Параллельные методы решения гравитационной задачи *n* тел

Постановка задачи и ее математическая модель

Задача. Дано начальное положение N точечных тел. Каждое тело характеризуется массой m (скалярная величина) и начальной скоростью v (вектор). Предполагается, что на тела не действуют никакие силы, кроме гравитационных. Под действием гравитации тела изменяют свое положение, а также скорость. Промоделировать эволюцию данного набора взаимодействующих тел.

Математическая модель задачи использует следующие формулы.

1. **Закон всемирного тяготения**

$F = Gm_1m_2/r^2$, где m_1 и m_2 – массы взаимодействующих тел, r – расстояние между ними, G – гравитационная постоянная, равная $6{,}67\cdot10^{-11}$ Н·м^2/кг^2, F – абсолютная величина силы, с которой каждое из тел действует на другое (сила, действующая на тело i со стороны тела j, направлена от тела i к телу j).

2. **Правило сложения сил**

Если на тело действуют несколько сил, то их можно заменить одной силой, равной векторной сумме всех исходных сил.

3. **Второй закон Ньютона**

$F = ma$, где F – сила, действующая на тело, m – масса тела, a – ускорение, которое тело приобретает под действием данной силы (F и a являются векторными величинами; их направления совпадают).

Взаимодействие тел моделируется пошагово с помощью дискретных отрезков времени фиксированной длительности dt. На каждом шаге вычисляется сила, действующая на каждое тело в начальный момент времени (с применением формул 1 и 2), по найденной силе определяется ускорение a (формула 3). Закон движения тела $S = S(t)$ определяется из дифференциального уравнения второго порядка

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

которое равносильно системе двух дифференциальных уравнений первого порядка:

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

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

где $V(t)$ – скорость тела в момент t.

Обозначая $S_i = S(t_i)$ и $V_i = V(t_i)$, где $t_i = t_0 + i\cdot dt$, и используя простейший численный метод решения системы дифференциальных уравнений, при котором функция $a(x)$ заменяется постоянным значением $a_i = a(t_i)$ на каждом отрезке $[t_i, t_{i+1}]$, получаем следующие формулы для пересчета положения и скорости тела:

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

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

Для простоты предполагается, что все тела расположены в одной плоскости. В этом случае все векторные величины, связанные с этими телами, также лежат в этой плоскости. Положение каждого тела определяется двумя координатами (x, y); каждая векторная величина также определяется двумя координатами: $F = (F_x, F_y)$. $a = (a_x. a_y)$. $V = (V_x, V_y)$.

В случае, когда взаимодействующие тела располагаются близко друг от друга, описанный простейший метод расчета оказывается неустойчивым в силу увеличения погрешностей вычисления. В данной ситуации можно использовать более точные методы интегрирования систем дифференциальных уравнений, а также переменный шаг по времени (мы не будем рассматривать подобные модификации).

Для того чтобы описанный выше простейший вариант алгоритма не приводил к нестабильному поведению системы близко расположенных тел, можно наложить ограничение на *максимальное значение силы*, действующей на тело со стороны другого (близко расположенного) тела. Разумеется, подобное ограничение нельзя считать оправданным с физической точки зрения, однако при моделировании реальной системы астрономических тел расстояние между ними является настолько большим, что силы никогда не достигают указанного порогового значения. Данное ограничение предназначено лишь для того, чтобы обеспечить устойчивое поведение *модельной системы тел*, используемой при тестировании полученных вариантов программы.

При моделировании системы (с учетом наложенного ограничения на максимальное значение силы) можно достаточно произвольным образом выбирать такие характеристики, как массы тел, их скорости, расстояния между ними и значение гравитационной постоянной.

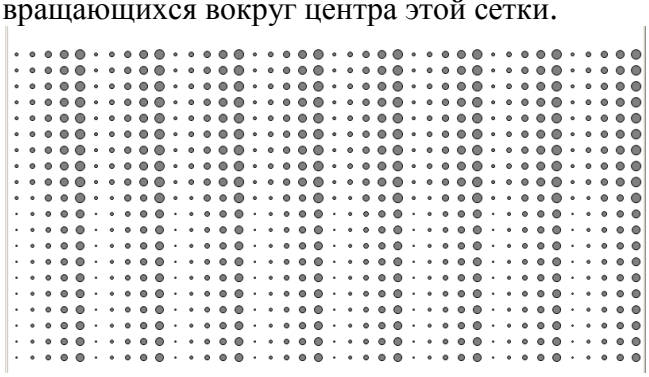
Непараллельный вариант программы

Вначале приведем менее эффективный вариант программы, в котором для каждого тела явным образом вычисляются все действующие на него силы.

