МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики

СУПЕРКОМПЬЮТЕРЫ
И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Задание
«Реализация параллельных программ
с использованием технологии OpenMP и MPI»

ОТЧЕТ
о выполненном задании

студент 328 группы
Морозов Илья Федорович

Москва 2023 г.

OpenMP

I. Директива «for»

Для реализации параллельной версии программы с использованием стандарта OpenMP и директивы «for» было добавлено следующее:

(1) Исходная функция *kernel_3mm* была заменена функцией *kernel_1mm*, которое реализует умножение двух матриц с использованием блочного алгоритма:

```
static void
kernel_1mm(
        int ni, int nj, int nk,
        double A[ni][nk],
        double B[nk][nj],
        double out[ni][nj])
{
    /* Local variable for every thread */
    double res = 0.;
    int i, j, k, ii, jj, kk;

    #pragma omp parallel for collapse(3) default(shared) \
            private(res, i, j, k, ii, jj, kk)

    for (i = 0; i < ni; i += MAX_BLOCK_SIZE) {
        for (j = 0; j < nj; j += MAX_BLOCK_SIZE) {
            for (k = 0; k < nk; k += MAX_BLOCK_SIZE) {

                /* ================ Blocked Matrix Multiplication ================ */
                for (ii = i; ii < MIN(ni, i + MAX_BLOCK_SIZE); ++ii) {
                    for (jj = j; jj < MIN(nj, j + MAX_BLOCK_SIZE); ++jj) {
                        res = 0;
                        for (kk = k; kk < MIN(nk, k + MAX_BLOCK_SIZE); ++kk) {
                            res += A[ii][kk] * B[kk][jj];
                        }
                        #pragma omp atomic
                        out[ii][jj] += res;
                    }
                }
            }
        }
    }
}
```

(2) Добавлена функция *verify*, которая выполняет проверку корректности нового алгоритма, используя стандартный — умножение матриц по правилу «строка-столбец»:

```
static int
verify(
        int ni, int nj, int nk,
        double A[ni][nk],
        double B[nk][nj],
        double res[ni][nj],
        double DBL_EPS)
{
    int i, j, k;

    for (i = 0; i < ni; i++) {
        for (j = 0; j < nj; j++) {
            double tmp = 0;
            for (k = 0; k < nk; ++k) {
                tmp += A[i][k] * B[k][j];
            }
            if (fabs(tmp - res[i][j]) >= DBL_EPS) {
                fprintf(stderr, "[-] Matrix Multiply incorrect.\n");
                return CODE_FAILURE;
            }
        }
    }
    printf("[+] Matrix Multiply correct.\n");
    return CODE_SUCCESS;
}
```

Результаты «for»

Ниже представлены результаты запуска алгоритма в разных условиях:

- Различное количество потоков;
- Различный объём данных во входных матрицах.

Так как алгоритм работает за кубическую сложность — O (N * M * K), временные шкалы были прологарифмированы для большей наглядности.

