

1 Parallel methods for solving the n -body gravitational problem

1.1 Statement of the problem and its mathematical model

The problem. The initial position of N bodies is given. Each body is also characterized by mass m (scalar quantity) and initial velocity v (vector). It is assumed that bodies are not affected by any forces other than gravitational ones. Under the influence of gravity, bodies change their position and velocity. You must model the evolution of a given set of interacting bodies.

The mathematical model of the problem uses the following formulas.

1. The gravitational law

$F = Gm_1 m_2 / r^2$, where m_1 and m_2 are the masses of interacting bodies, r is the distance between them, G is the gravitational constant equal to $6.67 \cdot 10^{-11} \text{ N}\cdot\text{m}^2/\text{kg}^2$, F is the absolute value the force with which each of the bodies acts on the other (the force acting on body i from body j is directed from body i to body j).

2. Rule of addition of forces

If several forces act on a body, then they can be replaced by one force equal to the vector sum of all the original forces.

3. Newton's second law

$F = ma$, where F is the force acting on the body, m is the mass of the body, a is the acceleration that the body acquires under the influence of a given force (F and a are vector quantities; their directions coincide).

The interaction of bodies is modeled step by step using discrete time intervals of fixed duration dt . At each step, the force acting on each body at the initial moment of time is calculated (using formulas 1 and 2), and the acceleration a is determined from the found force (formula 3). The law of body motion $S = S(t)$ is determined from the second order differential equation

$$d^2S / dt^2 = a(t),$$

which is equivalent to a system of two first-order differential equations:

$$dS / dt = V(t),$$

$$dV / dt = a(t),$$

where $V(t)$ is the velocity of the body at moment t .

Denoting $S_i = S(t_i)$ and $V_i = V(t_i)$, where $t_i = t_0 + i \cdot dt$, and using the simplest numerical method for solving a system of differential equations, in which the function $a(x)$ is replaced by a constant value $a_i = a(t_i)$ on each segment $[t_i, t_{i+1}]$, we obtain the following formulas for recalculating the position and velocity of the body:

$$V_{i+1} = V_i + a_i \cdot dt,$$

$$S_{i+1} = S_i + V_i \cdot dt + a_i \cdot dt^2 / 2.$$

For simplicity, it is assumed that all bodies are located in the same plane. In this case, all vector quantities associated with these bodies also lie in this plane. The position of each body is determined by two coordinates (x, y); each vector quantity is also defined by two coordinates: $F = (F_x, F_y)$, $a = (a_x, a_y)$, $V = (V_x, V_y)$.

In the case when the interacting bodies are located close to each other, the described simplest calculation method turns out to be unstable due to an increase in calculation errors. In this situation, one can use more accurate methods for integrating systems of differential equations, as well as a variable time step (we will not consider such modifications).

To ensure that the simplest version of the algorithm described above does not lead to unstable behavior of a system of closely located bodies, a threshold can be imposed on the maximum value of the force acting on the body from another (closely located) body. Of course, such a limitation cannot be considered justified from a physical point of view, however, when modeling a real system of astronomical bodies, the distance between them is so large that the forces never reach the specified threshold value. This limitation is intended only to ensure stable behavior of the model system of bodies used when testing the resulting program versions.

When modeling a system (taking into account the imposed restriction on the maximum value of the force), one can quite arbitrarily choose such characteristics as the masses of bodies, their velocities, the distances between them and the value of the gravitational constant.

1.2 Non-parallel version of the program

First, we present a less efficient version of the program, in which for each body all the forces acting on it are explicitly calculated.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <omp.h>

using namespace std;

#define gravity 10 // gravitational constant
#define dt 0.1 // time step
#define N 800 // number of bodies
#define fmax 1 // maximum force value
#define Niter 100 // number of iterations

struct Particle
{
    double x, y, vx, vy;
};

struct Force
{
    double x, y;
};

Particle p[N];
Force f[N];
double m[N];

void Init()
{
    for (int i = 0; i < N; i++)
    {
        p[i].x = 20 * (i / 20 - 20) + 10;
        p[i].y = 20 * (i % 20 - 10) + 10;

        p[i].vx = p[i].y / 15;
        p[i].vy = -p[i].x / 50;

        m[i] = 100 + i % 100;

        f[i].x = 0;
        f[i].y = 0;
    }
}

void CalcForces1()
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            if (i == j) continue;
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            f[i].x = f[i].x + fabs * dx * r_1;
            f[i].y = f[i].y + fabs * dy * r_1;
        }
}

void MoveParticlesAndFreeForces()
{
    for (int i = 0; i < N; i++)
```

```

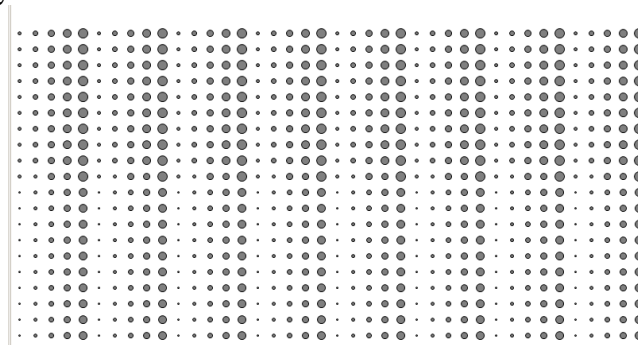
    {
        double dvx = f[i].x * dt / m[i],
               dvy = f[i].y * dt / m[i];
        p[i].x += (p[i].vx + dvx / 2) * dt;
        p[i].y += (p[i].vy + dvy / 2) * dt;
        p[i].vx += dvx;
        p[i].vy += dvy;
        f[i].x = 0;
        f[i].y = 0;
    }
}

void info(const char* s, double time)
{
    cout << setw(30) << left << s << "Time: " << fixed << setprecision(0) << setw(6) << 1000*time
         << setprecision(12) << "p0: " << setw(12) << p[0].x << ", " << setw(12) << p[0].y << endl;
}

