Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт

Высшая школа теоретической механики и математической физики

К У Р С О В А Я Р А Б О Т А

по дисциплине «Проектирование по компьютерным технологиям в механике»

Выполнил

студент гр.5040103/20101 Фролов М. М.

Руководитель

Ле-Захаров А. А.

«___» _____ 2023 г.

Санкт-Петербург

2023

Содержание

Введение

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом и другими [1].

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров [2]. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. Параллельная программа должна эффективно использовать вычислительные мощности и коммуникационную среду. В MPI вся работа по распределению нагрузки на узлы и сеть ложатся на программиста и для максимальной производительности необходимо знать особенности конкретного кластера.

Идея распараллеливания программ состоит в том, чтобы ускорить их работу, не повлияв на точность получаемых результатов. Достигается распараллеливание путем использования двух и более процессоров/ядер в комбинации для решения одной задачи [3].

В ходе данной работы будут решены задачи: вычисления определенного интеграла, вычисления числа Пи методом Монте-Карло и решения одномерной задачи теплопроводности. А также построены зависимости времени выполнения программы и коэффициента распараллеливания от числа процессоров.

Основные функции MPI

В ходе реализации процесса распараллеливания были использованы следующие методы:

- MPI_Init - Инициализация среды MPI.
- MPI_Comm_rank - Определение номера процессора.
- MPI_Comm_size - Количество задействованных процессоров.
- MPI_Send - Функция, позволяющая отправлять полученный результат, которая на вход получает: адрес буфера, в который помещаются данные, размер буфера, тип ячейки буфера, номер процессора, с которым происходит обмен данными, идентификатор сообщения, описатель области связи.
- MPI_Recv - Функция, позволяющая принимать сообщения от других процессоров, которая на вход получает: адрес буфера, из которого берутся данные, размер буфера, тип ячейки буфера, номер процессора, с которым происходит обмен данными, идентификатор сообщения, описатель области связи, статус завершения приёма.
- MPI_Finalize - Деактивация среды MPI.

Величина эффективности определяет среднюю долю времени выполнения алгоритма, в течение которой процессоры реально задействованы для решения задачи.

1. Вычисление числа Пи

Необходимо вычислить значение числа Пи методом Монте – Карло на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 4 процессорах). Для 3200000 точек. Каждый расчёт проведен 10 раз для получения усредненного результата времени вычисления. Каждый процессор будет работать на своём участке и передавать получившееся значение в процессор с номером “0“. Для решения этой задачи было рассмотрено отношение площадей квадрата и вписанного в него круга.

$$\begin{cases} S_{\text{кв}} = d^2 \\ S_{\text{кр}} = \pi \left(\frac{d}{2}\right)^2 \end{cases}$$

Таким образом, получаем:

$$\frac{S_{\text{кр}}}{S_{\text{кв}}} = \frac{\pi}{4}$$

Так, при большом количестве точек в численном эксперименте:

$$\pi = 4\frac{N_{\text{кр}}}{N_{\text{кв}}},$$

Где $N_{\text{кр}}$ – количество точек попавших в круг, $N_{\text{кв}}$ – общее количество точек.

Код реализации приведен ниже:

```
MPI_Init(&argc, &argv);
 int rank, size;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 double start = MPI_Wtime();
 srand(time(0));
 uint64_t total_points = 10000000;
 uint64_t num_for_core = total_points / size;
 uint64_t num_points_in_circle = 0;
 for (int i = 0; i < num_for_core; i++) {
```

```
        float x = (float)rand() / RAND_MAX;
        float y = (float)rand() / RAND_MAX;
        num_points_in_circle += (x * x + y * y < 1.0);
    }
    uint64_t global_num_points = 0;
    MPI_Reduce(&num_points_in_circle, &global_num_points, 1,
MPI_INT64_T, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        float pi = 4.0f * global_num_points / (num_for_core * size);
        printf("pi = %.5f\n", pi);
        printf("The process took %f seconds to run.\n", MPI_Wtime() -
start);
    }
    MPI_Finalize();
```

Полученные результаты приведены в таблице 1.

