Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный университет

Основы параллельного программирования

Отчет по лабораторной работе №1

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

Студент: Зуева Полина Геннадьевна, 23206

Преподаватель: Мичуров Михаил Антонович

Новосибирск, 2025 г.

## Цель работы

Изучить параллельную реализацию решения системы линейных алгебраических уравнений с помощью OpenMP.

## Описание работы

Была написана последовательная программа, реализующая итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, с помощью метода простой итерации. Исходные данные были выбраны такими: элементы главной диагонали A равны 2.0, остальные равны 1.0; вектор u = sin(2.0 \* PI \* i / N), где i = 1, …, N; элементы вектора b получаются путем умножения матрицы A на вектор u; начальные значения элементов вектора x равны 0.

После последовательной программы было написано два варианта параллельной программы с использованием OpenMP:

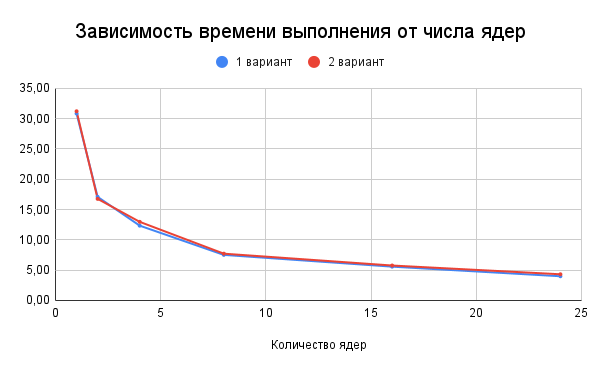
1). Для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for

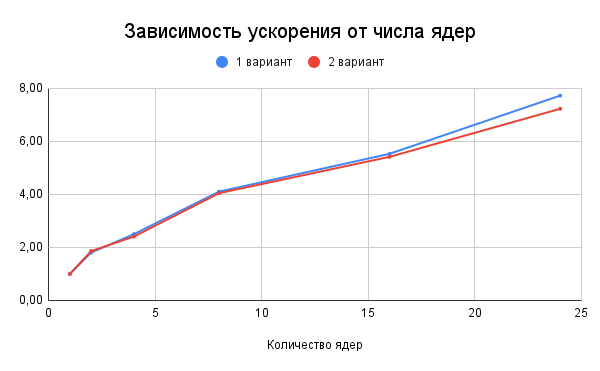
2). Создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

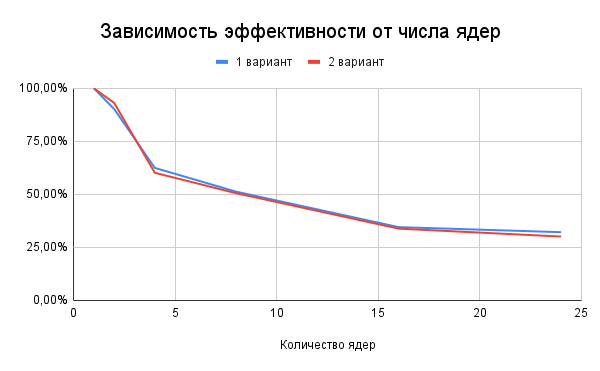
Программа на одном ядре работает чуть больше 30 секунд при числе N = 1580. Она была скомпилирована компилятором gcc на уровне оптимизации –О3.

У каждой распараллеленной программы был проведен замер времени. Каждая программа запускалась по 5 раз и было выбрано самое минимальное время ее работы. На основе этого была составлена таблица, где были посчитаны ускорения и эффективности распараллеливания от числа используемых ядер. После этого были построено 3 графика-сравнения, на которых можно увидеть, что первый вариант параллельного кода работает чуть-чуть быстрее, чем второй, но не всегда, например, при числе потоков равное 2. Можно сказать, что время работы обеих программы почти одинаково.