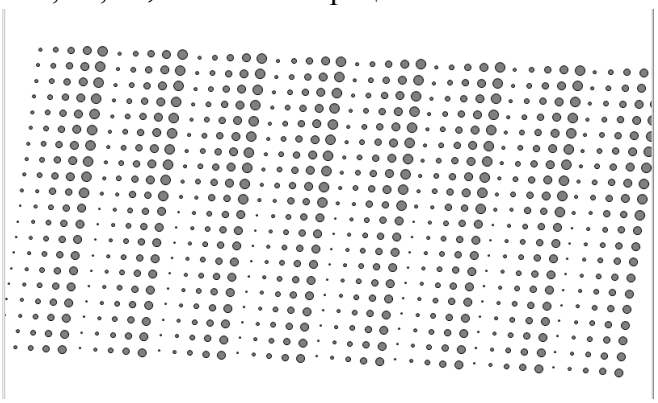
```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <omp.h>

using namespace std;

#define gravity 10 // гравитационная постоянная
#define dt 0.1     // шаг по времени
#define N 800      // количество частиц
#define fmax 1     // максимальное значение силы
#define Niter 100  // число итераций

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;
        }
}
```

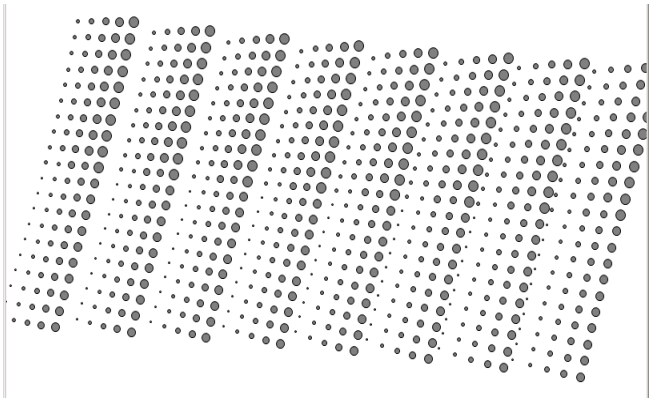
```
void MoveParticlesAndFreeForces()
{
    for (int i = 0; i < N; i++)
    {
        double dvx = f[i].x * dt / m[i],
            dvy = f[i].y * dt / m[i];
        p[i].x += (p[i].vx + dvx / 2) * dt;
        p[i].y += (p[i].vy + dvy / 2) * dt;
        p[i].vx += dvx;
        p[i].vy += dvy;
       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

void info(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;
}

void main()
{
    Init();
    double t = omp_get_wtime();
    for (int i = 0; i < Niter; i++)
    {
        CalcForces1();
        MoveParticlesAndFreeForces();
    }
    t = omp_get_wtime() - t;
    info("Non-Parallel (N*N)", t);
}
```

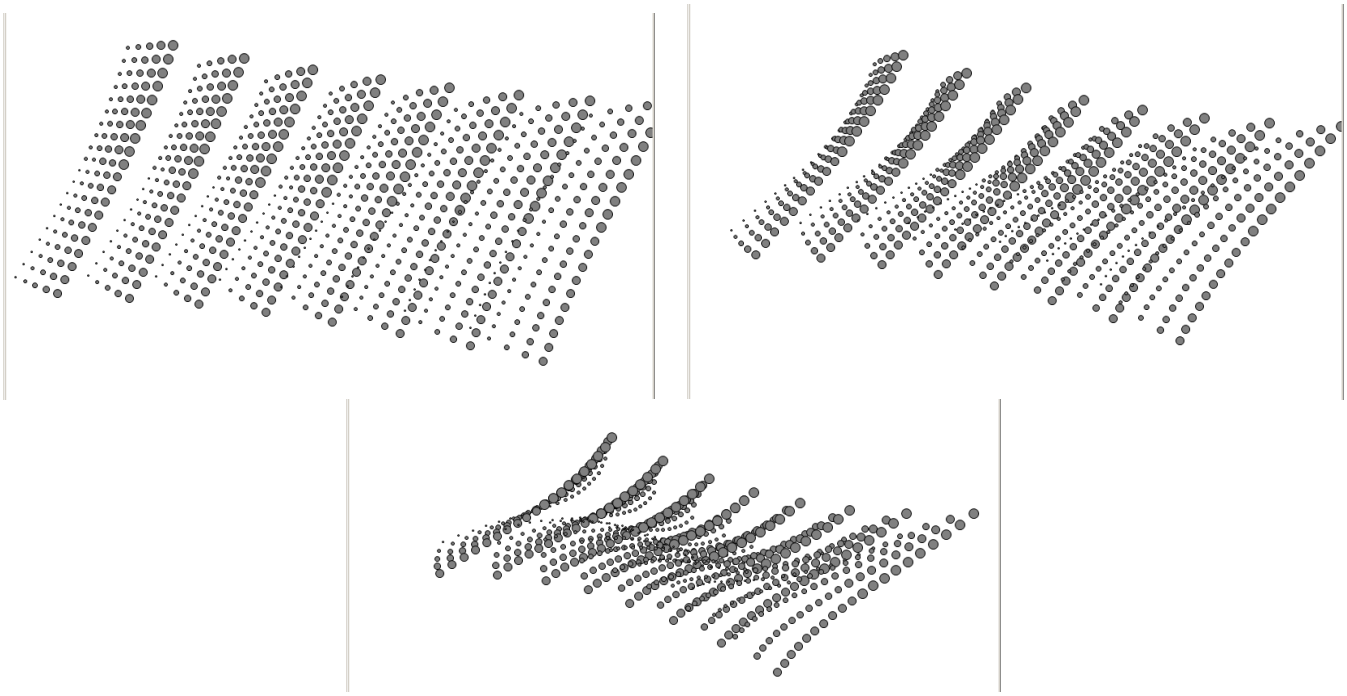
В качестве модельной системы используется набор из 800 точечных тел (частиц), расположенных в узлах прямоугольной сетки и вращающихся вокруг центра этой сетки.