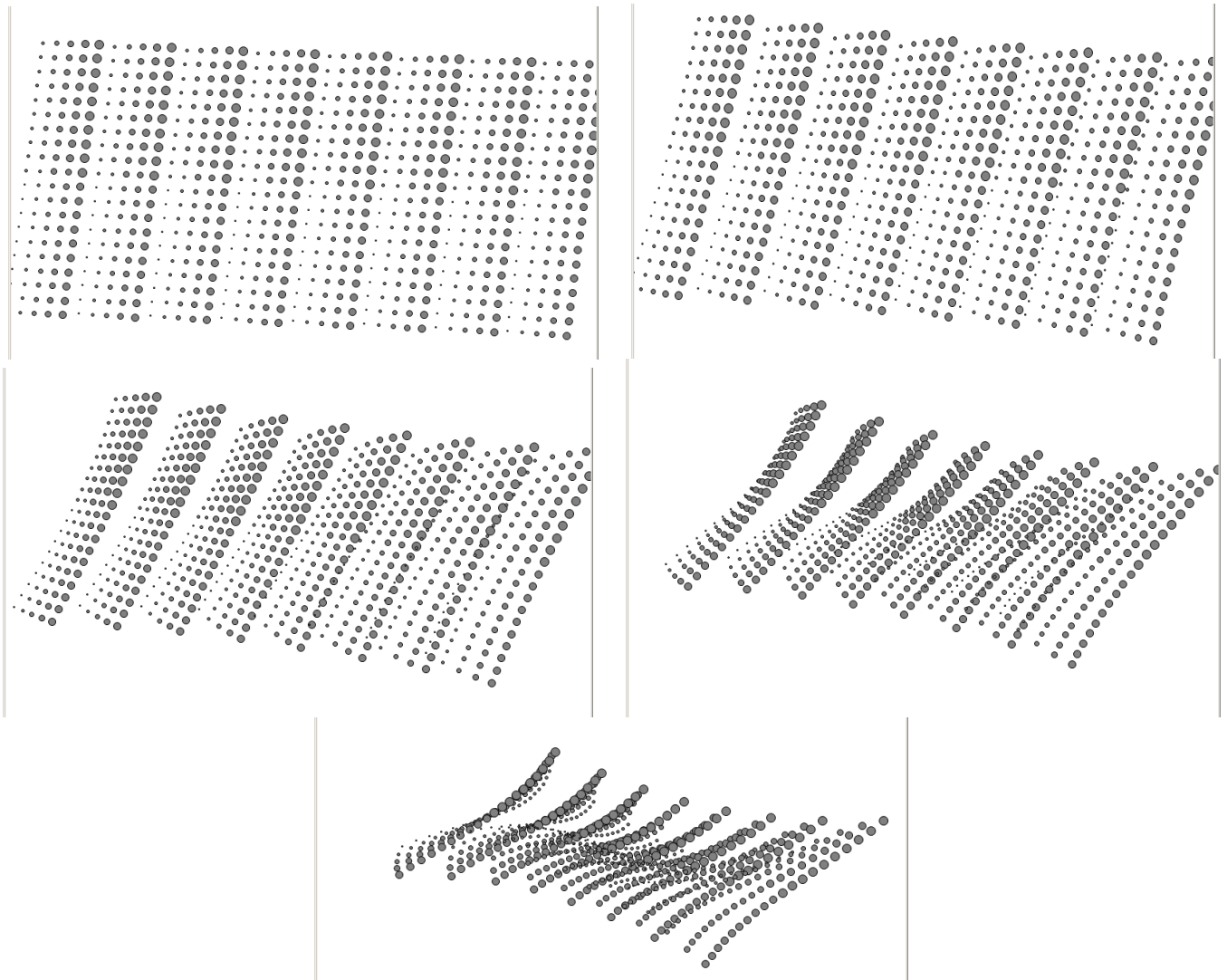
int main()
{
    double tt = 0;
    int Titer = 50;
    for (int kk = 1; kk <= Titer; kk++)
    {
        Init();
        double t = omp_get_wtime();
        for (int i = 0; i < Niter; i++)
        {
            CalcForces1();
            MoveParticlesAndFreeForces();
        }
        t = omp_get_wtime() - t;
        tt += t;
    }
    info("Non-parallel (N*N)", tt/Titer);
    std::system("pause");
    return 0;
}

```

A set of 800 point bodies (particles) located at the nodes of a rectangular grid and rotating around the center of this grid is used as a model system.



Particle masses vary in the range from 100 to 199. In the figures, more massive particles are presented in the form of circles with a large radius. Below are figures of the particle system after 20, 40, 60, 80 and 100 iterations.



The values of the coordinates of the initial particle after 100 iterations are output as test values, which will be used in the future to check the correctness of various modifications of the original program. In addition, the calculation time in milliseconds is displayed; the `omp_get_wtime()` function is used for this. For a more accurate estimate of time, the program performs 50 calculations and displays the average time required for one calculation.

Calculations were carried out in the Visual Studio 2017 environment.

When you run the above program, the following text will be output:

```
Non-parallel (N*N)           Time: 1665 p0: -285.496803732846 , 7.014089107234
```

The main part of the calculations is performed in the `CalcForces1` function, which determines the interaction forces between the bodies. The double loop used in it, in which both parameters are moved from 0 to $N-1$, leads to the fact that the same force arising from the interaction of two bodies is calculated twice. At the same time, when finding the force, the special techniques were used to speed up the calculations: in the auxiliary variable `r_2` the inverse square of the distance is stored, which allows us to subsequently avoid the use of the division operation (which takes longer than the multiplication operation); in another variable `r_1` the inverse value of the distance is stored; thus, the square root function is called only once.

An obvious way to speed up the calculation of forces is to arrange for the *simultaneous* determination of the forces acting between a pair of bodies. To do this, it is enough to change the loops so that the parameter of the inner loop `j` is always *greater than* the parameter of the outer loop `i`:

```
void CalcForces2()
{
    for (int i = 0; i < N - 1; i++)
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
```

```

        fabs = gravity * m[i] * m[j] * r_2;
        if (fabs > fmax) fabs = fmax;
        f[i].x += dx = fabs * dx * r_1;
        f[i].y += dy = fabs * dy * r_1;
        f[j].x -= dx;
        f[j].y -= dy;
    }
}

```

To avoid recalculating the force components, these components are stored in the dx and dy variables. When calculating using the CalcForces2 function, the result will be as follows:

Non-parallel (N*(N-1)/2) Time: 1087 p0: -285.496803732846, 7.014089107234

1.3 Parallel versions of the program using shared memory

1.3.1 Parallelizing an inefficient algorithm

Let's add a directive to the beginning of the program that determines the number of threads for the parallel version of the program:

```
#define Nthr 2 // number of threads
```

First we transform the function CalcForces1; to do this, just add the omp parallel for directive to its beginning:

```

void CalcForces1Par()
{
#pragma omp parallel for num_threads(Nthr)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            if (i == j) continue;
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            f[i].x += fabs * dx * r_1;
            f[i].y += fabs * dy * r_1;
        }
}

```

A similar directive can be added to the beginning of the MoveParticlesAndFreeForces function, although the effect of parallelizing this stage of the algorithm will be significantly less:

```

void MoveParticlesAndFreeForcesPar()
{
#pragma omp parallel for num_threads(Nthr)
    for (int i = 0; i < N; i++)
    {
        double dvx = f[i].x * dt / m[i],
            dvy = f[i].y * dt / m[i];
        p[i].x += (p[i].vx + dvx / 2) * dt;
        p[i].y += (p[i].vy + dvy / 2) * dt;
        p[i].vx += dvx;
        p[i].vy += dvy;
        f[i].x = 0;
        f[i].y = 0;
    }
}

```

Both threads will have the same load, so it is natural to expect that the program's running time will be approximately halved. In fact, the result is somewhat worse:

Parallel (N*N) Time: 969 p0: -285.496803732846, 7.014089107234

1.3.2 Options for parallelizing an efficient algorithm without load balancing

If you transform the CalcForces2 function in a similar way (by adding the directive `#pragma omp parallel` for `num_threads(Nthr)` to its beginning), the result will be as follows:

```
Parallel (N*(N-1)/2)      Time: 847   p0: -285.496720726482, 7.013726526215
```

In this case, the speed up compared to the original non-parallel algorithm turned out to be significantly less than two, since the threads are not load balanced. Even more important is the fact that the result of the calculations *differs* from what was previously obtained. This is explained by *thread competition* for access to the elements of array `f`, since each element is now modified by *different* iterations of the outer loop (which may be executed in parallel on different threads). In this case, a situation is possible when the value of some element of the array `f` will be simultaneously read and incremented by different threads; as a result, of course, one of the added forces will be lost.

To correct this problem, it is enough to protect the operators for changing the elements of the array `f` using a critical section:

```
void CalcForces2ParA()
{
#pragma omp parallel for num_threads(Nthr)
  for (int i = 0; i < N - 1; i++)
    for (int j = i + 1; j < N; j++)
    {
      double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
             r_2 = 1 / (dx * dx + dy * dy),
             r_1 = sqrt(r_2),
             fabs = gravity * m[i] * m[j] * r_2;
      if (fabs > fmax) fabs = fmax;
#pragma omp critical
      {
        f[i].x += dx = fabs * dx * r_1;
        f[i].y += dy = fabs * dy * r_1;
        f[j].x -= dx;
        f[j].y -= dy;
      }
    }
}
```

Unfortunately, the resulting program will be extremely inefficient, since one of the threads will regularly suspend its work waiting for the critical section to be freed:

```
Parallel (--, critical)   Time: 5627   p0: -285.496803732846, 7.014089107234
```

However, the modification achieves its goal: now the result of the calculations does not depend on the order in which threads access the elements of the array `f`.

In order to speed up calculations and at the same time prevent thread competition, it is enough to use *auxiliary arrays* of “additional forces” `tf`, connecting each thread with its own array of additives. As a result, there will be no competition when calculating additives. In this case, you will additionally have to provide a loop in which the elements of the `tf` arrays will be added to the corresponding elements of the `f` array:

```
Force tf[N][Nthr];
```

```
void Init()
{
  for (int i = 0; i < N; i++)
  {
    p[i].x = 20 * (i / 20 - 20) + 10;
    p[i].y = 20 * (i % 20 - 10) + 10;

    p[i].vx = p[i].y / 15;
    p[i].vy = -p[i].x / 50;

    m[i] = 100 + i % 100;

    f[i].x = 0;
```

```

    f[i].y = 0;
}
for (int i = 0; i < N; i++)
    for (int j = 0; j < Nthr; j++)
    {
        tf[i][j].x = 0;
        tf[i][j].y = 0;
    }
}

void CalcForces2ParB()
{
#pragma omp parallel for num_threads(Nthr)
    for (int i = 0; i < N - 1; i++)
    {
        int k = omp_get_thread_num();
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            tf[i][k].x += dx = fabs * dx * r_1;
            tf[i][k].y += dy = fabs * dy * r_1;
            tf[j][k].x -= dx;
            tf[j][k].y -= dy;
        }
    }
#pragma omp parallel for num_threads(Nthr)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < Nthr; j++)
        {
            f[i].x += tf[i][j].x;
            f[i].y += tf[i][j].y;
            tf[i][j].x = 0;
            tf[i][j].y = 0;
        }
}

```

This version of the program runs slower than the previous parallel versions (except for the program with a critical section), but it gives the correct result:

```
Parallel (--, add array)      Time: 1002  p0: -285.496803732846, 7.014089107234
```

1.3.3 Options for parallelizing an efficient algorithm with load balancing

It is natural to expect that if the latest version of the parallel program ensures that the load is balanced for the threads, the speed of the program will increase even more.

The simplest way to ensure a more balanced load on threads is to use the dynamic option :

```
#pragma omp parallel for num_threads(Nthr) schedule(dynamic, block)
```

Below are the calculation results for different values of the block parameter (the size of the “portion” of iterations that is assigned to each thread):

```
Parallel (--, --, dynamic 1) Time: 962  p0: -285.496803732846, 7.014089107234
```

```
Parallel (--, --, dynamic 25) Time: 725  p0: -285.496803732846, 7.014089107234
```

The speed of the program increases if each thread is assigned a sufficiently large portion of iterations.

Another way to ensure load balancing is to distribute iterations into “bands”: thread number k is assigned iteration numbers $k, k + Nthr, k + 2 * Nthr, \dots$, where $Nthr$ denotes the total number of threads (threads, like iterations, are numbered from zero). In this case, the loop iterations are distributed among the threads as follows:

0, 1, 2, ..., $Nthr-1$, 0, 1, 2, ..., $Nthr-1$, 0, 1, 2, ..., $Nthr-1$,

Here is a variant of the CalcForces function, which uses band distribution:

```

void CalcForces2ParD1()
{
#pragma omp parallel num_threads(Nthr)
{
    int k = omp_get_thread_num();
    for (int i = k; i < N - 1; i+=Nthr)
    {
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            tf[i][k].x += dx = fabs * dx * r_1;
            tf[i][k].y += dy = fabs * dy * r_1;
            tf[j][k].x -= dx;
            tf[j][k].y -= dy;
        }
    }
}
#pragma omp parallel for num_threads(Nthr)
for (int i = 0; i < N; i++)
for (int j = 0; j < Nthr; j++)
{
    f[i].x += tf[i][j].x;
    f[i].y += tf[i][j].y;
    tf[i][j].x = 0;
    tf[i][j].y = 0;
}
}

```

This version runs faster than the original version without load balancing, but it will lose to the version with dynamic thread distribution:

```
Parallel (--, --, bands)      Time: 799   p0: -285.496803732846, 7.014089107234
```

The most optimal load balancing is achieved by distributing flows across “backward bands”. In this case, iterations are distributed across threads in a “mirror” order:

0, 1, 2, ..., Nthr-1, Nthr-1, ..., 2, 1, 0, 0, 1, 2, ..., Nthr-1, Nthr-1, ..., 2, 1, 0,

Here is a variant of the CalcForces function, which uses backward band distribution:

```

void CalcForces2ParD2()
{
#pragma omp parallel num_threads(Nthr)
{
    int k = omp_get_thread_num();
    for (int i0 = 0; i0 < N / Nthr; i0++)
    {
        int i = (i0 % 2 == 0) ? i0 * Nthr + k : (i0 + 1) * Nthr - k - 1;
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            tf[i][k].x += dx = fabs * dx * r_1;
            tf[i][k].y += dy = fabs * dy * r_1;
            tf[j][k].x -= dx;
            tf[j][k].y -= dy;
        }
    }
}
#pragma omp parallel for num_threads(Nthr)
for (int i = 0; i < N; i++)

```



```

    for (int j = 0; j < Nthr; j++)
    {
        f[i].x += tf[i][j].x;
        f[i].y += tf[i][j].y;
        tf[i][j].x = 0;
        tf[i][j].y = 0;
    }
}

```

This version is not much faster than the "regular" band partitioning version, but is still slightly slower than the dynamic thread allocation option:

Parallel (--, --, b-bands) Time: 789 p0: -285.496803732846, 7.014089107234

You can try to speed up the backward band option by getting rid of the ternary operator and organizing the calculations in the form of *two* loops:

```

void CalcForces2ParD3()
{
#pragma omp parallel num_threads(Nthr)
{
    int k = omp_get_thread_num();
    for (int i = k; i < N - 1; i += 2 * Nthr)
    {
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            tf[i][k].x += dx = fabs * dx * r_1;
            tf[i][k].y += dy = fabs * dy * r_1;
            tf[j][k].x -= dx;
            tf[j][k].y -= dy;
        }
    }
    for (int i = 2 * Nthr - k - 1; i < N - 1; i += 2 * Nthr)
    {
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            tf[i][k].x += dx = fabs * dx * r_1;
            tf[i][k].y += dy = fabs * dy * r_1;
            tf[j][k].x -= dx;
            tf[j][k].y -= dy;
        }
    }
}
#pragma omp parallel for num_threads(Nthr)
for (int i = 0; i < N; i++)
    for (int j = 0; j < Nthr; j++)
    {
        f[i].x += tf[i][j].x;
        f[i].y += tf[i][j].y;
        tf[i][j].x = 0;
        tf[i][j].y = 0;
    }
}

```

This version is comparable in speed to the option with dynamic thread distribution:

Parallel (--, --, b-bands2) Time: 726 p0: -285.496803732846, 7.014089107234

For a more visual comparison of the speed of execution of various variants of the algorithm, we present them in a common list:

N = 800, Niter = 100, Nthr = 2

Non-parallel (N*N)	Time: 1665	p0: -285.496803732846 , 7.014089107234
Non-parallel (N*(N-1)/2)	Time: 1087	p0: -285.496803732846 , 7.014089107234
Parallel (N*N)	Time: 969	p0: -285.496803732846 , 7.014089107234
Parallel (N*(N-1)/2)	Time: 847	p0: -285.496720726482 , 7.013726526215
Parallel (--, critical)	Time: 5627	p0: -285.496803732846 , 7.014089107234
Parallel (--, add array)	Time: 1002	p0: -285.496803732846 , 7.014089107234
Parallel (--, --, dynamic 1)	Time: 962	p0: -285.496803732846 , 7.014089107234
Parallel (--, --, dynamic 25)	Time: 725	p0: -285.496803732846 , 7.014089107234
Parallel (--, --, bands)	Time: 799	p0: -285.496803732846 , 7.014089107234
Parallel (--, --, b-bands)	Time: 789	p0: -285.496803732846 , 7.014089107234
Parallel (--, --, b-bands2)	Time: 726	p0: -285.496803732846 , 7.014089107234

1.4 Parallel algorithms for solving the n-body problem using message passing

When implementing a parallel algorithm for solving the n -body problem based on message passing, you can use any of the three main models of process interaction used in distributed programming:

- *the manager-workers model* associated with the *task portfolio paradigm*;
- *a pulsing model*, in which groups of interacting processes alternate in the roles of sender and receiver;
- *a conveyor model*, in which messages circulate through a chain of processes, with each process always receiving data from the preceding process and passing data to the next.

In what follows we assume that the number of bodies N is proportional to the number of worker processes W_{proc} ; in this case, a block of N / W_{proc} bodies will be associated with each process. In the pulsing and conveyor models, the total number of processes is equal to the number of worker processes; in the manager-workers model, the total number of processes is $W_{\text{proc}} + 1$ (W_{proc} of worker processes + manager process).

Information about bodies includes two sets of data: a constant set of masses of bodies and a changing set of positions and velocities of bodies. While positions and velocities have to be sent between different processes in any implementation of a parallel algorithm, it is advisable to store the masses of bodies in each process in full. Therefore, in all the algorithms described below, it is assumed that an array m of size N is used to store the mass of bodies, and arrays of corresponding structures are used to store other characteristics of bodies (position, velocity, force acting on the body), similar to the Particle and Force structures described above. In this case, the sizes of these arrays of structures will, as a rule, be less than N .

1.4.1 Manager-workers model

The main problem that arises when parallelizing the algorithm for solving the n -body problem is due to the fact that in an effective non-parallel implementation of this algorithm, the outer loop associated with finding forces performs a different number of calculations at different iterations. Thus, if you simply divide the original set of bodies into blocks of equal size and process each block in a separate process, then the load on the processes will be different.

In the “manager-workers” model, the manager process first forms a portfolio of tasks, each of which is determined by a pair of block numbers (i, j) . Task (i, j) consists of calculating, for all bodies included in blocks i and j , the forces of interaction between them. It is clear that it is sufficient to consider blocks for which $i \leq j$. By numbering the blocks from 0 to $W_{\text{proc}} - 1$, we get the following pairs:

$(0, 0), (0, 1), \dots, (0, W_{\text{proc}} - 1), (1, 1), (1, 2), \dots, (1, W_{\text{proc}} - 1), \dots, (W_{\text{proc}} - 1, W_{\text{proc}} - 1)$.

So the total number of tasks is $W_{\text{proc}} \cdot (W_{\text{proc}} + 1)/2$.

Using a portfolio of tasks provides a good balance for work processes, but does not allow you to determine in advance which specific blocks of bodies will be processed by each process. Therefore, in this model, it makes sense to store information about the current state of all bodies in each process.

So, each worker process stores information about the mass of all bodies (double $m[N]$), about the current state of all bodies (Particle $p[N]$) and about the forces acting on bodies (Force $f_{\text{Total}}[N]$). In addition, each process must store another array of type Force (Force $f[N]$), which will store the part of the total force acting on each body, which is calculated in this process.

In this case, the manager process may not store data about interacting bodies at all, exchanging with worker processes only pairs of block numbers.

When the manager and worker processes interact, each worker process, which finishes its task, sends the manager a request message about a new pair of blocks for processing, and the manager sends this worker process either the next pair of blocks from the task portfolio, or, if the portfolio is empty, a special notification that the tasks have ended (as such a notification, you can, for example, send a pair $(-1, -1)$).

In order not to use a large number of checks when working with a portfolio of tasks, it is advisable to immediately add to it not only “real” pairs of blocks (in the amount of $W_{\text{proc}} \cdot (W_{\text{proc}} + 1)/2$), but also notifications about the exhaustion of tasks (in the amount of W_{proc}). This set of blocks is formatted as an array and is filled once — at the beginning of the manager process. All subsequent actions are repeated N_{iter} times. In this loop, an inner loop is launched on tasks from the portfolio, in which the manager process waits for notification from any of the worker processes that it is ready to accept the next task, and then sends to this process a task with the next number (in this case, it can be guaranteed that in the end each worker process will receive “empty” problem $(-1, -1)$). For such messaging, it is enough to use point-to-point communication functions.

Now let us describe the actions of the worker processes. First, in each process the characteristics of all bodies are initialized. All subsequent actions are repeated N_{iter} times.

Until the process receives an “empty” task in response to a request for the next task, actions are performed to calculate the interaction forces between the bodies included in the blocks i and j of the next received task (i, j) (the calculated forces are placed in the array f). In this case, two versions of the force calculation function should be provided: one for processing blocks of the form (i, i) , and the other for blocks (i, j) with $i \neq j$. After receiving an “empty” task, the process exits the task processing loop. After all work processes exit the task processing loop, it is necessary to combine, for each body, all the forces, which are calculated in each of the work processes. To do this, the easiest way is to use the collective reduction operation `MPI_Allreduce`, which will ensure that the resulting total values are sent to all working processes. After this, each work process must recalculate the states (i.e., positions and velocities) of those bodies that are included in the corresponding block (it is natural to associate a block of bodies of number i with a work process of rank i). In this case, you should also reset the values of the elements of the array f (the elements of the f_{Total} array do not need to be reset). Finally, after recalculating the states of the bodies, each process must send the new states of “its” bodies to all processes. To do this, it is convenient to use the collective operation `MPI_Bcast` (it is necessary to ensure that each worker process sends only that part of the state array p that was recalculated by this process).

Note. To enable collective operations to be performed only on worker processes, you must define a new communicator at the beginning of the program that includes only worker processes, and use this communicator in collective operations.

Let us present two versions of the results output by the program when processing the set of initial data described earlier. The program was implemented as an `MPIDebug 9` task; the process of rank 8 was used as the manager. The output was done using the `Show` and `ShowLine` functions. In addition to the total calculation time for each process, information was displayed on the number of pairs of bodies for which the gravitational interaction forces were calculated. Also, in order to check the correctness of the algorithm, the final positions for the first and last body from the original set were output. Calculations show a not very good balance of processes in terms of the number of calculated forces, but at the same time a high speed, which is comparable to the best speed achieved in a parallel program with shared memory. Due to the features of the algorithm, the number of forces found in each process differs in different runs.

