Отчёт: «Распараллеливание метода Рунге-Кутты 4 порядка»

Радько В. А.

Решение дифференциального уравнения методом Рунге-Кутты 4 порядка

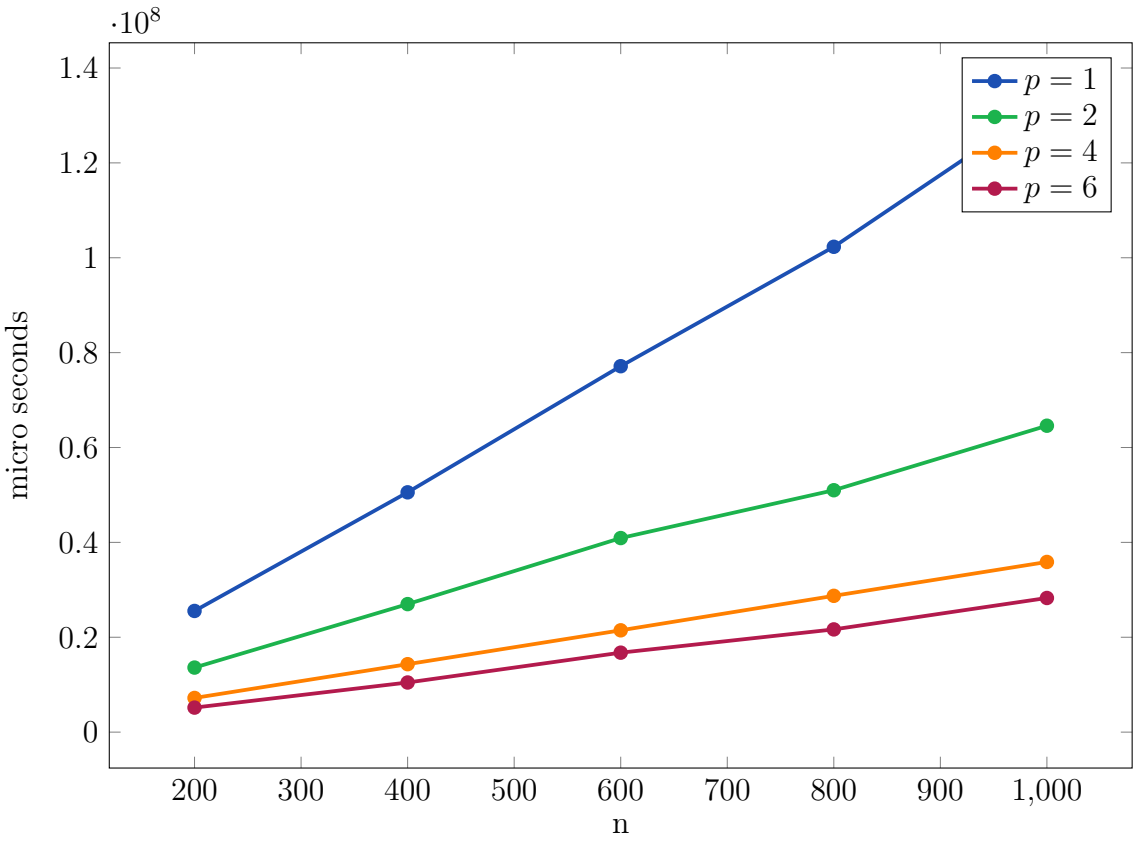

Рис. 1: Время работы

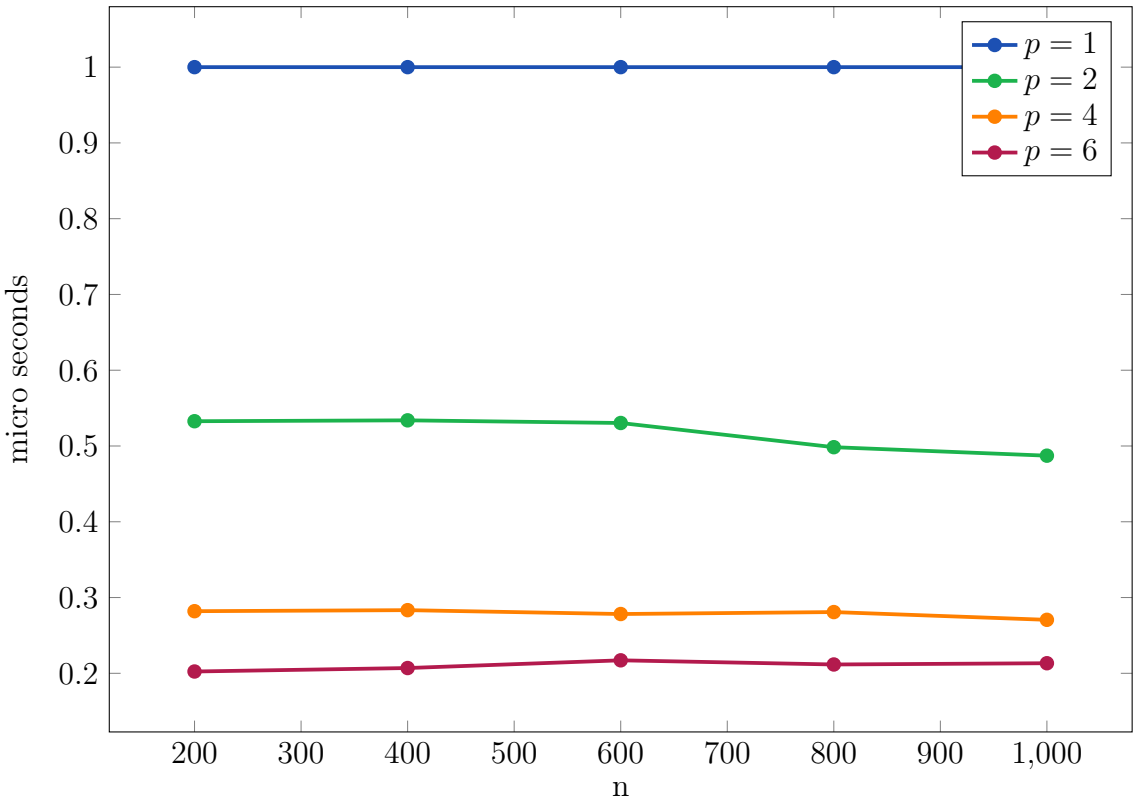

Рис. 2: Ускорение

Листинг 1: Исходный код программы для MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <mpi.h>
#include <sys/time.h>
#include <unistd.h>

const double pi = 3.14159265359;

long long get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);

    return tv.tv_usec + tv.tv_sec*1000000LL;
 }
```

```
long seconds(long long t) {
    return (t / 1000000);
}

long useconds(long long t) {
    return (t % 1000000);
}

double uniform_distr(double a, double b) {
    return rand()*(b - a)/RAND_MAX + a;
}

size_t array_size(int rank, size_t n) {
    int comm_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    return (rank+1)*n/comm_size - rank*n/comm_size;
}

void output(FILE* f, double* z, size_t n) {
    int comm_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    int rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    size_t local_amount_points = array_size(rank, n);
    double* x = z;
    double* y = z + local_amount_points;

    if (rank == 0) {
        for (int i = 0; i < local_amount_points; ++i) {
            fprintf(f, "%f %f\n", x[i], y[i]);
        }

        for (int node = 1; node < comm_size; ++node) {
            size_t node_size = array_size(node, n);
            double* buffer = calloc(3*node_size, sizeof(double));
```

```
            x = buffer;
            y = buffer + node_size;

            MPI_Status stat;
