Лекция 9
Стандарт OpenMP: гравитационная задача N тел

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

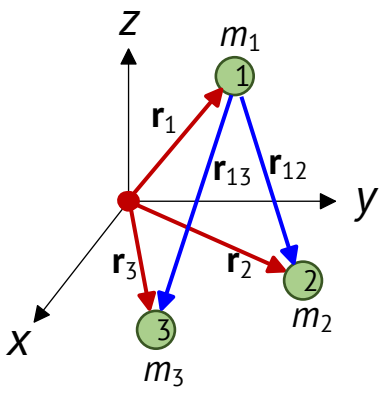
CC-BY

Гравитационная задача N тел (N-body)

- **Гравитационная задача N тел (n-body simulation)** – задача небесной механики и гравитационной динамики Ньютона
- **Имеется** N материальных точек с заданными массами m_i
- Попарное взаимодействие точек подчинено закону тяготения Ньютона, а силы гравитации аддитивны
- Известны начальные на момент времени $t = 0$ положения и скорости каждой точки.
- **Требуется** найти положения точек для всех последующих моментов времени.

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i, \qquad i = 1,2,\ldots,N,$$

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{F}_i, \qquad i = 1,2,\ldots,N,$$



- Сила гравитации между телами i и j

$$F = \frac{G m_i m_j}{r^2}$$

- $G = 6{,}67430(15)\cdot 10^{-11}$ м^3·с$^{-2}$·кг$^{-1}$ – гравитационная постоянная,
- r – расстояние между телами

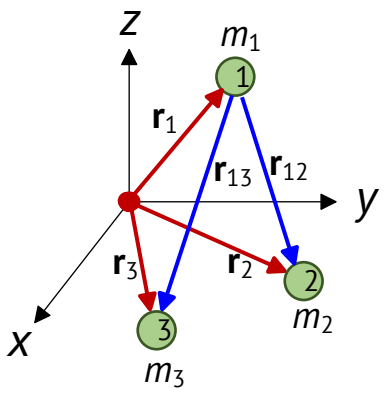
- На данный момент в общем виде задача для $N > 3$ может быть решена только численно

Численное решение задачи N тел (N-body)

- **Прямое моделирование задачи N-тел – $O(N^2)$**

  Цикл по времени

  1. Вычисление сил, действующих на каждое тело – $O(N^2)$
     (вычисление сил => вычисление ускорением => вычисление скорости => вычисление новых положений тел)
  2. Перемещение тел – $O(N)$



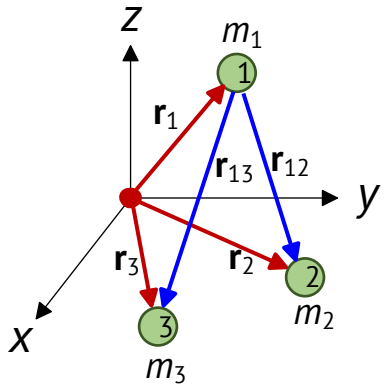
- **Приближенные методы**
  - **Метод Барнса-Хата** (Treecode, Barnes–Hut simulation) – $O(N\log N)$
  - **Fast multipole methods –** $O(N)$
  - Particle mesh methods
  - P3M and PM-tree methods
  - Mean field methods

Прямое моделирование задачи N-тел – $O(N^2)$

Цикл по модельному времени: пока $t < t_{end}$

1. **Вычисление сил, действующих на каждое тело**
   - Направление силы, действующей на тело *i* со стороны тела *j*, указывает от *i* в сторону *j* (вектор $\mathbf{r}_{ij}$)
   - Длина вектора $\mathbf{r}_{ij}$ – расстояние между точками $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2}$
   - Сила, действующая на тело *i*, – сумма действия сил со стороны всех тел

$$\mathbf{F}_i = \sum_{j\neq i}^{N} \frac{G m_i m_j \mathbf{r}_{ij}}{\left(\sqrt{(x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2}\right)^2}$$



2. **Перемещение тел**
   - Изменение скорости за интервал времени Δt: $d\mathbf{v}_i = \frac{\mathbf{F}_i}{m_i}\Delta t$
   - Изменение положения тела за время Δt (схема leapfrog): $d\mathbf{r}_i = \left(\mathbf{v}_i + \frac{d\mathbf{v}_i}{2}\right)\Delta t$
3. **Переход к следующем шагу по времени** $t = t + \Delta t$