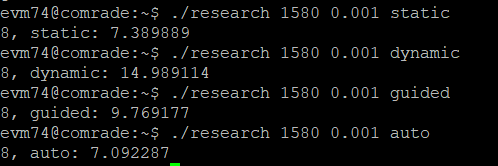
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 вариант | 2 вариант | 1 вариант | 2 вариант | 1 вариант | 2 вариант |
| Количество ядер | Минимальное время | | Ускорение | | Эффективность | |
| 1 | 30,83 | 31,21 | 1,00 | 1,00 | 100,00% | 100,00% |
| 2 | 17,09 | 16,76 | 1,80 | 1,86 | 90,22% | 93,11% |
| 4 | 12,34 | 12,97 | 2,50 | 2,41 | 62,47% | 60,17% |
| 8 | 7,52 | 7,72 | 4,10 | 4,04 | 51,28% | 50,56% |
| 16 | 5,57 | 5,76 | 5,53 | 5,42 | 34,56% | 33,85% |
| 24 | 3,99 | 4,32 | 7,72 | 7,23 | 32,18% | 30,11% |







Далее было проведено исследование на определение оптимальных параметров #pragma omp for schedule(...) при N = 1580 и количестве потоков равное 8. Проверялись такие расписания, как static, dynamic, guided, auto. Было выяснено, что при static и auto время работы программы почти одинаково, при guided больше, а при dynamic – максимальное время работы программы.



- Static: итерации цикла делятся на равные блоки, и каждый блок назначается определённому потоку, используется, когда нагрузка на каждой итерации цикла примерно одинаковая.

- Dynamic: итерации цикла распределяются динамически, используется, когда нагрузка на итерациях неравномерная, хорошо балансирует нагрузку между потоками.

- Guided: итерации распределяются блоками, размер которых уменьшается по мере выполнения. В начале блоки большие, но ближе к концу они становятся меньше. Это позволяет минимизировать накладные расходы, сохраняя баланс нагрузки.

- Auto: компилятор или среда выполнения OpenMP самостоятельно выбирает тип расписания.

На основании полученных данных можно сделать следующие выводы:

* Оба варианта параллельного кода дают практически одинаковые результаты по времени выполнения.
* В некоторых случаях первый вариант (#pragma omp parallel for для каждого цикла) работает немного быстрее второго варианта (единая параллельная секция #pragma omp parallel, охватывающая весь итерационный процесс), но разница незначительна.
* При увеличении количества потоков наблюдается рост ускорения и снижение эффективности параллельного выполнения, что объясняется накладными расходами на управление потоками и синхронизацию.
* Наибольшее ускорение достигается при 24 потоках, однако эффективность параллелизации снижается до 32,18% для первого варианта и до 30,11% для второго варианта.
* Таким образом, оба варианта параллельного кода демонстрируют схожие характеристики, и выбор между ними зависит от особенностей конкретной системы и возможных накладных расходов при управлении потоками.

## Приложения

Ссылка на репозиторий: [https://github.com](https://github.com/PolinaZueva/OPP/issues)

gcc -o var1 var1.c -Wall -Werror -Wextra -O3 -fopenmp -lm

gcc -o var2 var2.c -Wall -Werror -Wextra -O3 -fopenmp –lm

gcc -o research research.c -Wall -Werror -Wextra -O3 -fopenmp -lm

**Приложение 1.** Последовательный код.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define PI 3.14159265358979323846

#define E 1e-5

double calculateNorm(double\* b, int N){

    double sum = 0.0;

    for (int i = 0; i < N; i++){

        sum += b[i] \* b[i];

    }

    return sqrt(sum);

}

void subVectors(double\* Ax, double\* b, int N, double\* res) {

    for (int i = 0; i < N; i++){

        res[i] = Ax[i] - b[i];

    }

}

void multiplyMatrixVector(double\* A, double\* x, int N, double\* res) {

    for (int i = 0; i < N; i++) {

        double sum = 0.0;

        for (int j = 0; j < N; j++) {

            sum += A[i \* N + j] \* x[j];

        }

        res[i] = sum;

    }

}

int test(double\* tempVector, double\* b, int N) {

    return (calculateNorm(tempVector, N) / calculateNorm(b, N)) < E;