            MPI_Recv(buffer,     // buff
                    (int) 3*node_size, // count
                    MPI_DOUBLE,  // type
                    node,        // source
                    node,        // tag
                    MPI_COMM_WORLD, // comm
                    &stat);

            for (int i = 0; i < node_size; ++i) {
                fprintf(f, "%f %f\n", x[i], y[i]);
            }

            free(buffer);
        }
    } else {
        MPI_Send(z,          // buff
            (int) 3*local_amount_points, // count
            MPI_DOUBLE, // type
            0,          // dest
            rank,       // tag
            MPI_COMM_WORLD); // comm
    }
}

void F(double* z, double* res, size_t n) {
    int comm_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    int rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    size_t local_amount_points = array_size(rank, n);

    double* x = z;
```

```
double* y = z + local_amount_points;
double* omega = z + 2*local_amount_points;

double* rx = res;
double* ry = res + local_amount_points;
double* romega = res + 2*local_amount_points;

for (int node = 0; node < comm_size; ++node) {
    size_t node_size = array_size(node, n);
    double* buffer = calloc(3*node_size, sizeof(double));

    double* bx = buffer;
    double* by = buffer + node_size;
    double* bomega = buffer + 2*node_size;

    if (rank == node) {
        memcpy(buffer, z, 3*node_size*sizeof(double));
    }

    MPI_Bcast(buffer, //buffer
        (int) 3*node_size, //count of elements
        MPI_DOUBLE, //type of elements
        node,       //source
        MPI_COMM_WORLD); //network