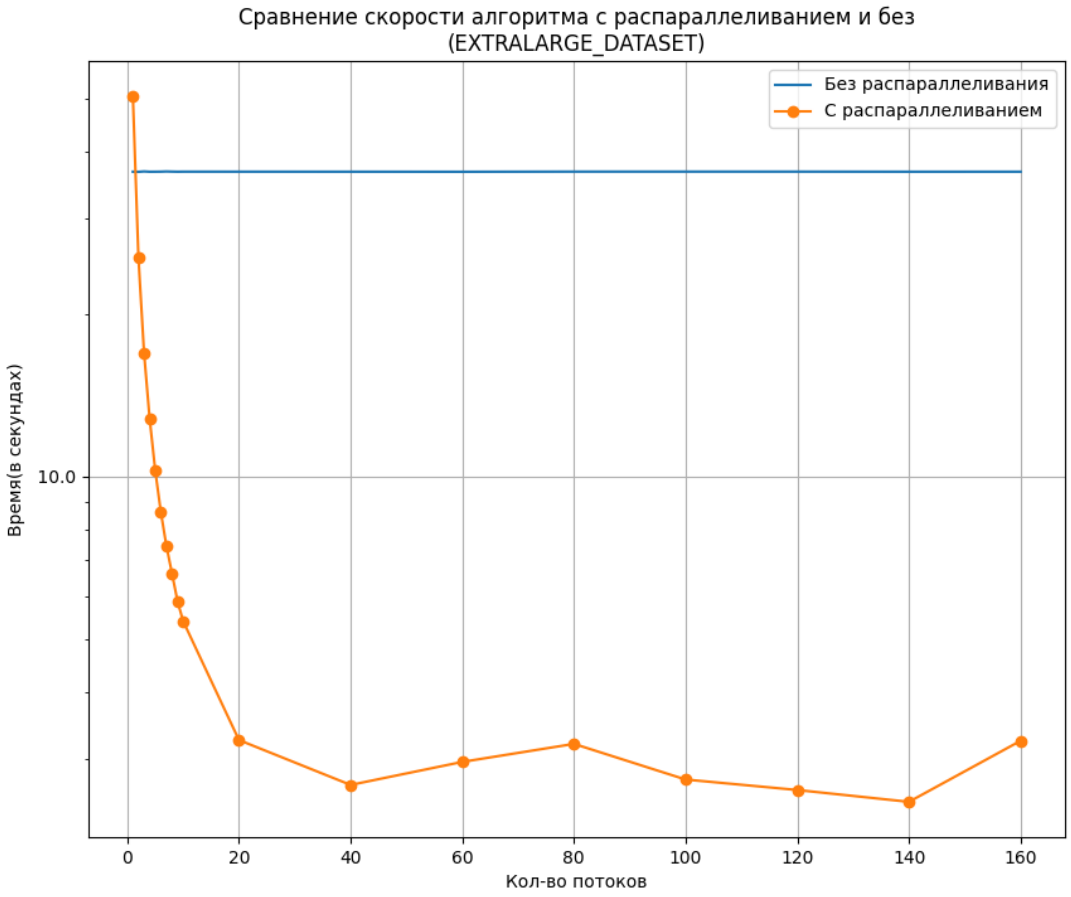

График 1: Сравнение скорости исходного алгоритма и полученного с помощью OpenMP «for» на EXTRALARGE_DATASET.