Массы частиц изменяются в диапазоне от 100 до 199; на приведенных рисунках более массивные частицы представлены в виде кругов с большим радиусом. Ниже приведены изображения системы частиц после 20, 40, 60, 80 и 100 итераций.







В качестве тестовых значений, которые в дальнейшем будут использоваться для проверки правильности различных модификаций исходной программы, выводятся значения координат начальной частицы после 100 итераций. Кроме того, выводится время расчета в миллисекундах; для этого используется функция omp_get_time().

При выполнении приведенной программы будет выведен следующий текст:

```
Non-Parallel (N*N)          Time: 8358  p0: -285.496803732846, 7.014089107234
```

Основная часть вычислений выполняется в функции CalcForces1, определяющей силы взаимодействия между телами. Использованный в ней двойной цикл, в котором оба параметра перебираются от 0 до N–1, приводит к тому, что одна и та же сила, возникающая при взаимодействии двух тел, вычисляется дважды. Вместе с тем, при нахождении силы использованы приемы, позволяющие ускорить вычисления: во вспомогательной переменной r_2 сохраняется величина, обратная квадрату расстояния, что позволяет в дальнейшем избежать применения операции деления (которая выполняется дольше, чем операция умножения); еще в одной переменной r_1 сохраняется величина, обратная расстоянию; таким образом, функция извлечения квадратного корня вызывается единственный раз.

Очевидным способом ускорить вычисление сил является организация *одновременного* нахождения сил, действующих между парой тел. Для этого достаточно изменить циклы так, чтобы параметр внутреннего цикла j был всегда *больше* параметра внешнего цикла 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;
        }
}
```

Чтобы избежать повторного вычисления компонентов сил, эти компоненты сохраняются в переменных dx и dy.

При вычислениях, использующих функцию CalcForces2, результат будет следующим:

```
Non-Parallel (N*(N-1)/2)    Time: 5024  p0: -285.496803732846, 7.014089107234
```

Параллельные варианты программы, использующие общую память

Распараллеливание неэффективного алгоритма

Добавим в начало программы директиву, определяющую число потоков для параллельного варианта программы:

```
#define Nthr 2     // число потоков
```

Вначале преобразуем функцию CalcForces1; для этого достаточно добавить в ее начало директиву omp parallel for:

```
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;
        }
}
```

Аналогичную директиву можно добавить и в начало функции MoveParticlesAndFreeForces, хотя эффект от распараллеливания данного этапа алгоритма будет существенно меньшим:

```
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;
    }
}
```

Оба потока будут иметь одинаковую нагрузку, поэтому естественно ожидать, что время работы программы уменьшится примерно вдвое. Результат оказывается даже лучшим, чем ожидалось:

```
Parallel (N*N)              Time: 3373  p0: -285.496803732846, 7.014089107234
```

Варианты распараллеливания эффективного алгоритма без балансировки нагрузки

При аналогичном преобразовании функции CalcForces2 (посредством добавления в ее начало директивы #pragma omp parallel for num_threads(Nthr)) результат будет следующим:

```
Parallel (N*(N-1)/2)        Time: 2887  p0: -285.496799994540, 7.014052806288
```

В данном случае ускорение, по сравнению с исходным непараллельным алгоритмом оказалось меньше двух, поскольку потоки не сбалансированы по нагрузке. Еще более важным является то обстоятельство, что результат вычислений отличается от ранее полученного. Это объясняется *конкуренцией потоков* за доступ к элементам массива f, поскольку теперь каждый элемент изменяется на различных итерациях внешнего цикла (которые могут выполняться параллельно на разных потоках). При этом возможна

ситуация, когда значение некоторого элемента массива f будет одновременно считано и увеличено разными потоками; в результате, разумеется, одна из добавленных сил будет потеряна.

Для исправления отмеченного недочета достаточно защитить операторы изменения элементов массива f с помощью критической секции:

```
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;
            }
        }
}
```

К сожалению, полученный вариант окажется крайне неэффективным, поскольку один из потоков регулярно будет приостанавливать свою работу в ожидании освобождения критической секции:

```
Parallel (--, critical)       Time: 7141  p0: -285.496803732846, 7.014089107234
```

Впрочем, сделанная модификация достигает своей цели: теперь результат вычислений не зависит от порядка доступа потоков к элементам массива f.

Для того чтобы ускорить вычисления и при этом не допускать конкуренции потоков, достаточно использовать вспомогательные массивы «добавочных сил» tf, связав с каждым потоком свой массив добавок. В результате при вычислении добавок не будет возникать конкуренции. При этом дополнительно придется предусмотреть цикл, в котором элементы массивов tf будут добавляться к соответствующим элементам массива f:

```
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;
        }
}
```

Интересно отметить, что данный вариант будет выполняться быстрее не только варианта с критической секцией, но и первоначального варианта (при котором возникала конкуренция потоков):

```
Parallel (--, add array)      Time: 1858  p0: -285.496803732846, 7.014089107234
```

Варианты распараллеливания эффективного алгоритма с балансировкой нагрузки

Естественно ожидать, что если в последнем варианте параллельной программы обеспечить сбалансированность нагрузки для потоков, скорость программы еще более увеличится.

Простейшим способом обеспечить более равномерную нагрузку потоков является использование опции dynamic:

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

Ниже приведены результаты расчетов для разных значений параметра block (размера «порции» итераций, которая назначается каждому потоку):

```
Parallel (--, --, dynamic 1)  Time: 2043  p0: -285.496803732846, 7.014089107234
Parallel (--, --, dynamic 25) Time: 1536  p0: -285.496803732846, 7.014089107234
```

Таким образом, скорость программы повышается в случае, если каждому потоку назначается достаточно большая порция итераций.

Другим способом обеспечения сбалансированности нагрузки является распределение итераций по «полосам»: потоку с номером k назначаются итерации с номерами k, k + Nthr, k + 2*Ntrh, …, где Nthr обозначает общее число потоков (потоки, как и итерации, нумеруются от нуля). При этом итерации цикла распределяются по потокам следующим образом:

0, 1, 2, …, Nthr–1, 0, 1, 2, …, Nthr–1, 0, 1, 2, …, Nthr–1, … .

Приведем вариант функции CaclForces, в котором используется распределение по полосам:

```
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;
        }
}
```

Данный вариант выполняется быстрее, чем первоначальный вариант без балансировки нагрузки, однако он будет проигрывать варианту с динамическим распределением потоков:

```
Parallel (--, --, stripes)    Time: 1825  p0: -285.496803732846, 7.014089107234
```

Наиболее оптимальная балансировка нагрузки достигается при распределении потоков по «обратным полосам». В этом случае итерации распределяются по потокам в «зеркальном» порядке:

0, 1, 2, …, Nthr–1, Nthr–1, …, 2, 1, 0, 0, 1, 2, …, Nthr–1, Nthr–1, …, 2, 1, 0, … .

Приведем вариант функции CaclForces, в котором используется распределение по обратным полосам:

```
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;
        }
}
```

Этот вариант выполняется быстрее, чем вариант с разбиением по «обычным» полосам, но по-прежнему немного медленнее, чем вариант с динамическим распределением потоков:

```
Parallel (--, --, b-stripes)  Time: 1695  p0: -285.496803732846, 7.014089107234
```

Для более наглядного сравнения скорости выполнения различных вариантов алгоритма приведем их в общем списке:

```
N = 800,  Niter = 100,  Nthr = 2
Non-Parallel (N*N)            Time: 8358  p0: -285.496803732846, 7.014089107234
Non-Parallel (N*(N-1)/2)      Time: 5024  p0: -285.496803732846, 7.014089107234
Parallel (N*N)                Time: 3373  p0: -285.496803732846, 7.014089107234
Parallel (N*(N-1)/2)          Time: 2887  p0: -285.496799994540, 7.014052806288
Parallel (--, critical)       Time: 7141  p0: -285.496803732846, 7.014089107234
Parallel (--, add array)      Time: 1858  p0: -285.496803732846, 7.014089107234
Parallel (--, --, dynamic 1)  Time: 2043  p0: -285.496803732846, 7.014089107234
Parallel (--, --, dynamic 25) Time: 1536  p0: -285.496803732846, 7.014089107234
Parallel (--, --, stripes)    Time: 1825  p0: -285.496803732846, 7.014089107234
Parallel (--, --, b-stripes)  Time: 1695  p0: -285.496803732846, 7.014089107234
```

Параллельные алгоритмы решения задачи n тел, использующие передачу сообщений

При программной реализации параллельного алгоритма решения задачи *n* тел, основанного на передаче сообщений, можно использовать любую из трех основных моделей взаимодействия процессов, применяемых при синхронном параллельном программировании:

- *модель «управляющий–рабочие»*, связанная с парадигмой *портфеля задач*;
- *модель пульсации*, при которой группы взаимодействующих процессов попеременно находятся в роли отправителя и получателя;
- *модель конвейера*, в которой сообщения циркулируют по цепочке процессов, причем каждый процесс всегда получает данные от предшествующего процесса и передает данные следующему.

В дальнейшем предполагаем, что число тел N пропорционально числу рабочих процессов Wproc; при этом с каждым процессом будет связан блок из N/Wproc тел. В моделях пульсации и конвейера общее число процессов равно числу рабочих процессов; в модели «управляющий–рабочие» общее число процессов равно Wproc + 1 (Wproc рабочих процессов + управляющий процесс).

Информация о телах включает два набора данных: постоянный набор масс тел и изменяющийся набор положений и скоростей тел. В то время как положения и скорости приходится пересылать между различными процессами при любой реализации параллельного алгоритма, массы тел целесообразно хранить в каждом процессе в полном объеме. Поэтому во всех описываемых далее алгоритмах предполагается, что для хранения массы тел используется отдельный массив m размера N, а для хранения других характеристик тел (положение, скорость, действующая на тело сила) используются массивы соответствующих структур, аналогичных структурам Particle и Force, описанным выше. При этом размеры этих массивов структур будут, как правило, меньше N.

Модель «управляющий–рабочие»

Основная проблема, возникающая при распараллеливании алгоритма решения задачи *n* тел, обусловлена тем, что в эффективной непараллельной реализации этого алгоритма внешний цикл, связанный с нахождением сил, выполняет различное число вычислений на разных итерациях. Таким образом, если просто разбить исходный набор тел на блоки равного размера и обрабатывать каждый блок в отдельном процессе, то нагрузка на процессы окажется неравномерной.

В модели «управляющий–рабочие» процесс-управляющий вначале формирует портфель задач, каждая из которых определяется парой номеров блоков (i, j), Задача (i, j) состоит в вычислении для всех тел, входящих в блоки i и j, значений сил взаимодействия между ними. Понятно, что достаточно рассматривать блоки, для которых $i \le j$. При нумерации блоков от 0 до Wproc–1 получаем следующие пары:

(0, 0), (0, 1), …, (0, Wproc–1), (1, 1), (1, 2), …, (1, Wproc–1), … , (Wproc–1, Wproc–1).

Таким образом, общее количество задач равно $Wproc \cdot (Wproc + 1)/2$.

Использование портфеля задач обеспечивает хорошую сбалансированность для рабочих процессов, однако не позволяет заранее определить, какие именно блоки тел будут обрабатываться каждым процессом. Поэтому в данной модели имеет смысл хранить в каждом процессе информацию о текущем состоянии всех тел.

Итак, в каждом рабочем процессе хранится информация о массе всех тел (double m[N]), о текущем состоянии всех тел (Particle p[N]) и о силах, действующих на тела (Force fTotal[N]). Кроме того, каждый процесс должен хранить еще один массив типа Force (Force f[N]), в котором будет храниться та часть суммарной силы, действующей на каждое тело, которая вычислена непосредственно в этом процессе.

В этом случае управляющий процесс может вообще не хранить данные о взаимодействующих телах, обмениваясь с рабочими процессами только парами чисел — номеров блоков.

При взаимодействии управляющего и рабочего процессов каждый освободившийся рабочий процесс посылает управляющему сообщение-запрос о новой паре блоков для обработки, а управляющий посылает этому рабочему процессу либо очередную пару блоков из портфеля задач, либо, если портфель пуст, специальное уведомление о том, что задачи закончились (в качестве такого уведомления можно, например, посылать пару (–1, –1)).

Чтобы не использовать большого количества проверок при работе с портфелем задач, в него целесообразно сразу занести не только «настоящие» пары блоков (в количестве Wproc·(Wproc + 1)/2), но и уведомления об исчерпании задач (в количестве Wproc). Данный набор блоков оформляется в виде массива и заполняется один раз — в начале работы управляющего процесса. Все последующие действия повторяются столько раз, каково число итераций алгоритма Niter. В этом цикле запускается цикл по задачам из портфеля, в котором управляющий процесс ожидает от любого из рабочих процессов уведомления о готовности принять очередную задачу, после чего посылает этому процессу задачу с очередным номером (при этом можно гарантировать, что в конце концов каждый рабочий процесс получит «пустую» задачу (–1, –1)). Для подобного обмена сообщениями достаточно использовать функции взаимодействия типа «точка–точка».

Теперь опишем действия рабочих процессов. Вначале в каждом процессе инициализируются характеристики всех тел. Все последующие действия повторяются столько раз, каково число итераций алгоритма Niter.

Пока в ответ на запрос очередной задачи процесс не получит «пустую» задачу, выполняются действия по вычислению сил взаимодействия между телами, входящими в блоки (i, j) очередной полученной задачи (вычисленные силы помещаются в массив f). При этом следует предусмотреть два варианта функции вычисления сил: один для обработки блоков вида (i, i), а другой — для блоков (i, j) при $i \neq j$. После получения «пустой» задачи процесс выходит из цикла по обработке задач. После того как все рабочие процессы выйдут из цикла обработки задач, необходимо объединить для каждого тела все действующие на него силы, вычисленные в каждом из рабочих процессов. Для этого проще всего использовать коллективную операцию редукции MPI_Allreduce, которая обеспечит рассылку полученных суммарных значений во все рабочие процессы. После этого каждый рабочий процесс должен пересчитать состояния (т. е. положения и скорости) тех тел, которые входят в соответствующий ему блок (естественно связать с рабочим процессом ранга i блок тел с тем же номером). При этом следует также обнулить значения элементов массива f (элементы массива fTotal обнулять не требуется). Наконец, после пересчета состояний тел каждый процесс должен переслать новые состояния «своих» тел всем процессам. Для этого удобно использовать коллективную операцию MPI_Bcast (необходимо обеспечить, чтобы каждый рабочий процесс выполнял пересылку только той части массива состояний p, которая была пересчитана этим процессом).

Замечание. Для того чтобы коллективные операции выполнялись только для рабочих процессов, в начале программы необходимо определить новый коммуникатор, включающий только рабочие процессы, и использовать этот коммуникатор в коллективных операциях.

Приведем два варианта результатов, выведенных программой при обработке набора исходных данных, описанного ранее. Программа была реализована в виде задания MPIDebug9; процесс ранга 8 использовался в качестве управляющего. Вывод выполнялся с помощью функций Show и ShowLine. Помимо общего времени расчета для каждого процесса выводилась информация о числе пар тел, для которых вычислялись силы гравитационного взаимодействия. Кроме того, в целях проверки правильности алгоритма выводились финальные позиции для первого и последнего тела из исходного набора. Расчеты показывают не слишком хорошую сбалансированность процессов по числу вычисленных сил, но при этом высокую скорость, сравнимую с лучшей скоростью, достигнутой в параллельной программе с общей памятью. В

силу особенностей алгоритма при различных запусках число сил, найденных в каждом процессе, различается.