Последовательная программа

```
struct particle { float x, y, z; };

int main(int argc, char *argv[])
{
    double ttotal, tinit = 0, tforces = 0, tmove = 0;
    ttotal = wtime();
    int n = (argc > 1) ? atoi(argv[1]) : 10;
    char *filename = (argc > 2) ? argv[2] : NULL;

    tinit = -wtime();
    struct particle *p = malloc(sizeof(*p) * n);  // Положение частиц (x, y, z)
    struct particle *f = malloc(sizeof(*f) * n);  // Сила, действующая на каждую частицу (x, y, z)
    struct particle *v = malloc(sizeof(*v) * n);  // Скорость частицы (x, y, z)
    float *m = malloc(sizeof(*m) * n);            // Масса частицы
    for (int i = 0; i < n; i++) {
        p[i].x = rand() / (float)RAND_MAX - 0.5;
        p[i].y = rand() / (float)RAND_MAX - 0.5;
        p[i].z = rand() / (float)RAND_MAX - 0.5;
        v[i].x = rand() / (float)RAND_MAX - 0.5;
        v[i].y = rand() / (float)RAND_MAX - 0.5;
        v[i].z = rand() / (float)RAND_MAX - 0.5;
        m[i] = rand() / (float)RAND_MAX * 10 + 0.01;
        f[i].x = f[i].y = f[i].z = 0;
    }
    tinit += wtime();
```

Последовательная программа

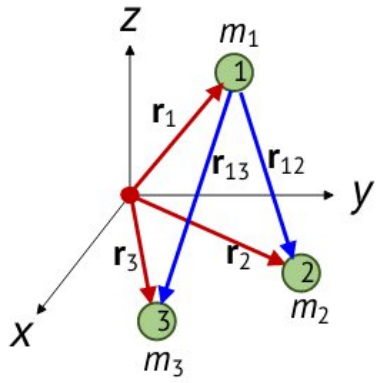
```
double dt = 1e-5;
for (double t = 0; t <= 1; t += dt) {      // Цикл по времени (модельному)
    tforces -= wtime();
    calculate_forces(p, f, m, n);          // Вычисление сил - O(N^2)
    tforces += wtime();

    tmove -= wtime();
    move_particles(p, f, v, m, n, dt);     // Перемещение тел O(N)
    tmove += wtime();
}
ttotal = wtime() - ttotal;

printf("# NBody (n=%d)\n", n);
printf("# Elapsed time (sec): ttotal %.6f, tinit %.6f, tforces %.6f, tmove %.6f\n",
       ttotal, tinit, tforces, tmove);
```

Последовательная программа

```
    if (filename) {
        FILE *fout = fopen(filename, "w");
        if (!fout) {
            fprintf(stderr, "Can't save file\n");
            exit(EXIT_FAILURE);
        }
        for (int i = 0; i < n; i++) {
            fprintf(fout, "%15f %15f %15f\n", p[i].x, p[i].y, p[i].z);
        }
        fclose(fout);
    }

    free(m);
    free(v);
    free(f);
    free(p);
    return 0;
}
```

Последовательная программа

```
const float G = 6.67e-11;

void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // Вычисление силы, действующей на тело i со стороны j
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) + powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            // Сумма сил, действующих на тело i
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;
            // Сумма сил, действующих на тело j (симметричность)
            f[j].x -= mag * dir.x / dist;
            f[j].y -= mag * dir.y / dist;
            f[j].z -= mag * dir.z / dist;
        }
    }
}
```



$$\mathbf{F}_i = \sum_{j \neq i}^{N} \frac{G m_i m_j \mathbf{r}_{ij}}{\left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}\right)^2}$$