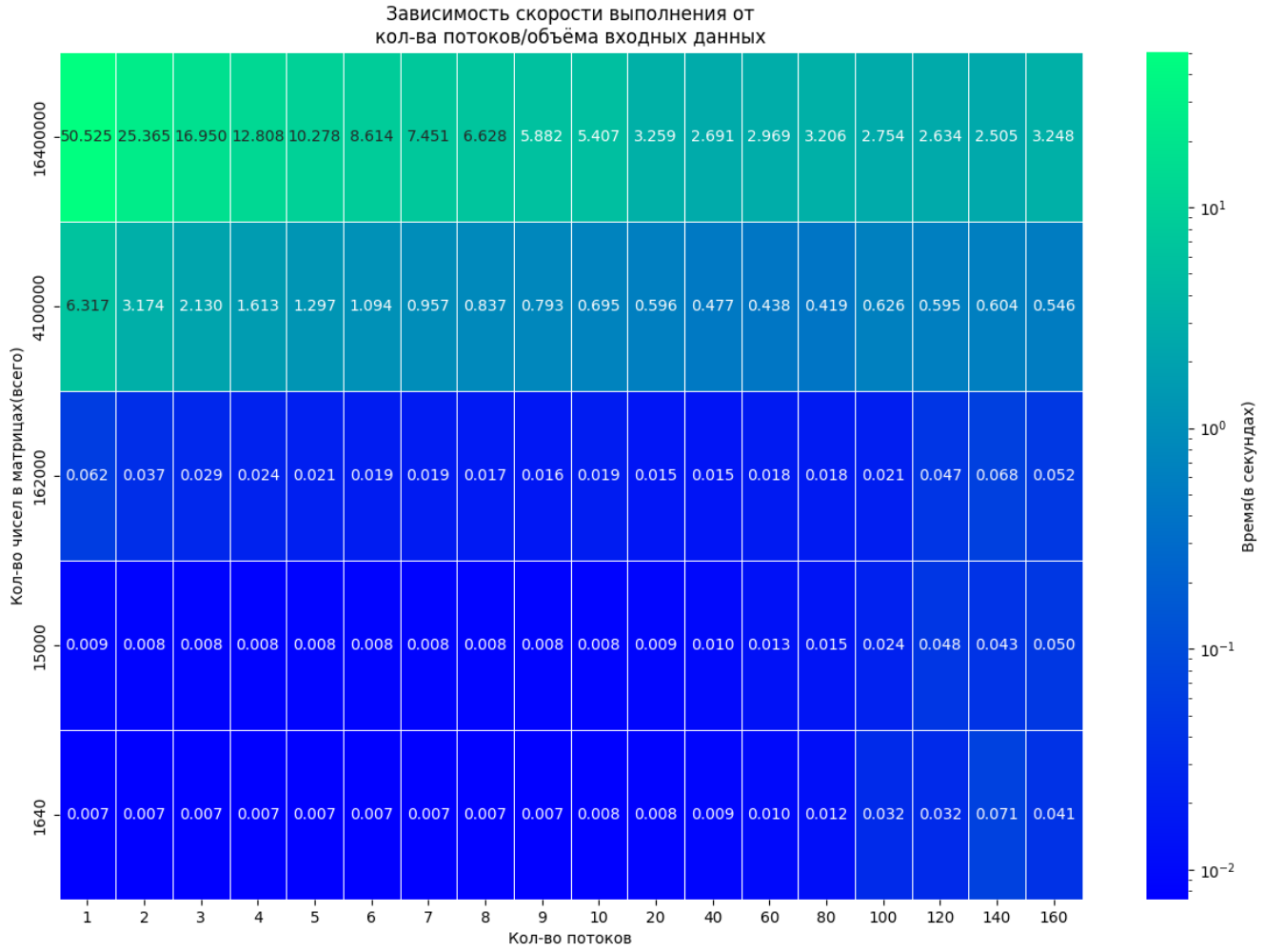

График 2: Зависимость времени выполнения алгоритма от количества потоков/объёма входных данных (суммарного количества чисел в перемножаемых матрицах).

II. Директива «task»

По аналогии с предыдущей директивой для реализации параллельной версии программы с использованием директивы «task» было добавлено следующее:

(1) Блочный алгоритм, расставлены необходимые «pragma»:

```
static void
kernel_1mm(
        int ni, int nj, int nk,
        double A[ni][nk],
        double B[nk][nj],
        double out[ni][nj])
{
    double res = 0.;
    /* Local variable for every thread */
    #pragma omp parallel firstprivate(res)
    /* Thread that divides into tasks */
    #pragma omp single //
    {
        int i, j, k;
        for (i = 0; i < ni; i += MAX_BLOCK_SIZE) {
            for (j = 0; j < nj; j += MAX_BLOCK_SIZE) {
                for (k = 0; k < nk; k += MAX_BLOCK_SIZE) {

                    /* ============== Blocked Matrix Multiplication ============== */
                    #pragma omp task
                    {
                        int ii, jj, kk;
                        for (ii = i; ii < MIN(ni, i + MAX_BLOCK_SIZE); ++ii) {
                            for (jj = j; jj < MIN(nj, j + MAX_BLOCK_SIZE); ++jj) {
                                res = 0;
                                for (kk = k; kk < MIN(nk, k + MAX_BLOCK_SIZE); ++kk) {
                                    res += A[ii][kk] * B[kk][jj];
                                }
                                #pragma omp atomic
                                out[ii][jj] += res;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

(2) Функция *verify*, которая выполняет проверку корректности нового алгоритма, используя стандартный — умножение матриц по правилу «строка-столбец»:

```
static int
verify(
        int ni, int nj, int nk,
        double A[ni][nk],
        double B[nk][nj],
        double res[ni][nj],
        double DBL_EPS)
{
    int i, j, k;

    for (i = 0; i < ni; i++) {
        for (j = 0; j < nj; j++) {
            double tmp = 0;
            for (k = 0; k < nk; ++k) {
                tmp += A[i][k] * B[k][j];
            }
            if (fabs(tmp - res[i][j]) >= DBL_EPS) {
                fprintf(stderr, "[-] Matrix Multiply incorrect.\n");
                return CODE_FAILURE;
            }
        }
    }
    printf("[+] Matrix Multiply correct.\n");
    return CODE_SUCCESS;
}
```

Результаты «task»

Ниже представлены результаты запуска алгоритма в разных условиях:

- Различное количество потоков;
- Различный объём данных во входных матрицах.

Временные шкалы также были прологарифмированы для большей наглядности.

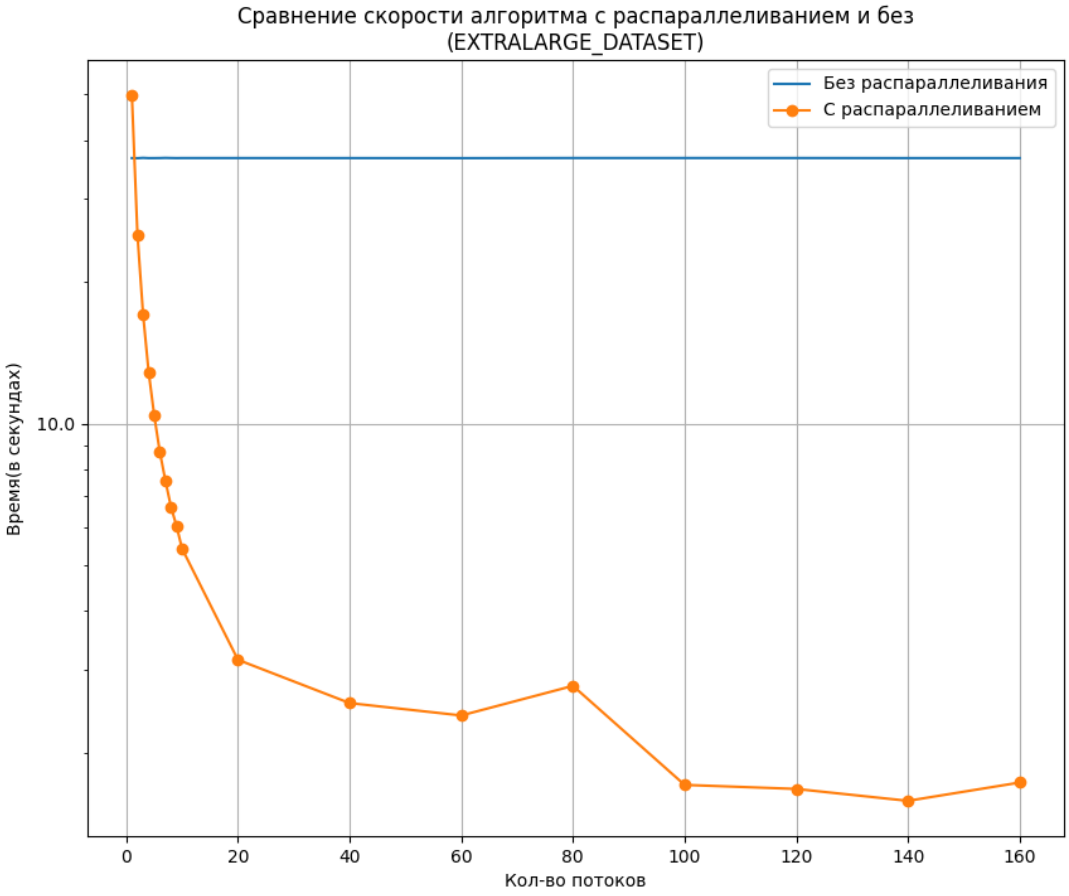

График 3: Сравнение скорости исходного алгоритма и полученного с помощью OpenMP «task» на EXTRALARGE_DATASET.

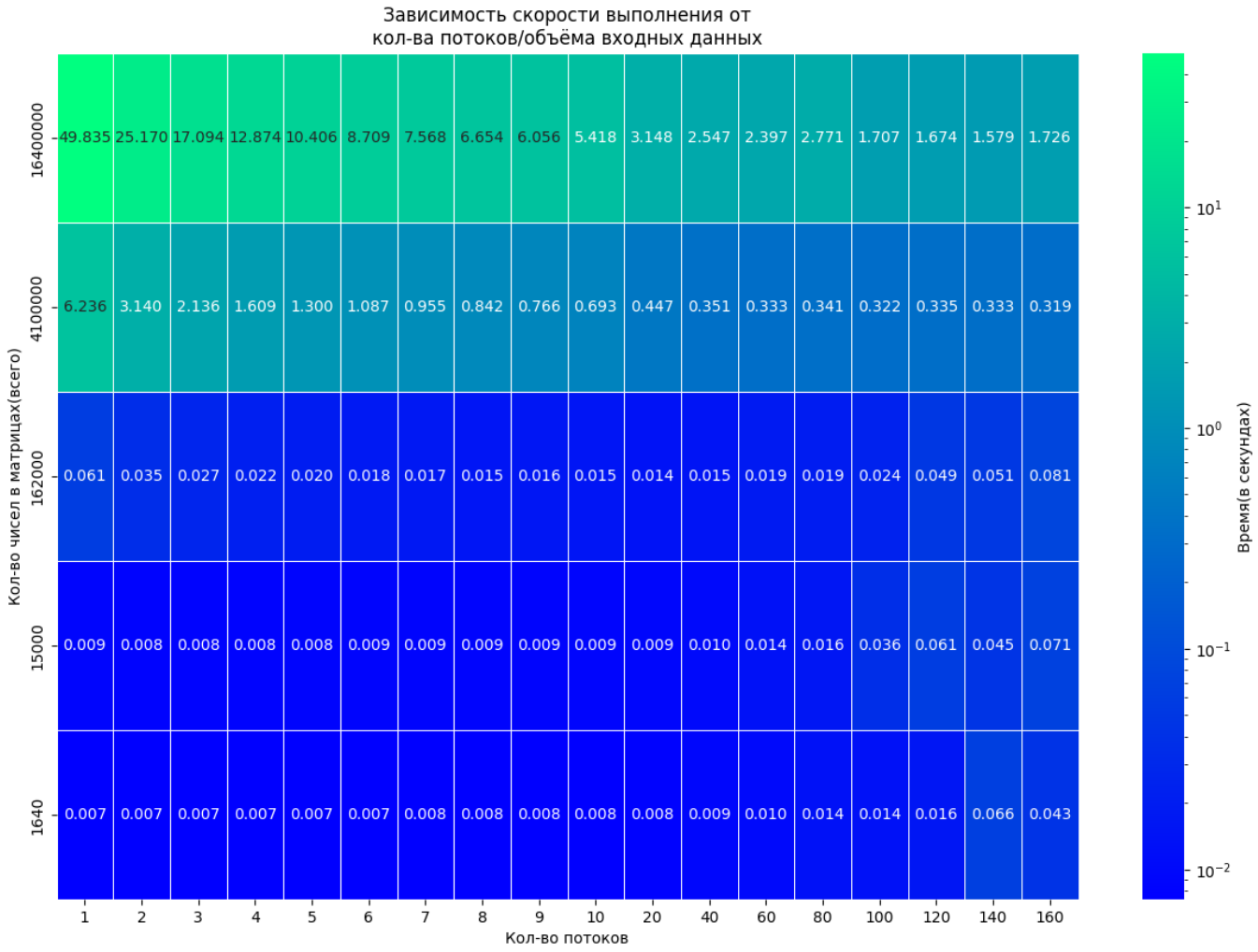

График 4: Зависимость времени выполнения алгоритма от количества потоков/объёма входных данных (суммарного количества чисел в перемножаемых матрицах).

MPI

Код реализованного **MPI** алгоритма приложен к данному отчёту. Из важных моментов стоит отметить следующее:

(1) **Scattering** строк первой матрицы, участвующей в умножении, а также, **Broadcasting** второй матрицы всем процессам:

```
int NUM_ROWS = ni / p;  /* Number of rows in Local Array */

/* ============ Scatter rows of first matrix to different processes ============ */

int disps[p], counts[p];
for (i = 0; i < p; ++i) {
    counts[i] = (i == p - 1) ? (NUM_ROWS + ni % p) * nk : NUM_ROWS * nk;
    disps[i] = i * nk * NUM_ROWS;
}

if (rank == p - 1) NUM_ROWS += ni % p;

double *rowsA = malloc(NUM_ROWS * nk * sizeof(double));
double *rowsOut = calloc(NUM_ROWS * nj, sizeof(double));

MPI_Scatterv(&(A[0][0]), counts, disps, MPI_DOUBLE,
             &(rowsA[0]), NUM_ROWS * nk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

/* ================== Broadcast Second Matrix to all processes ================== */

MPI_Bcast(B, nk * nj, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

(2) Умножение процессами выделенных им областей:

```
/* =============== Perform Vector Multiplication by all processes =============== */

for (i = 0; i < NUM_ROWS; ++i) {
    for (j = 0; j < nj; ++j) {
        for (k = 0; k < nk; ++k) {
            rowsOut[i * nj + j] += rowsA[i * nk + k] * B[k][j];
        }
    }
}
MPI_Barrier(MPI_COMM_WORLD);
```

