime()- time);
	}
	MPI_Finalize();
	return 0;
}
```