Таблица 1. Зависимость времени вычисления от количества процессоров

Количество процессоров	Результат вычисления интеграла	Время вычисления, с
1	0.33333	1.5200
2	0.33333	1.1345
4	0.33333	0.7498
8	0.33333	0.6857
16	0.33333	0.4534
32	0.33333	0.5438

2. Вычисление интеграла

Необходимо посчитать определенный интеграл на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 4 процессорах). При 100000 разбиениях. Каждый расчёт проведен 10 раз для получения усредненного результата времени вычисления. Каждый процессор будет работать на своём участке и передавать получившееся значение в процессор с номером «0».

Функция для расчёта:

$$\int_0^1 x^2\,dx$$

Код реализации приведен ниже:

```
MPI_Init(&argc, &argv);

    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double start = MPI_Wtime();


    double left = 0.0f;

    double right = 1.0f;

    double local_left = left + rank * (right - left) / size;


    int total_points = 1000000;

    int num_for_core = total_points / size;

    double dx = (right - left) / (size * num_for_core);


    double local_integral = 0.0f;

    for (int i = 0; i < num_for_core; i++) {

        double x = local_left + i * dx;
```

```
        local_integral += f(x) * dx;
    }

    double global_integral = 0.0f;
    MPI_Reduce(&local_integral, &global_integral, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("integral = %.5f\n", global_integral);
        printf("The process took %f seconds to run.\n", MPI_Wtime() - start);
    }
    MPI_Finalize();
```

Полученные результаты приведены в таблице 2.

Таблица 2. Зависимость времени вычисления от количества процессоров

Количество процессоров	Результат вычисления числа Пи	Время вычисления
1	3.14209	0.8191
2	3.14184	0.6549
4	3.14146	0.4593
8	3.14158	0.3718
16	3.14112	0.2824
32	3.14086	0.2886

3. Решение одномерной задачи теплопроводности

Необходимо решить одномерное уравнение теплопроводности на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 4 процессорах):

$$\rho C_v \dot{T} = \lambda \mathrm{T}''$$

При применении метода конечных разностей получаем уравнение:

$$T_i^{k+1} = \frac{\alpha dt}{h^2} (T_{i+1}^k - 2T_i^k + T_{i-1}^k) + T_i^k$$

где α – коэффициент температуропроводности, который равен:

$$\alpha = \frac{\lambda}{\rho C_v}$$

Каждый расчёт проведен 10 раз для получения усредненного результата времени вычисления. Каждый процессор будет не только работать на своём участке и передавать получившееся значение в процессор с номером «0», но еще и в каждый момент времени взаимодействовать с «соседними» процессорами для передачи граничных значений. Решим задачу для стержня при следующих н.у. и г.у:

$$l = 1,$$

$$h = 0.01,$$

$$\alpha = 0.01172$$

$$t = 10,$$

$$dt = 0.001,$$

$$T_{0,t} = 0,$$

$$T_{1,t} = 0.5403,$$

$$T_{x,0} = \cos(x),$$

где l – длина стержня, h – шаг по пространству, t – время моделирования, dt – шаг по времени, T – температура.

