Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Практикум по курсу

«Суперкомпьютеры и параллельная обработка данных»

Разработка параллельной версии программы для задачи перемножения матриц алгоритмом Фокса с использованием системы OpenMP.

Отчёт
О выполненном задании

Выполнил:
студент 323 группы
Латыпов Ш. И.

Москва
2021

Содержание

Описание задачи

Задача состояла в том, чтобы реализовать метод Фокса умножения квадратных матриц.

Для этого матрицы A и B делились на блоки размера q, кратного размеру самих матриц. Каждый процесс с координатами (i, j) получал блоки A_{ij}, B_{ij} исходных матриц. Также выполнялось обнуление блока C_{ij}, предназначенного для хранения соответствующего блока результирующего произведения матрицы $C = A \cdot B$.

Затем запускался цикл по m $(m = 0, ..., q - 1)$, в ходе которого выполнялось три действия:

1. Для каждой строки i $(i = 0, ..., q)$ блок A_{ij} одного из процессов пересылался во все процессы этой же строки; при этом индекс j пересылаемого блока определялся по формуле $j = (i + m)\ mod\ q$;

2. Полученный в результате подобной пересылки блок матрицы A и содержащийся в процессе (i, j) блок матрицы B перемножались, и результат прибавлялся к блоку C_{ij};

3. Для каждого столбца j $(j = 0, ..., q)$ выполнялась циклическая пересылка блоков матрицы B, содержащихся в каждом процессе (i, j) этого столбца, в направлении убывания номеров строк.

После завершения цикла каждый процесс преобразует блок C_{ij}, равный соответствующему блоку произведения $A \cdot B$, и соответственно матрица C будет равна произведернию матриц $A \cdot B$.

Алгоритм программы

Для всех матриц память выделяется динамически, размеры матриц и размер блоков задается пользователем (также проверяется, что входные данные корректны). Элементы матриц A и B принимают значения, соответсвующие их порядковому номеру в матрице (то есть элемент матрицы a_{ij} был соответсвенно раверн $i \cdot matrix\_size + j$). Так как каждый процесс считает один блок матрицы C, то всего процессов в программе $block\_cnt^2$, где $block\_cnt$ - количетсво блоков в строке/столбце матриц A и B ($block\_cnt = matrix\_size \ / \ block\_size$).

```c
int main(int argc, char **argv) {
    int matrix_size = atoi(argv[1]), block_size = atoi(argv[2]);
    if (matrix_size % block_size != 0) {
        printf("Incorrect input. Block_size must be a multiple of the matrix size\n");
        return 0;
    }
    int **matrix_a, **matrix_b, **matrix_c;

    matrix_a = calloc(matrix_size, sizeof(*matrix_a));
    matrix_b = calloc(matrix_size, sizeof(*matrix_b));
    matrix_c = calloc(matrix_size, sizeof(*matrix_c));

    for (int i = 0; i < matrix_size; i++) {
        matrix_a[i] = calloc(matrix_size, sizeof(*matrix_a[i]));
        matrix_b[i] = calloc(matrix_size, sizeof(*matrix_b[i]));
        matrix_c[i] = calloc(matrix_size, sizeof(*matrix_c[i]));
    }

    for (int i = 0; i < matrix_size; i++) {
        for (int j = 0; j < matrix_size; j++) {
            matrix_a[i][j] = i * matrix_size + j;
            matrix_b[i][j] = matrix_a[i][j];
        }
    }
```