Последовательная программа

```
void move_particles(struct particle *p, struct particle *f, struct particle *v, float *m, int n,
                    double dt)
{
    for (int i = 0; i < n; i++) {
        struct particle dv = {
            .x = f[i].x / m[i] * dt,
            .y = f[i].y / m[i] * dt,
            .z = f[i].z / m[i] * dt,
        };
        struct particle dp = {
            .x = (v[i].x + dv.x / 2) * dt,
            .y = (v[i].y + dv.y / 2) * dt,
            .z = (v[i].z + dv.z / 2) * dt,
        };
        v[i].x += dv.x;
        v[i].y += dv.y;
        v[i].z += dv.z;
        p[i].x += dp.x;
        p[i].y += dp.y;
        p[i].z += dp.z;
        f[i].x = f[i].y = f[i].z = 0;
    }
}
```

$$d\mathbf{v}_i = \frac{\mathbf{F}_i}{m_i}\Delta t$$

$$d\mathbf{r}_i = \left(\mathbf{v}_i + \frac{d\mathbf{v}_i}{2}\right)\Delta t$$

Профилирование последовательной программы

```
$ perf record ./nbody 100
# NBody (n=100)
# Elapsed time (sec): ttotal 9.696869, tinit 0.000043, tforces 9.559545, tmove 0.130254

$ perf report
```

```
Samples: 38K of event 'cycles:u', Event count (approx.): 25946061841
Overhead  Command  Shared Object       Symbol
  98.50%  nbody    nbody              [.] calculate_forces
   1.33%  nbody    nbody              [.] move_particles
   0.07%  nbody    [vdso]             [.] __vdso_gettimeofday
   0.05%  nbody    [vdso]             [.] 0x0000000000000949
   0.04%  nbody    nbody              [.] main
   0.02%  nbody    [kernel.kallsyms]  [k] __irqentry_text_start
   0.00%  nbody    [vdso]             [.] 0x0000000000000947
   0.00%  nbody    libc-2.24.so       [.] _dl_addr
   0.00%  nbody    nbody              [.] gettimeofday@plt
   0.00%  nbody    ld-2.24.so         [.] dl_main
   0.00%  nbody    [kernel.kallsyms]  [k] page_fault
   0.00%  nbody    ld-2.24.so         [.] _start
```

Профилирование последовательной программы (GNU gprof)

```
$ gcc nbody.c -o nbody -pg
$ ./nbody 100
# NBody (n=100)
# Elapsed time (sec): ttotal 9.584010, tinit 0.000045, tforces 9.449003, tmove 0.128511

$ gprof ./nbody

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 98.99      9.40     9.40                             calculate_forces
  0.95      9.49     0.09                             move_particle
```

Параллельная версия 1

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;

            f[j].x -= mag * dir.x / dist;
            f[j].y -= mag * dir.y / dist;
            f[j].z -= mag * dir.z / dist;
        }
    }
}
```

Распределим по *P* потокам вычисление элементов вектора сил f[N]

- Поток 0: f[0], f[1], ..., f[N/P - 1]
- Поток 1: f[N/P], f[N/P + 1], ..., f[2N/P - 1]
- ...
- Поток P-1: f[(P-1)N/P], f[(P-1)N/P + 1], ..., f[N - 2]

Параллельная версия 1

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;

            f[j].x -= mag * dir.x / dist;
            f[j].y -= mag * dir.y / dist;
            f[j].z -= mag * dir.z / dist;
        }
    }
}
```

Распределим по *P* потокам вычисление элементов вектора сил f[N]

- Поток 0: f[0], f[1], ..., f[N/P - 1]
- Поток 1: f[N/P], f[N/P + 1], ..., f[2N/P - 1]
- ...
- Поток P-1: f[(P-1)N/P], f[(P-1)N/P + 1], ..., f[N - 2]

Возможно ли возникновение гонки данных (data race)?

- Локальные переменные потоков: i, j, dist, mag, dir
- Потоки читают и перезаписывают элементы: f[i], f[j]

Возможна ли ситуация совпадения i и/или j в разных потоках (гонка за f[i], f[j])?