}

void multiplyNumberVector(double tau, double\* tempVector, int N, double\* res) {

    for (int i = 0; i < N; i++) {

        res[i] = tau \* tempVector[i];

    }

}

int main(int argc, char\* argv[]) {

    if (argc != 3) {

        printf("Use: %s <N> <tau> \n", argv[0]);

        return 0;

    }

    int N = atoi(argv[1]);

    double tau = atof(argv[2]);

    double\* A = (double\*)malloc(N \* N \* sizeof(double));

    double\* x = (double\*)malloc(N \* sizeof(double));

    double\* b = (double\*)malloc(N \* sizeof(double));

    double\* u = (double\*)malloc(N \* sizeof(double));

    double\* Ax = (double\*)malloc(N \* sizeof(double));

    double\* tempVector = (double\*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++) {

        u[i] = sin(2.0 \* PI \* i / N);

        x[i] = 0.0;

        for (int j = 0; j < N; j++) {

            A[i \* N + j] = 1.0;

            if (i == j) {

                A[i \* N + j] = 2.0;

            }

        }

    }

    multiplyMatrixVector(A, u, N, b);

    struct timespec start, end;

    clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

    int countIterations = 0;

    while (1) {

        multiplyMatrixVector(A, x, N, Ax);

        subVectors(Ax, b, N, tempVector);

        if (test(tempVector, b, N)) break;

        multiplyNumberVector(tau, tempVector, N, tempVector);

        subVectors(x, tempVector, N, x);

        if (++countIterations > N \* 100) {

            printf("Метод не сошелся, попробуйте другое значение tau\n");

            break;

        }

    }

    clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

    double difTime = (end.tv\_sec - start.tv\_sec) + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec);

    printf("Время выполнения: %lf секунд\n", difTime);

    for (int i = 0; i < 5 && i < N; i++) {

        printf("x[%d] = %f\n", i, x[i]);

    }

    free(A);

    free(x);

    free(b);

    free(u);

    free(Ax);

    free(tempVector);

    return 0;

}

**Приложение 2.** Параллельный код, 1 вариант.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <omp.h>

#define PI 3.14159265358979323846

#define E 1e-5

double calculateNorm(double\* b, int N){

    double sum = 0.0;

    #pragma omp parallel for reduction(+:sum) schedule(static)

    for (int i = 0; i < N; i++){

        sum += b[i] \* b[i];

    }

    return sqrt(sum);

}

void subVectors(double\* Ax, double\* b, int N, double\* res) {

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < N; i++){

        res[i] = Ax[i] - b[i];

    }

}

void multiplyMatrixVector(double\* A, double\* x, int N, double\* res) {

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < N; i++) {

        double sum = 0.0;

        for (int j = 0; j < N; j++) {

            sum += A[i \* N + j] \* x[j];

        }

        res[i] = sum;

    }

}

int test(double\* tempVector, double\* b, int N) {

    return (calculateNorm(tempVector, N) / calculateNorm(b, N)) < E;

}

void multiplyNumberVector(double tau, double\* tempVector, int N, double\* res) {

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < N; i++) {

        res[i] = tau \* tempVector[i];

    }

}

int main(int argc, char\* argv[]) {

    if (argc != 3) {

        printf("Use: %s <N> <tau> \n", argv[0]);

        return 0;

    }

    int N = atoi(argv[1]);

    double tau = atof(argv[2]);

    double\* A = (double\*)malloc(N \* N \* sizeof(double));

    double\* x = (double\*)malloc(N \* sizeof(double));

    double\* b = (double\*)malloc(N \* sizeof(double));

    double\* u = (double\*)malloc(N \* sizeof(double));

    double\* Ax = (double\*)malloc(N \* sizeof(double));

    double\* tempVector = (double\*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++) {

        u[i] = sin(2.0 \* PI \* i / N);

        x[i] = 0.0;

        for (int j = 0; j < N; j++) {

            A[i \* N + j] = 1.0;

            if (i == j) {

                A[i \* N + j] = 2.0;