Рис. 1: Выделение памяти для матриц и заполнение

```
int block_cnt = matrix_size / block_size;

double start_time = omp_get_wtime();

int p = 0;
omp_set_num_threads(block_cnt * block_cnt);
#pragma omp parallel for private(p) shared(matrix_a, matrix_b, matrix_c, matrix_size, block_size, block_cnt)
for (p = 0; p < block_cnt * block_cnt; p++) {
    int i = p / block_cnt, j = p % block_cnt;
    int **get_a, **get_b;
    get_a = calloc(block_size, sizeof(*get_a));
    get_b = calloc(block_size, sizeof(*get_b));

    for (int k = 0; k < block_size; k++) {
        get_a[k] = calloc(block_size, sizeof(*get_a[k]));
        get_b[k] = calloc(block_size, sizeof(*get_b[k]));
    }

    get_matrix_b(matrix_b, get_b, j, block_size, i * block_size);

    int m = 0;
    for (m = 0; m < block_cnt; m++) {
        int col_num_sended_A = (i + m) % block_cnt;
        get_matrix_a(matrix_a, get_a, i, block_size, col_num_sended_A * block_size);

        matrix_mul(get_a, get_b, matrix_c, i, j, block_size);

        if ((i + m + 1) * block_size >= matrix_size) {
            get_matrix_b(matrix_b, get_b, j, block_size, (i + m + 1) * block_size - matrix_size);
        } else {
            get_matrix_b(matrix_b, get_b, j, block_size, (i + m + 1) * block_size);
        }
    }

    for (int k = 0; k < block_size; k++) {
        free(get_a[k]);
        free(get_b[k]);
    }
    free(get_a);
    free(get_b);
}

printf("Result matrix C is:");
print_m(matrix_c, matrix_size);

double end_time = omp_get_wtime();
printf("Time taken to execute is\n%lf\n", end_time - start_time);
```

Рис. 2: Параллельная часть программы

Алгоритм подразумевает, что в каждом процессе будет дополнительно выделяться память для блоков A_{ij} и B_{ij} соответсвенно. Затем процесс запускает цикл по m и в зависимости от своих координат (i, j) заполняет свои блоки нужными значениями. Для этого в программе были написаны отдельные функции $get\_matrix\_a()$ и $get\_matrix\_b()$, которые в зависимости из входных параметров заполняли блок значениями из исходных матриц A, B. Умножение блоков реализовано в функции $matrix\_mul()$

```
void get_matrix_a(int **m_a, int **get_a, int block_i, int block_size, int col_param) {
    for (int i = 0; i < block_size; i++) {
        for (int j = 0; j < block_size; j++) {
            get_a[i][j] = m_a[block_i * block_size + i][col_param + j];
        }
    }
}

void get_matrix_b(int **m_b, int **get_b, int block_j, int block_size, int row_param) {
    for (int i = 0; i < block_size; i++) {
        for (int j = 0; j < block_size; j++) {
            get_b[i][j] = m_b[row_param + i][block_j * block_size + j];
        }
    }
}

void matrix_mul(int **m_a, int **m_b, int **m_c, int block_i, int block_j, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                m_c[block_i * size + i][block_j * size + j] += m_a[i][k] * m_b[k][j];
            }
        }
    }
}
```