    for (int i = 0; i < local_amount_points; ++i) {
        for (int j = 0; j < node_size; ++j) {
            if (rank == node && i == j) continue;

            double dx = x[i] - bx[j];
            double dy = y[i] - by[j];

            rx[i] += bomega[j]*(y[i] - by[j])/(dx*dx + dy*dy);
            ry[i] += bomega[j]*(x[i] - bx[j])/(dx*dx + dy*dy);
        }
    }

    free(buffer);
```

```
    }

    for (int i = 0; i < local_amount_points; ++i) {
        rx[i] /= -1/(2*pi);
        ry[i] /=  1/(2*pi);
    }
}

void runge_kutta_step(double* z, size_t n, double h) {
    int comm_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    int rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    size_t local_amount_points = array_size(rank, n);

    double* k_1 = calloc(3*local_amount_points, sizeof(double));
    double* k_2 = calloc(3*local_amount_points, sizeof(double));
    double* k_3 = calloc(3*local_amount_points, sizeof(double));
    double* k_4 = calloc(3*local_amount_points, sizeof(double));

    double* arg = calloc(3*local_amount_points, sizeof(double));

    F(z, k_1, n);

    for (int i = 0; i < 2*local_amount_points; ++i)
        arg[i] = z[i] + h*k_1[i]/2;

    F(arg, k_2, n);

    for (int i = 0; i < 2*local_amount_points; ++i)
        arg[i] = z[i] + h*k_2[i]/2;

    F(arg, k_3, n);

    for (int i = 0; i < 2*local_amount_points; ++i)
        arg[i] = z[i] + h*k_3[i];
```

```
    F(arg, k_4, n);

    for (int i = 0; i < 2*local_amount_points; ++i)
        z[i] += (k_1[i] + 2*k_2[i] + 2*k_3[i] + k_4[i])*h/6;

    free(k_1);
    free(k_2);
    free(k_3);
    free(k_4);

    free(arg);
}

int main(int argc, char** argv) {
    if (argc < 3) {
        printf("usage: ./m init_file turns h\n");
        return 1;
    }

    MPI_Init(&argc, &argv);

    int comm_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    int rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    double h = atof(argv[3]);
    int turns = atoi(argv[2]);

    FILE* init = fopen(argv[1], "r");
    if (init == NULL) {
        printf("bad file\n");
        return 1;
    }

    size_t n = 0;
```

```
fscanf(init, "%ld", &n);

size_t local_amount_points = array_size(rank, n);
double* z = calloc(3*local_amount_points, sizeof(double));

double* x = z;
double* y = z + local_amount_points;
double* omega = z + 2*local_amount_points;

printf("%d initial data: n = %ld, h = %3.1lf, lap = %ld\n",
    rank, n, h, local_amount_points);

for (int node = 0; node < comm_size; ++node) {
    size_t node_size = array_size(node, n);

    if (rank == 0) {
        double* buffer = calloc(3*node_size, sizeof(double));
        double* _x = buffer;
        double* _y = buffer + node_size;
        double* _w = buffer + node_size*2;

        for (int p = 0; p < node_size; ++p) {
            if (node == 0)
                fscanf(init, "%lf %lf %lf", x + p, y + p, omega + p);
            else
                fscanf(init, "%lf %lf %lf", _x + p, _y + p, _w + p);
        }

        if (node != 0) {
            MPI_Send(buffer, // buff
                (int) 3*node_size, // count
                MPI_DOUBLE,  // type
                node,        // dest
                node,        // tag
                MPI_COMM_WORLD); // comm
        }

        free(buffer);
```

```
    } else if (rank == node) {
        MPI_Status stat;
        MPI_Recv(z,      // buff
            (int) 3*node_size, // count
            MPI_DOUBLE,  // type
            0,           // source
            rank,        // tag
            MPI_COMM_WORLD, // comm
            &stat);
    }
}

if (rank == 0) fclose(init);

FILE* timeline_file;

if (rank == 0) {
    timeline_file = fopen("data/timeline.data", "w");
    if (timeline_file == NULL) {
        printf("can't open timeline file\n");
        return 1;
    }

    fprintf(timeline_file, "%ld\n", n);
}

long long t1 = get_time();

output(timeline_file, z, n);
for (int k = 0; k < turns; ++k) {
    runge_kutta_step(z, n, h);
    output(timeline_file, z, n);
}

long long t2 = get_time();

printf("%d done work in %lld us.\n", rank, t2 - t1);
```

```
    if (rank == 0) fclose(timeline_file);

    free(z);

    MPI_Finalize();

    return 0;
}
```