```
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
```

Модель пульсации

В модели пульсации все процессы являются равноправными, т. е. рабочими. В этой модели каждый процесс «отвечает» за вычисление сил, связанных с телами своего блока. При этом, разумеется, процесс должен получать состояния всех тел, которые взаимодействуют с «его» телами.

Напомним, что в эффективном варианте непараллельного алгоритма для каждого тела вычислялись силы его взаимодействия с телами, имеющими бóльшие порядковые номера; при этом найденные силы сразу добавлялись к суммарной силе для каждого из двух взаимодействующих тел.

С учетом этого обстоятельства любому процессу в алгоритме пульсации достаточно получать состояния тел только от тех процессов, ранг которых превосходит ранг данного процесса. Однако при этом каждый процесс должен «возвращать» тем процессам, которые передали ему состояния, соответствующие значения вычисленных сил. Таким образом, часть суммарной силы, действующей на каждое тело из блока, связанного с некоторым процессом, вычисляется в самом процессе, а часть пересылается этому процессу от процессов меньшего ранга.

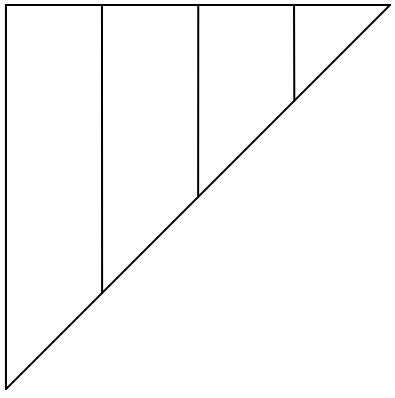


Рис. 1. Блоки одинакового размера

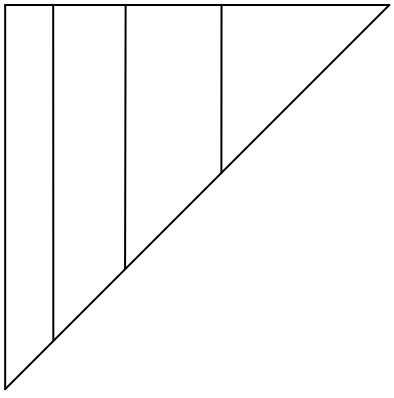


Рис. 2. Блоки разного размера

Понятно, что при равномерном распределении тел по блокам работа процессов не будет сбалансирована: процесс ранга 0 должен получить от всех других процессов состояния их тел и вычислить все силы, действующие на свои тела (попутно переслав соответствующие силы всем процессам более высокого ранга), а процесс ранга Wproc – 1 должен вычислить лишь силы взаимодействия между своими телами, переслать состояния своих тел всем процессам меньшего ранга и получить от них значения вычисленных в этих процессах сил. Используя схему, приведенную на рис. 1, можно сказать, что каждый процесс должен обработать свой столбец данных, переслав вычисленные силы «вверх» по своему столбцу, при этом «площадь» столбца соответствует объему вычислений, которые должен произвести данный процесс.

Для более равномерной балансировки нагрузки достаточно использовать *блоки разного размера*, отведя для процесса ранга 0 наименьшее число тел, а для процесса ранга Wproc – 1 — наибольшее (см. рис. 2).

Таким образом, в данном методе потребуется предварительно сформировать массивы blockSize и blockStart, размер которых равен числу процессов; при этом для процесса ранга i значение blockSize[i] определяет число тел, входящее в данный блок, а значение blockStart[i] — индекс начального тела, входящего в данный блок.

Для формирования этих массивов можно использовать следующий алгоритм: вначале вычисляется среднее количество avrSize взаимодействующих пар тел для каждого блока, равное $N \cdot (N - 1)/2Wproc$, затем, начиная от правой верхней пары (N–1, N–1) суммируются пары тел расположенные в столбцах, до тех пор, пока количество пар не превысит avrSize, после этого все тела, соответствующие просмотренным столбцам, связываются с очередным блоком, и процесс повторяется. Указанные вычисления следует провести для каждого рабочего процесса. В ходе этих вычислений необходимо также определить размер максимального блока maxBlockSize, поскольку именно такой размер надо предусматривать для массивов, используемых для пересылки данных между процессами.

Кроме формирования массивов blockSize и blockStart, в начале работы каждого процесса требуется инициализировать состояния тех тел, которые относятся к блоку, связанному с этим процессом. Напомним, что для уменьшения числа пересылок данных целесообразно хранить в каждом процессе массы всех тел m[N]. Память для прочих массивов следует выделять динамически: для процесса ранга i массивы p и f (типа Particle и Force соответственно) должны иметь размер blockSize[i], а массивы tp и tf, предназначенные для получения данных (состояний и сил) из других процессов, должны иметь размер maxBlockSize.

Теперь кратко опишем последующие действия, выполняемые каждым рабочим процессом. Все эти действия повторяются столько раз, каково число итераций алгоритма Niter.

Вначале каждый процесс выполняет пересылку данных из своего массива p всем процессам меньшего ранга. Для этой пересылки можно использовать асинхронную команду MPI_Isend, позволяющую продолжать выполнение программы, не ожидая окончания действий по пересылке сообщения. Затем процесс вычисляет силы, действующие между телами своего блока.

После этого для всех процессов, ранг которых превосходит ранг данного процесса, выполняются следующие три действия: процесс получает от процесса большего ранга состояния их тел, вычисляет силы, действующие между своими телами и телами, полученными от этого процесса (при этом силы для своих тел накапливаются в массиве f, а силы для тел другого процесса — в массиве tf, который в начале процедуры вычисления сил должен быть обнулен), и пересылает обратно массив сил tf.

Наконец, процесс получает от всех процессов меньшего ранга вычисленные в них массивы сил и добавляет эти силы (помещенные в массив tf) к массиву f.

Последнее действие состоит в пересчете состояния тел с использованием сил f и обнулении массива f.

Приведем вариант результатов, выведенных программой при обработке набора исходных данных, описанного ранее. Программа была реализована в виде задания MPIDebug8; все процессы являлись рабочими. Вывод выполнялся с помощью функций Show и ShowLine. Помимо общего времени расчета для каждого процесса выводилась информация о числе пар тел, для которых вычислялись силы гравитационного взаимодействия. Кроме того, в целях проверки правильности алгоритма выводились финальные позиции для первого и последнего тела из исходного набора. Расчеты показывают хорошую сбалансированность процессов по числу вычисленных сил. В то же время по скорости выполнения данный вариант уступает варианту «управляющий–рабочие».