(3) Gathering результата в общую матрицу:

```
 /* =========== Gather rows of result matrix from different processes =========== */

 for (i = 0; i < p; ++i) {
     counts[i] = (i == p - 1) ? (NUM_ROWS + ni % p) * nj : NUM_ROWS * nj;
     disps[i] = i * nj * NUM_ROWS;
 }

 MPI_Gatherv(&(rowsOut[0]), NUM_ROWS * nj, MPI_DOUBLE,
             &(out[0][0]), counts, disps, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Результаты MPI

Ниже представлены результаты запуска алгоритма в разных условиях:

- Различное количество потоков;
- Различный объём данных во входных матрицах.

Временные шкалы были прологарифмированы для большей наглядности.

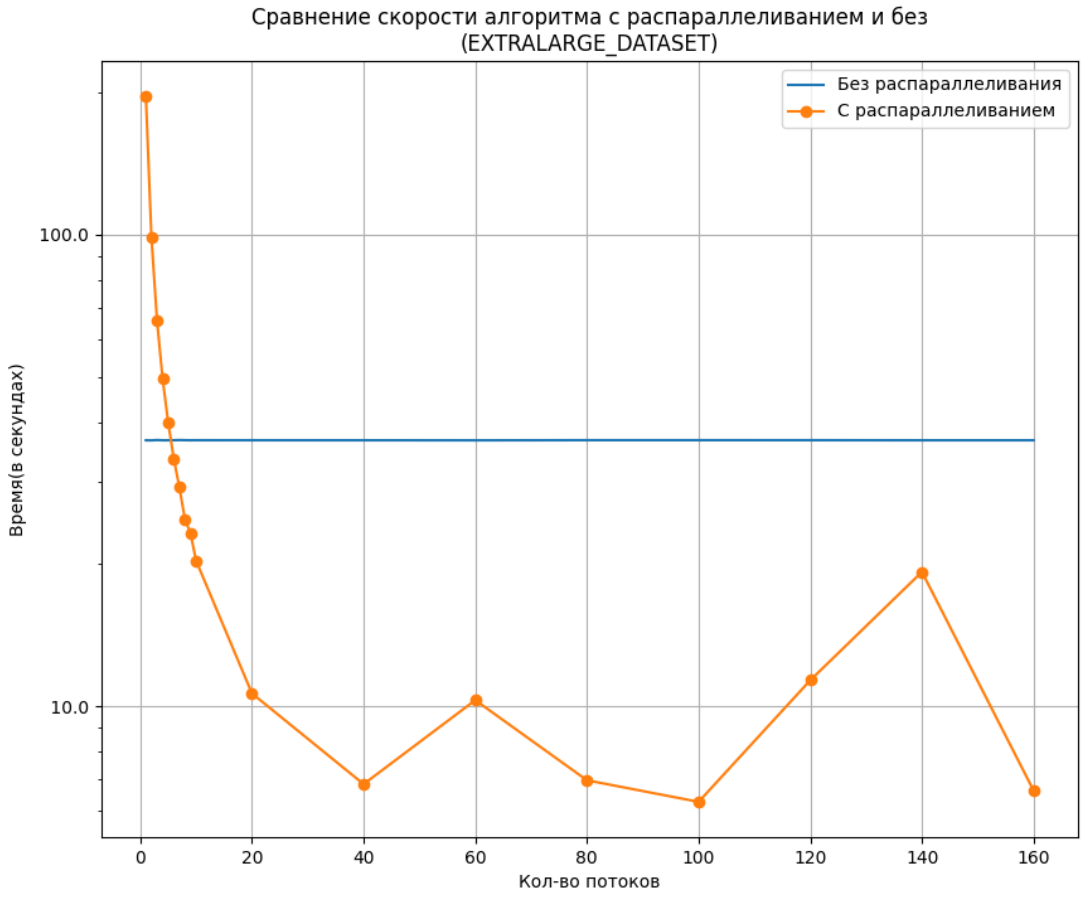

График 5: Сравнение скорости исходного алгоритма и полученного с помощью MPI на EXTRALARGE_DATASET.

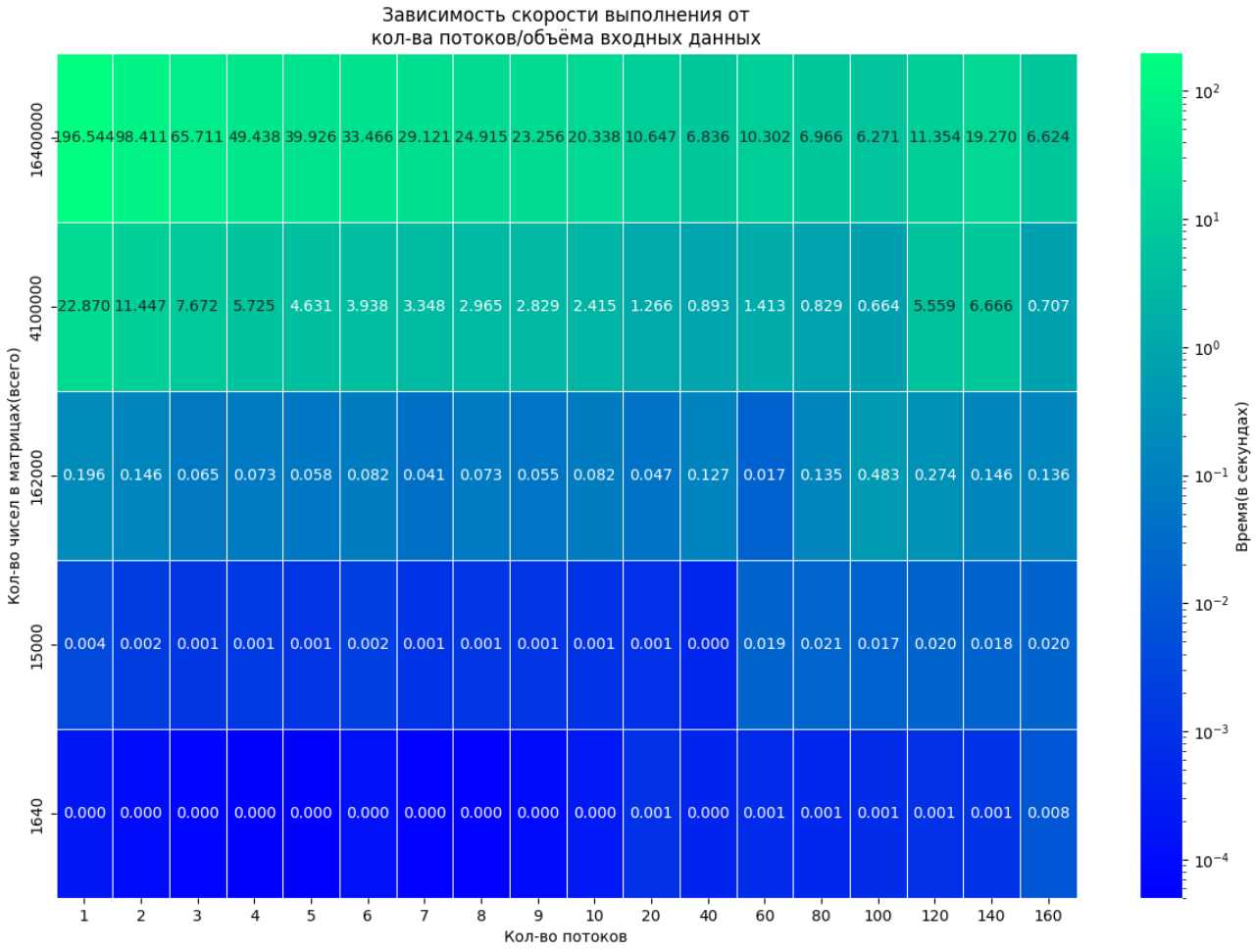

График 6: Зависимость времени выполнения алгоритма от количества потоков/объёма входных данных (суммарного количества чисел в перемножаемых матрицах).

Выводы

Заметим, что и директива «for», и директива «task» даёт ощутимый прирост в производительности в задаче умножения матриц.
Блочный алгоритм является хорошей основой для дальнейшего распараллеливания, но нужен правильный размера блока, который стоит выбирать по следующей формуле:

$$BlockSize = \sqrt{\left(\frac{M}{3}\right)}, \text{где M — объём быстрой памяти (в нашем случае L1 — кэш).}$$

В случае **MPI** также получаем хороший выигрыш в скорости. Заметим, что в данном случае приведён ленточный алгоритм, использующий правило «строка-столбец».
При изменении числа потоков необходимо корректно избирать размеры «лент» и верно обрабатывать случай неделимости числа строк матрицы на число процессов.
В коде использована данная формула:

$$NumRows = \left[\frac{N}{p}\right], \text{где } N - \text{число строк в матрице, а } p - \text{число процессов.}$$

Также для каждого из вариантов использования **OpenMP**, а также **MPI** проведена проверка корректности с помощью функции *verify*. Все алгоритмы реализованы правильно.

Из данных графиков наблюдаем следующие явления:

- при увеличении числа потоков производительность растёт (почти во всех случаях);
- распараллеленный алгоритм работает лучше исходного (прирост производительности примерно **в 20 раз**);

Программы

Изменённый код, bash/python скрипты, графики и результаты тестирования приложены к данному отчёту.