```

0| 1> Time = 1561.64 PairsCount = 4850550
0| 2> -285.496803732846 7.014089107234
1| 1> Time = 1572.00 PairsCount = 4136000
2| 1> Time = 1571.87 PairsCount = 3740750
3| 1> Time = 1571.74 PairsCount = 4050950
4| 1> Time = 1571.60 PairsCount = 3490350
5| 1> Time = 1572.09 PairsCount = 3440750
6| 1> Time = 1571.69 PairsCount = 3040250
7| 1> Time = 1571.69 PairsCount = 5210400
7| 2> 368.910141051039 41.575105017689
8| 1> Manager Time = 1549.99

```

```

0| 1> Time = 1579.54 PairsCount = 4870650
0| 2> -285.496803732846 7.014089107234
1| 1> Time = 1589.97 PairsCount = 4320250
2| 1> Time = 1589.61 PairsCount = 3955500
3| 1> Time = 1589.77 PairsCount = 4000750
4| 1> Time = 1589.45 PairsCount = 3465200
5| 1> Time = 1590.03 PairsCount = 3196100
6| 1> Time = 1589.41 PairsCount = 2896200
7| 1> Time = 1589.64 PairsCount = 5255350
7| 2> 368.910141051039 41.575105017689
8| 1> Manager Time = 1539.45

```

1.4.2 Pulsing model

In the pulsing model, all processes are equal, that is, they are worker processes. In this model, each process is “responsible” for calculating the forces associated with the bodies of its block. In this case, of course, the process must receive the states of all bodies that interact with “its” bodies.

Let us recall that in an efficient version of the non-parallel algorithm, for each body, the forces of its interaction with bodies having *higher* order numbers were calculated; in this case, the found forces were immediately added to the total force for each of the two interacting bodies.

Taking this feature into account, it is enough, for any process in the pulsing algorithm, to receive the states of bodies only from those processes whose rank *exceeds* the rank of the given process. However, each process must “return” to those processes that transferred states to it the corresponding values of the calculated forces. Thus, one part of the total force acting on each body from a block associated with a certain process is calculated in the process itself, and the other part is sent to this process from processes of *lower* rank.

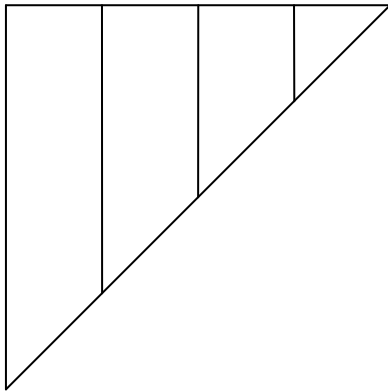


Fig. 1. Blocks of the same size

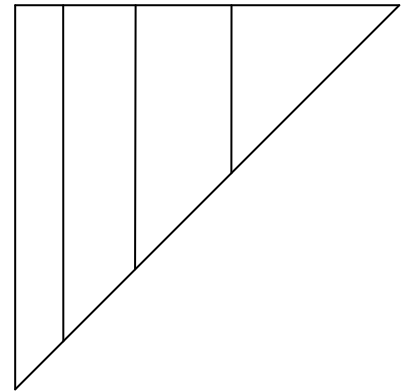


Fig. 2. Blocks of different sizes

It is clear that with a uniform distribution of bodies across blocks, the work of the processes will not be balanced: a process of rank 0 must receive the states of bodies from all other processes and calculate all the forces acting on these bodies (and also it must send the corresponding forces to all processes of a higher rank), and the process of rank $W_{\text{proc}} - 1$ must calculate *only* the forces of interaction between its bodies, send the states of its bodies to all processes of lower rank and receive from them the values of the forces calculated in these processes. Using the diagram shown in Fig. 1, we can say that each process must process its own *column* of data, sending the calculated forces “up” its column, with the “area” of the column corresponding to the amount of calculations that a given process must perform.

For more uniform load balancing, it is enough to use *blocks of different sizes*, allocating the smallest number of bodies for a process of rank 0, and the largest number for a process of rank $W_{\text{proc}} - 1$ (see Fig. 2).

Thus, in this method, you will need to first generate `blockSize` and `blockStart` arrays, the size of which is equal to the number of processes. For a process of rank i , the `blockSize[i]` value determines the number of bodies included in this block, and the `blockStart[i]` value is the index of the initial body included in this block.

To form these arrays, you can use the following algorithm: first, calculate the average number `avrSize` of interacting pairs of bodies for each block, equal to $N \cdot (N - 1) / 2 W_{\text{proc}}$, then, starting from the top right pair $(N - 1, N - 1)$, pairs of bodies located in columns are summed up until the number of pairs exceeds `avrSize`, after which all bodies corresponding to the processed columns, are associated with the next block, and the process is repeated. These calculations should be carried out for each work process. During these calculations, you also need to determine the size of the *maximum* block, `maxBlockSize`, since this is the size that needs to be provided for the arrays used to transfer data between processes.

In addition to forming the `blockSize` and `blockStart` arrays, at the beginning of each process, it is necessary to initialize the states of those bodies that belong to the block associated with this process. Recall that to reduce the number of data transfers, it is advisable to store the masses of all bodies `m[N]` in each process. Memory for other arrays should be allocated dynamically: for a process of rank `i`, arrays `p` and `f` (of type `Particle` and `Force`, respectively) should have size `blockSize[i]`, and arrays `tp` and `tf`, intended for receiving data (states and forces) from other processes, must have size `maxBlockSize`.

We now briefly describe the subsequent actions performed by each worker process. All these actions are repeated `Niter` times.

First, each process sends data from its array `p` to all processes of *lower* rank. For this transfer, you can use the non-blocking `MPI_Isend` command, which allows you to continue program execution without waiting for the end of the message transfer actions. The process then calculates the forces acting between the bodies of its block.

After this, for all processes whose rank *exceeds* the rank of this process, the following three actions are performed: the process receives the states of bodies from a process of higher rank, calculates the forces acting between its bodies and the bodies obtained from this process (in this case, the forces for their bodies are accumulated in the `f` array, and the forces for bodies of another process are accumulated in the `tf` array, which must be reset to zero at the beginning of the force calculation procedure), and sends back the force array `tf`.

Finally, the process receives, from all processes of lower rank, the arrays of forces calculated in them and adds these forces (located in the array `tf`) to the array `f`.

The last action is to recalculate the state of the bodies using forces `f` and reset the array `f`.

Let us present a version of the results produced by the program when processing the set of initial data described earlier. The program was implemented as an `MPIDebug8` task; all processes were working processes. The output was performed using the `Show` and `ShowLine` functions. In addition to the total calculation time for each process, information was displayed on the number of pairs of bodies for which the gravitational interaction forces were calculated. Also, in order to check the correctness of the algorithm, the final positions for the first and last body from the original set were output. Calculations show a good balance of processes in terms of the number of calculated forces. At the same time, in terms of execution speed, this algorithm is inferior to the “manager-workers” algorithm.