            }

        }

    }

    multiplyMatrixVector(A, u, N, b);

    int threads[] = {1, 2, 4, 8, 16, 24};

    int numberThreads = sizeof(threads) / sizeof(threads[0]);

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

        omp\_set\_num\_threads(threads[i]);

        for (int j = 0; j < N; j++) {

            x[j] = 0.0;

        }

        double start = omp\_get\_wtime();

        while (1) {

            multiplyMatrixVector(A, x, N, Ax);

            subVectors(Ax, b, N, tempVector);

            if (test(tempVector, b, N)) break;

            multiplyNumberVector(tau, tempVector, N, tempVector);

            subVectors(x, tempVector, N, x);

        }

        double end = omp\_get\_wtime();

        printf("%d: %lf\n", threads[i], end - start);

    }

    free(A);

    free(x);

    free(b);

    free(u);

    free(Ax);

    free(tempVector);

    return 0;

}

**Приложение 3.** Параллельный код, 2 вариант.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <omp.h>

#define PI 3.14159265358979323846

#define E 1e-5

double calculateNorm(double\* b, int N){

    double sum = 0.0;

    #pragma omp for

    for (int i = 0; i < N; i++){

        #pragma omp atomic

        sum += b[i] \* b[i];

    }

    return sqrt(sum);

}

void subVectors(double\* Ax, double\* b, int N, double\* res) {

    #pragma omp for schedule(static)

    for (int i = 0; i < N; i++){

        res[i] = Ax[i] - b[i];

    }

}

void multiplyMatrixVector(double\* A, double\* x, int N, double\* res) {

    #pragma omp for schedule(static)

    for (int i = 0; i < N; i++) {

        double sum = 0.0;

        for (int j = 0; j < N; j++) {

            sum += A[i \* N + j] \* x[j];

        }

        res[i] = sum;

    }

}

int test(double\* tempVector, double\* b, int N) {

    return (calculateNorm(tempVector, N) / calculateNorm(b, N)) < E;

}

void multiplyNumberVector(double tau, double\* tempVector, int N, double\* res) {

    #pragma omp for schedule(static)

    for (int i = 0; i < N; i++) {

        res[i] = tau \* tempVector[i];

    }

}

int main(int argc, char\* argv[]) {

    if (argc != 3) {

        printf("Use: %s <N> <tau> \n", argv[0]);

        return 0;

    }

    int N = atoi(argv[1]);

    double tau = atof(argv[2]);

    double\* A = (double\*)malloc(N \* N \* sizeof(double));

    double\* x = (double\*)malloc(N \* sizeof(double));

    double\* b = (double\*)malloc(N \* sizeof(double));

    double\* u = (double\*)malloc(N \* sizeof(double));

    double\* Ax = (double\*)malloc(N \* sizeof(double));

    double\* tempVector = (double\*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++) {

        u[i] = sin(2.0 \* PI \* i / N);

        x[i] = 0.0;

        for (int j = 0; j < N; j++) {

            A[i \* N + j] = 1.0;

            if (i == j) {

                A[i \* N + j] = 2.0;

            }

        }

    }

    multiplyMatrixVector(A, u, N, b);

    int threads[] = {1, 2, 4, 8, 16, 24};

    int numberThreads = sizeof(threads) / sizeof(threads[0]);

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

        omp\_set\_num\_threads(threads[i]);

        for (int j = 0; j < N; j++) {

            x[j] = 0.0;

        }

        double start = omp\_get\_wtime();

        #pragma omp parallel

        {

            while (1) {

                multiplyMatrixVector(A, x, N, Ax);

                subVectors(Ax, b, N, tempVector);

                if (test(tempVector, b, N)) break;

                multiplyNumberVector(tau, tempVector, N, tempVector);

                subVectors(x, tempVector, N, x);

            }

        }

        double end = omp\_get\_wtime();

        printf("%d: %lf\n", threads[i], end - start);

    }

    free(A);

    free(x);

    free(b);

    free(u);

    free(Ax);

    free(tempVector);

    return 0;

}