Параллельная версия 1

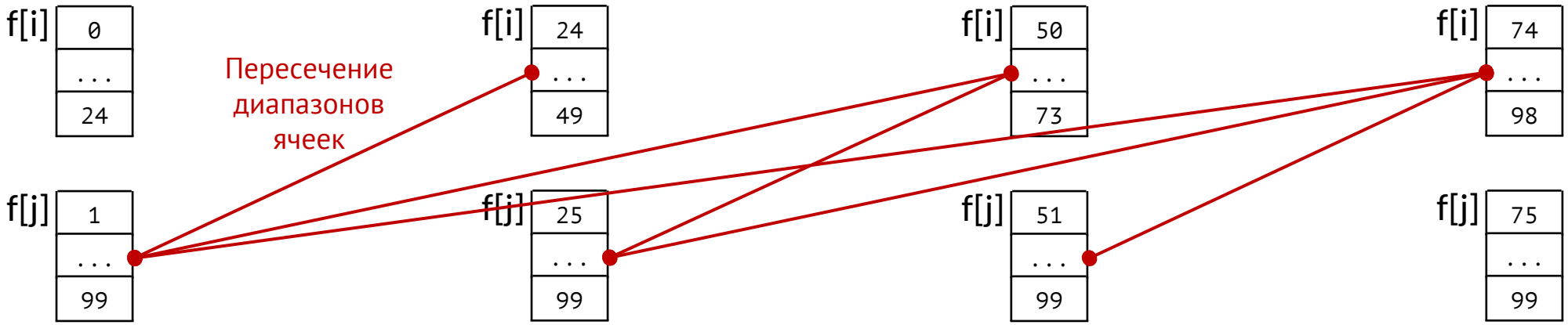
Пусть N = 100, OMP_NUM_THREADS=4

```
// Thread 0
for (i = 0; i < 24; i++)
 for (j = 1; j < 99; j++)
    f[i] +=
    f[j] +=
```

```
// Thread 1
for (i = 24; i < 49; i++)
 for (j = 25; j < 99; j++)
    f[i] +=
    f[j] +=
```

```
// Thread 2
for (i = 50; i < 73; i++)
 for (j = 51; j < 99; j++)
    f[i] +=
    f[j] +=
```

```
// Thread 3
for (i = 74; i < 98; i++)
 for (j = 75; j < 99; j++)
    f[i] +=
    f[j] +=
```



Состояние гонки данных (data race)

Параллельная версия 1

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;

            f[j].x -= mag * dir.x / dist;
            f[j].y -= mag * dir.y / dist;
            f[j].z -= mag * dir.z / dist;
        }
    }
}
```

Распределим по *P* потокам вычисление элементов вектора сил f[N]

- Поток 0: f[0], f[1], ..., f[N/P - 1]
- Поток 1: f[N/P], f[N/P + 1], ..., f[2N/P - 1]
- ...
- Поток P-1: f[(P-1)N/P], f[(P-1)N/P + 1], ..., f[N - 2]

Ошибка!
Потоки конкурируют за чтение и обновление элементов f[i], f[j] — состояние гонки данных (data race)

Результаты вычислений (силы) будут меняться от запуска к запуску

Параллельная версия 1: проблемы

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            #pragma omp critical
            {
                f[i].x += mag * dir.x / dist;
                f[i].y += mag * dir.y / dist;
                f[i].z += mag * dir.z / dist;
                f[j].x -= mag * dir.x / dist;
                f[j].y -= mag * dir.y / dist;
                f[j].z -= mag * dir.z / dist;
            }
        }
    }
}
```

Потоки загружены неравномерно

i = 0: (0,0), (0, 1), (0, 2), (0, 3), (0, 4),
i = 1: (1, 2), (1, 3), (1, 4),
i = 2: (2, 3), (2, 4)
i = 3: (3, 4)

Варианты потокобезопасного обновления сил

1. Использовать одну критическую секцию #pragma omp critical – за нее конкурируют все потоки
2. Использовать атомарную операцию для обновления каждой компоненты вектора сил
3. Использовать отдельную блокировку для каждой частицы (массив) – снижает конкуренцию потоков за блокировку
4. Реализовать алгоритм без использования блокировок

Параллельная версия 2: atomic

```
double dt = 1e-5;
#pragma omp parallel     // Параллельный регион активируется один раз
{
    for (double t = 0; t <= 1; t += dt) {
        calculate_forces(p, f, m, n);

        #pragma omp barrier    // Ожидание завершения расчетов f[i]
        move_particles(p, f, v, m, n, dt);

        #pragma omp barrier    // Ожидание завершения обновления p[i], f[i]
    }
}
ttotal = wtime() - ttotal;
printf("# NBody (n=%d)\n", n);
printf("# Elapsed time (sec): ttotal %.6f, tinit %.6f, tforces %.6f, tmove %.6f\n",
       ttotal, tinit, tforces, tmove);
```

Параллельная версия 2: atomic

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 4) nowait        // Циклическое распределение итераций
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) + powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));
            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,  .y = p[j].y - p[i].y,  .z = p[j].z - p[i].z
            };