Рис. 3: Дополнительные функции для заполнения блоков процессов и их умножения

Также для вывода матриц на экран реализована функция $print\_m()$, которая дополнительно выравнивает вывод на экран в зависимости от кратности и размера чисел в нем

```
void print_m(int **m, int size) {
    printf("\n");
    char output_format[] = "%%d ";
    int max_elem = m[size - 1][size - 1];
    int x = 1, st = 0;
    while (x < max_elem) {
        x *= 10;
        st++;
    }
    output_format[1] = st + '0';
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf(output_format, m[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

Рис. 4: Вывод матрицы на экран

Результаты замеров

В таблице приведены результаты времени работы программы (в секундах) в параллельном режиме в зависимости от введённых данных.

	Количество нитей						
Размер матриц NxN		1	4	16	36	64	144
	24	0.007407	0.007503	0.007887	0.008966	0.011018	0.012704
	48	0.007642	0.007675	0.008137	0.009066	0.011048	0.012840
	72	0.007817	0.007670	0.008078	0.008980	0.011157	0.012788
	96	0.008231	0.007807	0.008142	0.009051	0.010976	0.012881
	120	0.008943	0.007984	0.008248	0.009058	0.011256	0.013055
	144	0.010024	0.008589	0.008209	0.009829	0.011004	0.012997
	168	0.011440	0.008618	0.008623	0.009618	0.011744	0.013148
	192	0.013466	0.009187	0.008808	0.009565	0.011685	0.013406
	216	0.015912	0.009771	0.009006	0.009915	0.012428	0.014438
	240	0.018880	0.010528	0.010492	0.010087	0.012544	0.021251

Рис. 5: Таблица времени работы программы

Также представлен 3D график по соответствующим данным.

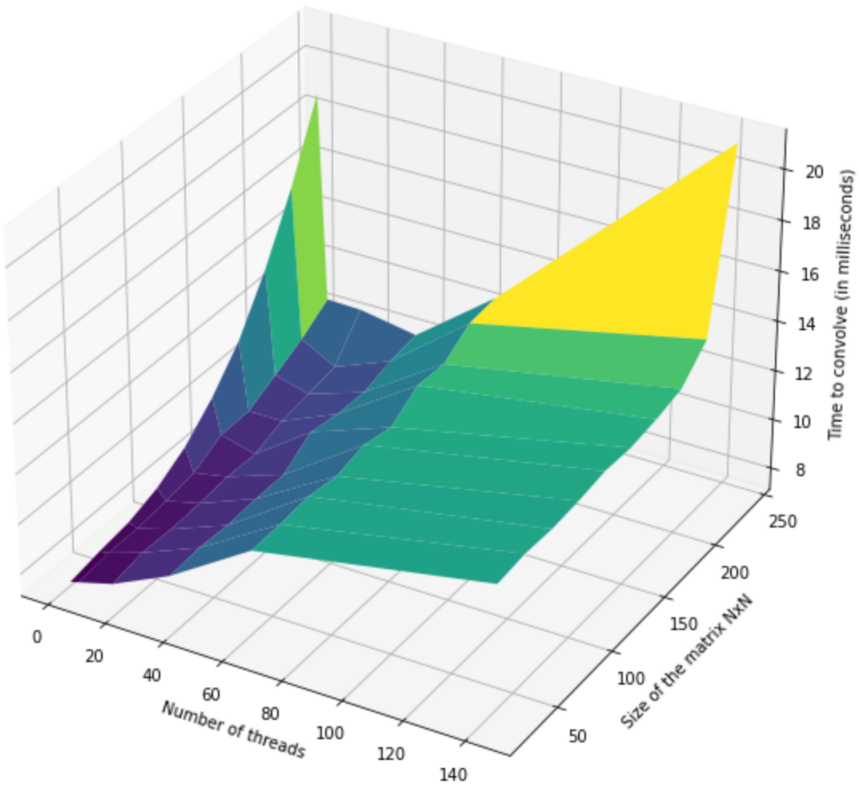

Рис. 6: Таблица времени работы программы

Правильность алгоритма проверялось функцией $matrix\_mul()$. Программа запускалась отдельно с вызовом только этой функции, куда передавались исходные матрицы A и B. Полученная матрица сравнивалась с матрицей, полученной путем распараллелирования программы.

Анализ результатов

При малых размерах матрицы распараллелирование не даёт преимущества по времени, а даже наоборот, увеличивает его. Это связано с издержками на планирование доступа к разделяемому ресурсу и с последующими операциями с выделением памяти на блоки для всех нитей и соответсвующего "копирования"данных из исходных матриц.

При рассмотрении матриц большего размера видно, что программа наиболее эффективна при "среднем"количестве нитей в программе (особенно заметно при размере матриц больше 168). Это следует из-за того, что при большом размере блоков (соответсвенно малом количестве нитей) программа долго умножает $A_{ij} \cdot B_{ij}$, но при большом количестве нитей возвращается проблема разделени ресурсов и выделения памяти.

Выводы

Выполнена работа по разработке параллельной версии программы для задачи перемножения матриц алгоритмом Фокса. Изучены средства параллельных алгоритмов системы OpenMP.

Также проанализировано время работы программы при разных входных параметрах на суперкомпьютере МГУ. Сделан вывод о том, что при правильном расчёте количества нитей для матриц разного размера можно заметить прирост производительности по времени на системах, рассчитаных для работы с многопоточными вычислениями.