```
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
```

Модель конвейера

В модели конвейера, как и в модели пульсации, все процессы являются равноправными, т. е. рабочими. Общим свойством этих моделей является и то, что каждый процесс «отвечает» за вычисление сил, связанных с телами своего блока. Однако в модели конвейера процесс получает данные о состоянии тел, связанных с другими процессами, не напрямую от них, а по цепочке, всегда принимая данные от своего «левого» соседа и посылая данные своему «правому» соседу. Для правильного определения рангов левого и правого соседа для процесса ранга i следует использовать формулы (i + Wproc – 1) % Wproc и (i + 1) % Wproc соответственно.

В простейшем варианте модели конвейера на каждой из Niter итераций алгоритма каждый процесс вначале пересылает тела из своего блока в правый процесс, затем вычисляет силы взаимодействия между телами своего блока, после чего в цикле, повторяющемся Wproc – 1 раз, получает очередной блок тел от своего левого соседа и вычисляет силы между своими телами и телами из полученного блока. После вычисления всех сил, действующих на тела из своего блока, процесс пересчитывает состояния своих тел.

К сожалению, в описанном простейшем варианте силы между телами из разных блоков вычисляются *дважды*: в каждом из процессов, связанных с этими блоками. В этом отношении данный вариант аналогичен первому из рассмотренных ранее непараллельных алгоритмов.

Чтобы не дублировать вычисление сил, можно модифицировать алгоритм следующим образом: каждый процесс пересылает своему правому соседу не только набор состояний очередного блока тел, но и *набор сил*, действующих на эти тела (вначале этот набор содержит нулевые значения). Получив эти данные от левого соседа, процесс вычисляет силы взаимодействия только для тех своих тел, «истинные» номера которых (т. е. номера тел в исходном наборе до разделения его на блоки) *меньше* истинных номеров взаимодействующих с ними тел, полученных из другого процесса. Кроме того, процесс добавляет вычисленные «парные» силы к полученному набору сил, после чего пересылает данные дальше. После того как отправленный блок данных совершит «полный оборот», он вернется к пославшему его процессу, и при этом будет содержать ту часть сил, действующих на тела данного процесса, которые были вычислены в других процессах. Останется добавить эту часть к силам, вычисленным в самом процессе, и пересчитать состояния своих тел. Кроме пересчета состояния на завершающем этапе каждой итерации необходимо обнулять значения сил.

Указанная модификация исходного неэффективного алгоритма приводит к несбалансированности процессов: процессы с меньшими рангами будут вычислять большее количество сил. В данной модели, вместо того чтобы вводить в рассмотрение блоки разного размера (как в рассмотренной ранее модели пульсации), можно применить другой прием: распределения тел *по полосам* при сохранении одинакового размера blockSize для всех блоков (blockSize = N / Wproc). Распределение по полосам в данном случае означает, что в каждый блок будет входить по одному телу из каждой полосы, на которые разбивается весь набор тел. Например, в случае разбиения на простые полосы в i-й блок будут входить тела с номерами i, i + Wproc, i + 2·Wproc, … .

Подобный подход потребует модификации процедуры инициализации состояний тел, связанных с i-м процессом, однако обеспечит хорошую сбалансированность процессов. При определении того, какие силы требуется вычислять в данном процессе, а какие силы оставить для вычисления «парному» процессу, достаточно, как и в предыдущей модификации алгоритма, сравнивать «истинные» номера обрабатываемых тел и вычислять силы только в том случае, когда номер тела, относящегося к текущему процессу, *меньше* номера тела, полученного от другого процесса (вид неравенства роли не играет: можно использовать и вариант алгоритма, в котором силы вычисляются в случае, если номер тела из текущего процесса *больше* номера тела, полученного от другого процесса).

Достичь идеальной сбалансированности, при которой каждый процесс вычисляет одно и то же число сил, можно с помощью разбиения тел на *обратные полосы* (данное разбиение описано в разделе, посвященном параллельным алгоритмам с общей памятью).

При подобной модификации исходного алгоритма целесообразно определить новую структуру данных ParticleForce, включающую как данные о состоянии некоторого тела (положение x, y и скорость vx, vy), так и данные о действующих на него силах (fx, fy). В каждом процессе надо описать массивы p и tp типа ParticleForce, каждый из которых имеет размер blockSize. В массиве p хранится информация о «собственных» телах процесса (в том числе и значения сил, вычисленные в этом процессе), а массив tp используется для получения и последующей пересылки данных о состоянии тел из других процессов, а после завершения конвейерных вычислений — для получения данных о добавочных силах для «собственных» тел процесса, вычисленных в других процессах.

Приведем варианты результатов, выведенных программой при обработке набора исходных данных, описанного ранее. Программа была реализована в виде задания MPIDebug8; все процессы являлись рабочими. Вывод выполнялся с помощью функций Show и ShowLine. Помимо общего времени расчета для каждого процесса выводилась информация о числе пар тел, для которых вычислялись силы гравитационного взаимодействия. Кроме того, в целях проверки правильности алгоритма выводились финальные позиции для первого и последнего тела из исходного набора.

Первый вариант соответствует алгоритму без балансировки процессов (в каждом процессе силы вычислялись при обработке пары тел, в которой номер «своего» тела был *меньше* номера тела, полученного из другого процесса). Несмотря на явную несбалансированность по количеству найденных сил, время работы процессов сравнимо с временем, полученным для модели пульсации.