```

0| 1> Time = 1757.50 PairsCount = 3797500
0| 2> -285.496803732846 7.014089107234
1| 1> Time = 1783.95 PairsCount = 4046000
2| 1> Time = 1781.43 PairsCount = 4050400
3| 1> Time = 1785.88 PairsCount = 4020000
4| 1> Time = 1780.60 PairsCount = 4016600
5| 1> Time = 1790.49 PairsCount = 4009500
6| 1> Time = 1786.08 PairsCount = 4001400
7| 1> Time = 1790.46 PairsCount = 4018600
7| 2> 368.910141051039 41.575105017689

```

1.4.3 Conveyor model

In the conveyor model, as in the pulsing model, all processes are equal, that is, are the worker processes. A common property of these models is that each process is “responsible” for calculating the forces associated with the bodies of its block. However, in the conveyor model, a process receives data about the state of bodies associated with other processes not directly from them, but along a chain, always receiving data from its “left” neighbor and sending data to its “right” neighbor. To correctly determine the ranks of the left and right neighbors for a process of rank `i`, one should use the formulas $(i + W_{\text{proc}} - 1) \% W_{\text{proc}}$ and $(i + 1) \% W_{\text{proc}}$ respectively.

In the simplest version of the conveyor model, at each of the Niter iterations of the algorithm, each process first sends bodies from its block to the right process, then calculates the interaction forces between the bodies of its block, after which, in a loop repeating $W_{\text{proc}} - 1$ time, receives the next block of bodies from its left neighbor and calculates the forces between its bodies and the bodies from the received block. After calculating all the forces acting on bodies from its block, the process recalculates the states of its bodies.

Unfortunately, in the simplest version described, the forces between bodies from different blocks are calculated *twice* : in each of the processes associated with these blocks. In this respect, this version is similar to the first of the previously considered non-parallel algorithms.

In order not to duplicate the calculation of forces, you can modify the algorithm as follows: each process sends to its right neighbor not only a set of states of the next block of bodies, but also *a set of forces* acting on these bodies (initially this set contains zero values). Having received this data from the left neighbor, the process calculates the interaction forces only for those of its bodies whose “true” numbers (that is, the numbers of bodies in the original set before dividing it into blocks) *are less than* the true numbers of the bodies interacting with them, obtained from another process . In addition, the process adds the calculated “paired” forces to the resulting set of forces, and then sends the data further. After the sent data block has made a “full loop”, it will return to the process that sent it, and at the same time it will contain the part of the forces acting on the bodies of this process that were calculated in other processes. All that remains is to add this part to the forces calculated in the process itself and recalculate the states of their bodies. In addition to recalculating the state, at the final stage of each iteration it is necessary to reset the force values.

This modification of the original inefficient algorithm leads to an imbalance of processes: processes with *lower* ranks will calculate a *larger* number of forces. In this model, instead of introducing blocks of different sizes into consideration (as in the pulsing model discussed earlier), another technique can be used: distributing bodies *into bands* while maintaining the same blockSize for all blocks ($\text{blockSize} = N / W_{\text{proc}}$). Distribution by bands in this case means that each block will include one body from each band, into which the entire set of bodies is divided. For example, in the case of partitioning into simple bands, the i -th block will include bodies with numbers i , $i + W_{\text{proc}}$, $i + 2 \cdot W_{\text{proc}}$,

Such an approach will require modification of the procedure for initializing the states of bodies associated with the i -th process, but will ensure a good balance of processes. When determining which forces need to be calculated in a given process and which forces are left to be calculated by the “paired” process, it is enough, as in the previous modification of the algorithm, to compare the “true” numbers of the processed bodies and calculate the forces only in the case when the body number, related to the current process, *is less than* the number of the body received from another process (the type of inequality does not matter: you can also use a variant of the algorithm in which forces are calculated if the number of the body from the current process *is greater than* the number of the body received from another process).

The perfect balance, in which each process calculates *the same number of forces*, can be achieved by dividing the bodies into *backward bands* (this partition is described in the section devoted to parallel algorithms with shared memory).

With such a modification of the original algorithm, it is advisable to define a new ParticleForce data structure, including both data on the state of a certain body (position x , y and velocity v_x , v_y), and data on the forces acting on it (f_x , f_y). In each process, it is necessary to declare arrays p and tp of type ParticleForce, each of which has a size of blockSize . The p array stores information about the “own” bodies of the process (including the values of forces calculated in this process), and the tp array is used to receive and subsequently send data about the state of bodies from other processes, and, after completion of conveyor calculations, the tp array is used to receive data on additional forces for the “own” bodies of the process, calculated in other processes.

Let us present variants of the results output by the program when processing the set of initial data described earlier. The program was implemented as an MPIDebug8 task; all processes were worker processes. The output was done using the Show and ShowLine functions . In addition to the total calculation time for each process, information was displayed on the number of pairs of bodies for which the gravitational interaction forces were calculated. Also, in order to check the correctness of the algorithm, the final positions for the first and last body from the original set were output.

The first version corresponds to an algorithm without process balancing (in each process, the forces were calculated when processing a pair of bodies in which the number of “its” body was *less than* the number of the

body obtained from another process). Despite the obvious imbalance in the number of forces found, the operating time of the processes is comparable to the time obtained for the pulsing model.