Код реализации приведен ниже:

```
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // rank - номер процесса.

    MPI_Comm_size(MPI_COMM_WORLD,  &size);    //  size  -  общее  число
процессов.

    // Создание декартовой топологии.

    MPI_Comm comm_2D;

    int const ndim = 2;       // Размерность декартовой топологии

    int dims[ndim];            // Число процессов в каждом направлении
создаваемой декартовой топологии

    int periods[ndim];

    int  coords[ndim];            //  Координаты  процесса  в  декартовой
топологии

    dims[0] = (int)sqrt(size);     // Программу запускам на 1, 4 или 9
процессах

    dims[1] = (int)sqrt(size);

    periods[0] = 0;

    periods[1] = 0;

    MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, 0, &comm_2D);
// Создана декартова топология

    MPI_Cart_coords(comm_2D,        rank,        ndim,        coords);
// Определены координаты в декартовой топологии

    // Постановка задачи.

    int N = 240;               // N - размер задачи

    int m = (int)(N / sqrt(size));    // m - размер подзадачи

    auto*  T0  =  new  double[(m  +  2)  *  (m  +  2)];    //  Начальное
приближение

    auto* T1 = new double[(m + 2) * (m + 2)];  // Решение на новом
временном слое


    // Создание производных типов данных

    MPI_Datatype Column, Row;

    MPI_Type_vector(m + 2, 1, m + 2, MPI_DOUBLE, &Column);
```

```
    MPI_Type_commit(&Column);
    MPI_Type_vector(1, m + 2, m + 2, MPI_DOUBLE, &Row);
    MPI_Type_commit(&Row);
    // Опреденение направления сдвига
    int left, right, lower, upper;
    MPI_Cart_shift(comm_2D, 1, 1, &left, &right);
    MPI_Cart_shift(comm_2D, 0, 1, &lower, &upper);
    // Распределение процессов
    //  CPU 2 (1,0)   CPU 3 (1,1)
    //  CPU 0 (0,0)   CPU 1 (0,1)
    // Задание начальных условий
    for (int i = 0; i <= m + 1; i++)
        for (int j = 0; j <= m + 1; j++) {
            T0[i * (m + 2) + j] = 50;    // T0[i][j] = 50;
            T1[i * (m + 2) + j] = 50;    // T0[i][j] = 50;
        }
    // Задание граничных условий
    for (int i = 0; i <= m + 1; i++) {
        if (coords[1] == 0) T0[i * (m + 2)] = 10;                  //
T0[i][0]   =10;
        if (coords[1] == dims[0] - 1) T0[i * (m + 2) + m + 1] = 10;
// T0[i][m+1] =10;
    }
    for (int j = 0; j <= m + 1; j++) {
        if (coords[0] == 0) T0[0 + j] = 10;                // T0[j][0]
=10;
        if (coords[0] == dims[0] - 1) T0[(m + 1) * (m + 2) + j] = 10;
// T0[j][m+1] =10;
    }
    // Подготовка к расчетам
    double alfa = 0.01;
    double h = 1.0 / (N + 1);
    double tau = 0.2 * (h * h * h * h) / (alfa * 2 * h * h);
```

```
    double ae = alfa * tau / (h * h);
    double aw = alfa * tau / (h * h);
    double an = alfa * tau / (h * h);
    double as = alfa * tau / (h * h);
    double ap = ae + aw + an + as;
    double time = 0;
    double time_fin = max_time;
    // Расчет (движение на временной оси)
    double time_s = MPI_Wtime();
    for (time = 0; time < time_fin; time += tau) {
        // Расчет решение на новом временном шаге
        for (int i = 1; i < m + 1; i++)
            for (int j = 1; j < m + 1; j++)
                T1[i * (m + 2) + j] = (1.0 - ap) * T0[i * (m + 2) + j]
                + ae * T0[(i + 1) * (m + 2) + j] + aw * T0[(i - 1) *
(m + 2) + j]
                + an * T0[i * (m + 2) + j + 1] + as * T0[i * (m + 2) +
j - 1];
        for (int i = 1; i < m + 1; i++)
            for (int j = 1; j < m + 1; j++)
                T0[i * (m + 2) + j] = T1[i * (m + 2) + j];
        // Пересылка данных в фиктивные ячейки
        MPI_Status status;
        int tag = 10;
        MPI_Send(&T0[1], 1, Column, left, tag, comm_2D);
        MPI_Recv(&T0[m + 1], 1, Column, right, tag, comm_2D, &status);
        MPI_Send(&T0[m], 1, Column, right, tag, comm_2D);
        MPI_Recv(&T0[0], 1, Column, left, tag, comm_2D, &status);
        MPI_Send(&T0[1 * (m + 2)], 1, Row, lower, tag, comm_2D);
        MPI_Recv(&T0[(m + 1) * (m + 2)], 1, Row, upper, tag, comm_2D,
&status);
        MPI_Send(&T0[m * (m + 2)], 1, Row, upper, tag, comm_2D);
        MPI_Recv(&T0[0], 1, Row, lower, tag, comm_2D, &status);
```

```
}
double time_e = MPI_Wtime() - time_s;
// Сбор решения на нулевом процессе
auto** TG = new double* [N + 2];
for (int i = 0; i < N + 2; i++)
    TG[i] = new double[N + 2];
int l = 0;
auto* t_send = new double[m * m];
auto* t_recv = new double[N * N];
for (int i = 1; i < m + 1; i++)
    for (int j = 1; j < m + 1; j++) {
        t_send[l] = T0[i * (m + 2) + j];
        l = l + 1;
    }
MPI_Gather(&t_send[0], m * m, MPI_DOUBLE,
    &t_recv[0], m * m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank == 0) {
    l = 0;
    for (int di = 0; di < dims[0]; di++)
        for (int dj = 0; dj < dims[1]; dj++)
            for (int i = 1; i < m + 1; i++)
                for (int j = 1; j < m + 1; j++) {
                    TG[di * m + i][dj * m + j] = t_recv[l];
                    l = l + 1;
                }
}
// Результат
if (rank == 0) {
    std::cout << " Time work  = " << time_e << std::endl;

    if (writeFlag) {
        std::ofstream f;
```

```
            f.open("res_"   +   std::to_string(max_time)   +   ".txt",
std::ios::out);
            for (int i = 1; i < N + 1; i++) {
                for (int j = 1; j < N + 1; j++) {
                    f << TG[i][j] << " ";
                }
                f << std::endl;
            }
            f.close();
        }
    }
    MPI_Finalize();
```