```
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
```

Второй вариант соответствует алгоритму с балансировкой на основе разбиения на полосы. Алгоритм оказался почти сбалансированным по числу вычисляемых сил, однако время работы каждого процесса увеличилось (за счет дополнительных расчетов, связанных с определением «истинных» номеров взаимодействующих тел).

```
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
```

Третий вариант соответствует алгоритму с балансировкой на основе разбиения на обратные полосы. Алгоритм полностью сбалансирован по числу вычисляемых сил и дает немного меньшее время по сравнению с предыдущим вариантом балансировки, однако полученное время по-прежнему уступает варианту без какой-либо балансировки. Следует обратить внимание на то, что в силу особенностей разбиения на обратные полосы как первое, так и последнее тело из исходного набора обрабатывается в данном случае в процессе ранга 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
```

Задания для самостоятельного выполнения

MPIGravit1. Реализовать метод «управляющий–рабочие» в виде задания MPIDebug9 электронного задачника Programming Taskbook for MPI и протестировать его на описанном ранее наборе исходных данных, выводя те же сведения, что и в листинге 3.10.

MPIGravit2. Реализовать метод пульсации в виде задания MPIDebug8 электронного задачника Programming Taskbook for MPI и протестировать его на описанном ранее наборе исходных данных, выводя те же сведения, что и в листинге 3.11.

MPIGravit3. Реализовать метод конвейера с разбиением на полосы в виде задания MPIDebug8 электронного задачника Programming Taskbook for MPI и протестировать его на описанном ранее наборе исходных данных, выводя те же сведения, что и в листинге 3.13.

MPIGravit4. Реализовать метод конвейера с разбиением на обратные полосы в виде задания MPIDebug8 электронного задачника Programming Taskbook for MPI и протестировать его на описанном ранее наборе исходных данных, выводя те же сведения, что и в листинге 3.14.