```
0| 1> Time = 1736.41 PairsCount = 7495000
0| 2> -285.496803732846 7.014089107234
1| 1> Time = 1748.09 PairsCount = 6495000
2| 1> Time = 1750.67 PairsCount = 5495000
3| 1> Time = 1752.22 PairsCount = 4495000
4| 1> Time = 1750.43 PairsCount = 3495000
5| 1> Time = 1750.01 PairsCount = 2495000
6| 1> Time = 1752.19 PairsCount = 1495000
7| 1> Time = 1752.83 PairsCount = 495000
7| 2> 368.910141051039 41.575105017689
```

The second version corresponds to an algorithm with balancing based on partitioning into bands. The algorithm turned out to be almost balanced in terms of the number of calculated forces, but the operating time of each process increased (due to additional calculations associated with determining the “true” numbers of interacting bodies).

```
0| 1> Time = 2212.17 PairsCount = 4030000
0| 2> -285.496803732846 7.014089107234
1| 1> Time = 2226.92 PairsCount = 4020000
2| 1> Time = 2228.65 PairsCount = 4010000
3| 1> Time = 2229.17 PairsCount = 4000000
4| 1> Time = 2227.52 PairsCount = 3990000
5| 1> Time = 2228.85 PairsCount = 3980000
6| 1> Time = 2226.93 PairsCount = 3970000
7| 1> Time = 2227.18 PairsCount = 3960000
7| 2> 368.910141051039 41.575105017689
```

The third option corresponds to an algorithm with balancing based on partitioning into backward bands. The algorithm is completely balanced in terms of the number of calculated forces and gives a slightly shorter time compared to the previous balancing option, but the resulting time is still inferior to the version without any balancing. It should be noted that, due to the partitioning into backward bands, both the first and last bodies from the original set are processed in the process of rank 0.

```
0| 1> Time = 2210.63 PairsCount = 3995000
0| 2> -285.496803732846 7.014089107234
0| 3> 368.910141051039 41.575105017689
1| 1> Time = 2225.66 PairsCount = 3995000
2| 1> Time = 2225.72 PairsCount = 3995000
3| 1> Time = 2222.78 PairsCount = 3995000
4| 1> Time = 2219.32 PairsCount = 3995000
5| 1> Time = 2222.57 PairsCount = 3995000
6| 1> Time = 2219.82 PairsCount = 3995000
7| 1> Time = 2224.73 PairsCount = 3995000
```

2 Learning tasks (the MPIGravit group)

2.1 Tasks formulations

MPIGravit1. Implement the *manager-workers method* and test it on an 800-point input data set of 800 points (described in the program template). The number of processes is always 9, the manage process has a rank 8. In addition to the results required in the task, output (in the debug section) additional information about the running time of each process and the number of processed pairs of points during the force calculation. Output the number of processed pairs only in work processes. When debugging the program, use the non-parallel version of the algorithm already available in the template.

MPIGravit2. Implement the *pulsing method* and test it on an 800-point input data set (described in the program template). The number of processes is always 8. In addition to the results required in the task, output (in the debug section) additional information about the running time of each process and the number of processed pairs of points during the force calculation. When debugging the program, use the non-parallel version of the algorithm already available in the template.

MPIGravit3. Implement the *band partitioned conveyor method* and test it on an 800-point input dataset (described in the program template). The number of processes is always 8. In addition to the results required in the task, output (in the debug section) additional information about the running time of each process and the number of processed pairs of points during the force calculation. When debugging the program, use the non-parallel version of the algorithm already available in the template.

MPIGravit4. Implement the *backward band partitioned conveyor method* and test it on an 800-point input data set (described in the program template). The number of processes is always 8. In addition to the results required in the task, output (in the debug section) additional information about the running time of each process and the number of processed pairs of points during the force calculation. When debugging the program, use the non-parallel version of the algorithm already available in the template.

2.2 Screenshots and program templates

View of the taskbook window during a demo run of the MPIGravit1 task:

```

Programming Taskbook [C++]
PARALLEL METHODS FOR SOLVING THE GRAVITY PROBLEM
Task: MPIGravit1
Demo running: Smith John
Color (F3) Mode (F4)

New data (Space) Previous task (BS) Next task (Enter) Exit (Esc)

Implement the manager-workers method and test it on an 800-point
input data set of 800 points (described in the program template). The number
of processes is always 9, the manage process has a rank 8. In addition
to the results required in the task, output (in the debug section)
additional information about the running time of each process and the number
of processed pairs of points during the force calculation. Output the number
of processed pairs only in work processes. When debugging the program, use
the non-parallel version of the algorithm already available in the template.