Полученные результаты приведены в таблице 3.

Таблица 3. Зависимость времени вычисления от количества процессоров

Количество процессоров	Время вычисления
1	0.0105
2	0.0098
4	0.0104
8	0.1924
16	0.1956
32	0.0233

4. Моделирование методом динамики частиц

В рамках данного задания требуется смоделировать движение систему тел-точек методом динамики частиц.

Метод динамики частиц основан на представлении материала совокупностью взаимодействующих частиц (материальных точек или твердых тел), для которых записываются классические уравнения динамики. Взаимодействие частиц описывается посредством потенциалов взаимодействия, основным свойством которых является отталкивание при сближении и притяжение при удалении. Перед началом моделирования задается некоторое начальное распределение частиц в пространстве (исходная структура материала) и начальное распределение скоростей частиц (механическое и тепловое движение системы в исходном состоянии). Далее задача сводится к решению задачи Коши для системы обыкновенных дифференциальных уравнений. Традиционно, метод динамики частиц развивался на двух противоположных сторонах масштабной шкалы - для описания молекулярных систем, где в качестве частиц выступали атомы и молекулы, и для описания астрофизических систем, где в качестве частиц выступали объекты значительно большего масштабного уровня, такие как звезды или даже галактики. Не смотря на внешнюю несхожесть, и те и другие системы описываются сходными уравнениями. Постепенно, по мере развития вычислительной техники, данный метод стал все более широко применяться к описанию процессов на промежуточных масштабных уровнях, для моделирования физико-механических свойств материалов и гранулированных сред. В этом случае частицы могут представлять гранулы или зерна материала, однако они могут быть, и не связаны напрямую с некоторыми физическими объектами, а использоваться как конечные элементы для изучения процессов, в которых нарушается континуальность материала.

Рассмотрим двумерную задачу, где в некотором квадрате находится 1000 частиц, распределенных случайным образом. Начальные скорости равны нулю. Частицы взаимодействуют с потенциалом Леннарда-Джонса. Моделируется 0.1 секунда взаимодействия в 1000 шагов. Результаты расчета приведены в таблице 4.

Таблица 4. Результаты расчета метода динамики частиц

Шаг	Потенциальная энергия П	Кинетическая энергия К	Ошибка по полной энергии П + К
0	489123	0	0
100	489119	3.97766	E-11
200	489107	16.1924	E-10
300	489086	36.7639	E-9
400	489057	65.8928	E-9
500	489019	103.864	E-8
600	488972	151.052	E-8
700	488915	207.924	E-8
800	488848	275.050	E-8
900	488770	353.114	E-7
1000	488680	442.924	E-7

Заключение

Таким образом, в ходе выполнения работы был приобретен навык распараллеливания процесса с использованием программного интерфейса MPI. Исходя из полученных результатов видно, что при превышении количества имеющихся логических процессоров, в данном случае 4, скорость вычисления падает, что особенно заметно в 3 задаче.

Также было замечено, что скорость выполнения программы не имеет линейной зависимости с количеством используемых процессоров, так как часть ресурсов уходит на передачу, прием и обработку данных.

Список использованной литературы

1. Gropp W. et al. Using MPI: portable parallel programming with the message-passing interface. – MIT press, 1999. – Т. 1.

2. Pacheco P. Parallel programming with MPI. – Morgan Kaufmann, 1997.

3. Snir M. et al. MPI - the Complete Reference: the MPI core. – MIT press, 1998. – Т. 1.