            #pragma omp atomic
            f[i].x += mag * dir.x / dist;
            #pragma omp atomic
            f[i].y += mag * dir.y / dist;
            #pragma omp atomic
            f[i].z += mag * dir.z / dist;
            #pragma omp atomic
            f[j].x -= mag * dir.x / dist;
            #pragma omp atomic
            f[j].y -= mag * dir.y / dist;
            #pragma omp atomic
            f[j].z -= mag * dir.z / dist;
        }
    }
}
```

Параллельная версия 2: atomic

```
void move_particles(struct particle *p, struct particle *f, struct particle *v,
                    float *m, int n, double dt)
{
    #pragma omp for nowait
    for (int i = 0; i < n; i++) {
        struct particle dv = {
            .x = f[i].x / m[i] * dt,
            .y = f[i].y / m[i] * dt,
            .z = f[i].z / m[i] * dt,
        };
        struct particle dp = {
            .x = (v[i].x + dv.x / 2) * dt,
            .y = (v[i].y + dv.y / 2) * dt,
            .z = (v[i].z + dv.z / 2) * dt,
        };
        v[i].x += dv.x;
        v[i].y += dv.y;
        v[i].z += dv.z;
        p[i].x += dp.x;
        p[i].y += dp.y;
        p[i].z += dp.z;
        f[i].x = f[i].y = f[i].z = 0;
    }
}
```

Параллельная версия 3: N блокировок

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));

            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            #pragma omp critical
            {
                f[i].x += mag * dir.x / dist;
                f[i].y += mag * dir.y / dist;
                f[i].z += mag * dir.z / dist;
                f[j].x -= mag * dir.x / dist;
                f[j].y -= mag * dir.y / dist;
                f[j].z -= mag * dir.z / dist;
            }
        }
    }
}
```

```
// Thread 0
for (i = 0; i < 24; i++)
 for (j = 1; j < 99; j++)
   #pragma omp critical
     f[0] +=
     f[1] +=
```

```
// Thread 1
for (i = 24; i < 49; i++)
 for (j = 25; j < 99; j++)
   #pragma omp critical
     f[24] +=
     f[25] +=
```

Потоки обновляют разные ячейки — блокировка не нужна!

Избыточная блокировка (на весь массив)

мы блокируем фрагмент кода от одноврменного выполнения, без учета номеров ячеек, которые модифицируются потоками

(P потоков конкурируют за один мьютекс)

Параллельная версия 3: N блокировок

```
omp_lock_t *locks;   // Массив из N блокировок (мьютексов) — блокировка на уровне отдельных ячеек

void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 4) nowait
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) + powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));
            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            omp_set_lock(&locks[i]);
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;
            omp_unset_lock(&locks[i]);
            omp_set_lock(&locks[j]);
            f[j].x -= mag * dir.x / dist;
            f[j].y -= mag * dir.y / dist;
            f[j].z -= mag * dir.z / dist;
            omp_unset_lock(&locks[j]);
        }
    }
}
```

- Меньшее количество потоков конкурирует за одноврменный захват мьютекса (блокировки, lock, mutex)
- Если потоки модифицируют разные ячейки они не ждут освобождение мьютекса — разные блокировки, одна на каждую ячейку массива сил

Параллельная версия 3: N блокировок

```
int main(int argc, char *argv[])
{
    // ...
    locks = malloc(sizeof(omp_lock_t) * n);
    for (int i = 0; i < n; i++)
        omp_init_lock(&locks[i]);

    double dt = 1e-5;
    #pragma omp parallel
    {
        for (double t = 0; t <= 1; t += dt) {
            calculate_forces(p, f, m, n);
            #pragma omp barrier
            move_particles(p, f, v, m, n, dt);
            #pragma omp barrier
        }
    }
    // ...

    free(locks);
    // ...
}
```

Параллельная версия 4: дополнительные вычисления вместо блокировок

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 4) nowait
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j)
                continue;

            float dist = sqrtf(powf(p[i].x - p[j].x, 2) +
                               powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));
            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = {
                .x = p[j].x - p[i].x,
                .y = p[j].y - p[i].y,
                .z = p[j].z - p[i].z
            };
            f[i].x += mag * dir.x / dist;
            f[i].y += mag * dir.y / dist;
            f[i].z += mag * dir.z / dist;
        }
    }
}
```

- Необходимость использование блокировок для защиты обновления f[i], f[j]— из –за симметрии при вычислении сил
- Если отказаться от симметрии, то в потоках не будет пересечений по номерам ячеек
- Поток, отвечающий за тело i, полностью вычисляет силы действия со стороны всех тел (не используется симметричность сил между делами, вычисления дублируются)
- Нет мьютексов (ожиданий их освобождения), но появились дополнительные вычисления

Параллельная версия 5: дополнительная память вместо вычислений