Input data
Process 0: number of iterations: 42
Sample output of debug information:
0| 1> Time = 1561.64 PairsCount = 4850550
1| 1> Time = 1572.00 PairsCount = 4136000
2| 1> Time = 1571.87 PairsCount = 3740750
3| 1> Time = 1571.74 PairsCount = 4050950
4| 1> Time = 1571.60 PairsCount = 3490350
5| 1> Time = 1572.09 PairsCount = 3440750
6| 1> Time = 1571.69 PairsCount = 3040250
7| 1> Time = 1571.69 PairsCount = 5210400
8| 1> Manager Time = 1549.99

Example of right solution
PARALLEL ALGORITHM
After one iteration
Process 0: coordinates of point 0: -391.246774969040 -189.205356091407
Process 7: coordinates of point 799: 391.253372618512 189.211411283936
After the required number of iterations
Process 0: coordinates of point 0: -406.651483017894 -133.211219368686
Process 7: coordinates of point 799: 418.930563967868 143.157898448842

```


Contents of the template program for MPIGravit1 task:

```

#include "pt4.h"
#include "mpi.h"
#include <cmath>

#define gravity 10 // gravitational constant
#define dt 0.1 // time step
#define N 800 // number of particles
#define fmax 1 // max force value

int k; // number of processes
int r; // rank of the current process
int Niter; // number of iterations

struct Particle
{
    double x, y, vx, vy;
};

struct Force
{
    double x, y;
};

Particle p[N];
Force f[N];
double m[N];

void Init()
{
    for (int i = 0; i < N; i++)
    {
        p[i].x = 20 * (i / 20 - 20) + 10;
        p[i].y = 20 * (i % 20 - 10) + 10;
        p[i].vx = p[i].y / 15;
        p[i].vy = -p[i].x / 50;

        m[i] = 100 + i % 100;

        f[i].x = 0;
        f[i].y = 0;
    }
}

// Implementation of a non-parallel algorithm

void CalcForces2()
{
    for (int i = 0; i < N - 1; i++)
        for (int j = i + 1; j < N; j++)
        {
            double dx = p[j].x - p[i].x, dy = p[j].y - p[i].y,
                r_2 = 1 / (dx * dx + dy * dy),
                r_1 = sqrt(r_2),
                fabs = gravity * m[i] * m[j] * r_2;
            if (fabs > fmax) fabs = fmax;
            f[i].x += dx = fabs * dx * r_1;
            f[i].y += dy = fabs * dy * r_1;
            f[j].x -= dx;
            f[j].y -= dy;
        }
}

```

```

void MoveParticlesAndFreeForces()
{
    for (int i = 0; i < N; i++)
    {
        double dvx = f[i].x * dt / m[i],
               dvy = f[i].y * dt / m[i];
        p[i].x += (p[i].vx + dvx / 2) * dt;
        p[i].y += (p[i].vy + dvy / 2) * dt;
        p[i].vx += dvx;
        p[i].vy += dvy;
        f[i].x = 0;
        f[i].y = 0;
    }
}

void NonParallelCalc(int n)
{
    Init();
    for (int i = 0; i < n; i++)
    {
        CalcForces2();
        MoveParticlesAndFreeForces();
    }
}

// End of the non-parallel algorithm implementation

// Implementation of the parallel algorithm
// based on the "manager-worker" method

// End of parallel algorithm implementation

void Solve()
{
    Task("MPIGravit1_en");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
    if (r == 0)
    {
        pt >> Niter;
        // Testing the non-parallel algorithm:

        ShowLine("NON-PARALLEL ALGORITHM");
        NonParallelCalc(1);

        ShowLine("After one iteration");
        SetPrecision(12);
        Show("    Coordinates of point 0:  ", p[0].x, 17);
        ShowLine(p[0].y, 17);
        Show("    Coordinates of point 799: ", p[799].x, 17);
        ShowLine(p[799].y, 17);

        double t = MPI_Wtime();
        NonParallelCalc(Niter);
    }
}

```

```

t = MPI_Wtime() - t;

ShowLine("After the required number of iterations");
Show("    Coordinates of point 0:  ", p[0].x, 17);
ShowLine(p[0].y, 17);
Show("    Coordinates of point 799: ", p[799].x, 17);
ShowLine(p[799].y, 17);

SetPrecision(2);
ShowLine("Non-parallel algorithm running time: ", t * 1000);
}
MPI_Bcast(&Niter, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Testing the parallel algorithm:
}

```

View of the taskbook window during the trial run of the MPIGravit1 task:

```

Programming Taskbook [C++]
PARALLEL METHODS FOR SOLVING THE GRAVITY PROBLEM
Task: MPIGravit1
Student: Smith John
Results (F2) Color (F3) Mode (F4)

Correct data input:
all required data are input, no data are output.
Exit (Esc)

Input: 1 of 1
Output: 0 of 8
Tests: 1 of 3

Implement the manager-workers method and test it on an 800-point
input data set of 800 points (described in the program template). The number
of processes is always 9, the manage process has a rank 8. In addition
to the results required in the task, output (in the debug section)
additional information about the running time of each process and the number
of processed pairs of points during the force calculation. Output the number
of processed pairs only in work processes. When debugging the program, use
the non-parallel version of the algorithm already available in the template.

Input data
Process 0: number of iterations: 42
Sample output of debug information:
0| 1> Time = 1561.64 PairsCount = 4850550
1| 1> Time = 1572.00 PairsCount = 4136000
2| 1> Time = 1571.87 PairsCount = 3740750
3| 1> Time = 1571.74 PairsCount = 4050950
4| 1> Time = 1571.60 PairsCount = 3490350
5| 1> Time = 1572.09 PairsCount = 3440750
6| 1> Time = 1571.69 PairsCount = 3040250
7| 1> Time = 1571.69 PairsCount = 5210400
8| 1> Manager Time = 1549.99

Example of right solution
PARALLEL ALGORITHM
After one iteration
Process 0: coordinates of point 0: -391.246774969040 -189.205356091407
Process 7: coordinates of point 799: 391.253372618512 189.211411283936
After the required number of iterations
Process 0: coordinates of point 0: -406.651483017894 -133.211219368686
Process 7: coordinates of point 799: 418.930563967868 143.157898448842

0| 1> NON-PARALLEL ALGORITHM
0| 2> After one iteration
0| 3> Coordinates of point 0: -391.246774969040 -189.205356091407
0| 4> Coordinates of point 799: 391.253372618512 189.211411283936
0| 5> After the required number of iterations
0| 6> Coordinates of point 0: -406.651483017894 -133.211219368686
0| 7> Coordinates of point 799: 418.930563967868 143.157898448842
0| 8> Non-parallel algorithm running time: 150.02

```