```
void calculate_forces(struct particle *p, struct particle *f[], float *m, int n)
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    for (int i = 0; i < n; i++) {
        f[tid][i].x = 0;
        f[tid][i].y = 0;
        f[tid][i].z = 0;
    }
    #pragma omp for schedule(dynamic, 8)
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            float dist = sqrtf(powf(p[i].x - p[j].x, 2) + powf(p[i].y - p[j].y, 2) +
                               powf(p[i].z - p[j].z, 2));
            float mag = (G * m[i] * m[j]) / powf(dist, 2);
            struct particle dir = { .x = p[j].x - p[i].x,  .y = p[j].y - p[i].y,  .z = p[j].z - p[i].z };

            f[tid][i].x += mag * dir.x / dist;
            f[tid][i].y += mag * dir.y / dist;
            f[tid][i].z += mag * dir.z / dist;
            f[tid][j].x -= mag * dir.x / dist;
            f[tid][j].y -= mag * dir.y / dist;
            f[tid][j].z -= mag * dir.z / dist;
        }
    } // barrier
```

- Создадим для каждого потока локальную копию вектора сил, что позволит избежать гонки данных
- Вместо вектора сила используется матрица сил
- Число строк в матрице равно количеству потоков
- Дублируется память, а не вычисления

Параллельная версия 5: дополнительная память вместо вычислений

```
    #pragma omp single   // Итоговый вектор сил сформируем в первой строке - f[0][i]
    {
        for (int i = 0; i < n; i++) {
            for (int tid = 1; tid < nthreads; tid++) {
                f[0][i].x += f[tid][i].x;
                f[0][i].y += f[tid][i].y;
                f[0][i].z += f[tid][i].z;
            }
        }
    }

} // calculate_forces
```

Параллельная версия 5: дополнительная память вместо вычислений

```
void move_particles(struct particle *p, struct particle *f[], struct particle *v, float *m, int n,
                    double dt)
{
    #pragma omp for
    for (int i = 0; i < n; i++) {
        struct particle dv = {
            .x = f[0][i].x / m[i] * dt,
            .y = f[0][i].y / m[i] * dt,
            .z = f[0][i].z / m[i] * dt,
        };
        struct particle dp = {
            .x = (v[i].x + dv.x / 2) * dt,
            .y = (v[i].y + dv.y / 2) * dt,
            .z = (v[i].z + dv.z / 2) * dt,
        };
        v[i].x += dv.x;
        v[i].y += dv.y;
        v[i].z += dv.z;
        p[i].x += dp.x;
        p[i].y += dp.y;
        p[i].z += dp.z;
        // f[i].x = f[i].y = f[i].z = 0;
    }
}
```

Параллельная версия 5: дополнительная память вместо вычислений

```
int main(int argc, char *argv[])
{
    // ...

    struct particle *f[omp_get_max_threads()];
    for (int i = 0; i < omp_get_max_threads(); i++)
        f[i] = malloc(sizeof(struct particle) * n);

    // ...
    return 0;
}
```

Варианты балансировки загрузки

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 4) nowait
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // Обработка пары (i, j)
        }
    }
}
```

Треугольное пространство итераций

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for collapse(2) schedule(dynamic, 1024) nowait
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // Обработка пары (i, j)
        }
    }
}
```

Варианты балансировки загрузки

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 4) nowait
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // Обработка пары (i, j)
        }
    }
}
```

Треугольное пространство итераций

Fusing a triangular loop (слияние циклов)

```
void calculate_forces(struct particle *p, struct particle *f, float *m, int n)
{
    #pragma omp for schedule(dynamic, 1024) nowait
    for (int w = 0; w < n * (n - 1) / 2; w++) {        // Одномерный цикл по всем связям (i, j)
        int i = (-1 + sqrt(1.0 + 8.0 * w)) / 2;
        int j = w - i * (i + 1) / 2;

        // Обработка пары (i, j)
    }
}
```