Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №2 по курсу

«Операционные системы»

Группа: М8О-211БВ-24

Студент: Мутагиров Т.Р.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 14.10.25

Москва, 2025

Постановка задачи

Вариант 6.

Произвести перемножение 2-ух матриц, содержащих комплексные числа

Общий метод и алгоритм решения

Использованные системные вызовы:

- **int pthread_create(pthread_t thread, const pthread_attr_t attr, void (work)(void), void arg); – Создает поток с заданными атрибутами, который начинает выполнение функции work
- **int pthread_join(pthread_t thread, void retval); – ожидает завершения указанного потока.
- int clock_gettime(clockid_t clk_id, struct timespec *tp); – получает текущее монотоное время системы

  struct timespec {
  time_t tv_sec; - секунды
  long tv_nsec; - наносекунды
  };

Я реализовал программу, которая использует многопоточность для умножения матриц содержащих комплексные числа. Использовать мьютекс не приишлось так как нету гонки данных, так как разные потоки записывают данные в разные ячейки памяти

Количество потоков подается как ключ к моей программе. Затем я отдаю на каждый поток какое то кол-во строк и столбцов которые должны быть перемножены. Значения ячеек матриц которые должны быть перемножены инициализируются случайными числами

Код программы

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

typedef struct {
    int row, column;
} MArg;
```

```
typedef struct {
    int st, end;
} TArg;



#define N 1000
#define M 1000
// matr[i][j][0] - real part of number in i row and j column
// matr[i][j][1] - compl part of i number in row and j column
int matr1[N][M][2], matr2[M][N][2], res[N][N][2];
MArg args[N * M];

void calc(MArg *arg) {
    int r = arg->row, c = arg->column;
    int *v1, *v2, *vres;
    for(int i = 0; i < M; ++i) {
        // matr1[r][i] * matr2[i][c];
        // (a + bi) * (c + di) = (ac - bd) + (ad + bc)i
        // a = matr1[r][i][0], b = matr1[r][i][1];
        // c = matr2[i][c][0], d = matr2[i][c][1];
        v1 = matr1[r][i], v2 = matr2[i][c];
        vres = res[r][c];
        vres[0] = v1[0] * v2[0] - v1[1] * v2[1];
        vres[1] = v1[0] * v2[1] + v1[1] * v2[0];
    }
}

void consistent() {
    for(int i = 0; i < N * M; ++i) {
        calc(args + i);
    }
}

static void *work(void *_arg) {
```

```
    TArg *arg = (TArg *)_arg;
    int st = arg->st, end = arg->end;
    for(int i = st; i < end; i++) {
        calc(args + i);
    }
    return NULL;
}

void paralel(int tcnt) {
    pthread_t *threads = malloc(tcnt * sizeof(pthread_t));
    int totTasks = N * M;
    int taskPerTread = totTasks / tcnt, rem = totTasks % tcnt;
    int curCnt, prevI = 0;
    for(int i = 0; i < tcnt; ++i) {
        curCnt = taskPerTread;
        if(rem) {
            ++curCnt;
            --rem;
        }
        TArg t = {prevI, prevI += curCnt};
        pthread_create(threads + i, NULL, work, &t);
    }
    for(int i = 0; i < tcnt; ++i) {
        pthread_join(threads[i], NULL);
    }
    free(threads);
}

double getDif(struct timespec start, struct timespec end) {
    return 1e3 * ((end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9);
}

int main() {
    for(int r = 0; r < N; ++r) {
```

```
        for(int c = 0; c < M; ++c) {
            args[r * M + c] = (MArg){r, c};
            matr1[r][c][0] = rand() % N;
            matr1[r][c][1] = rand() % N;
        }
    }
    for(int r = 0; r < M; ++r) {
        for(int c = 0; c < N; ++c) {
            matr2[r][c][0] = rand() % N;
            matr2[r][c][1] = rand() % N;
        }
    }

    struct timespec st, end;
    size_t tn = sysconf(_SC_NPROCESSORS_ONLN);
    double ts, curT;
    printf("Total thread count %lu\n\n", tn);

    {
    // experiment consistent
    printf("===== experiment consistent =====\n");
    clock_gettime(CLOCK_MONOTONIC, &st);
    consistent();
    clock_gettime(CLOCK_MONOTONIC, &end);
    ts = getDif(st, end);
    curT = getDif(st, end);
    printf("Time : %lf ms. Acceleration %lf . Efficiency %lf\n", curT, ts / curT, ts /
curT / 1);
    printf("\n\n");

    // experiment Paralel <= tn
    printf("===== experiment Paralel threads < logic Thread =====\n");
    for(int i = 1; i < tn; ++i) {
        clock_gettime(CLOCK_MONOTONIC, &st);
        paralel(i);
```

```
        clock_gettime(CLOCK_MONOTONIC, &end);
        curT = getDif(st, end);
        printf("ThreadsCount %d . Time : %lf ms. Acceleration %lf . Efficiency %lf\n",
                i, curT, ts / curT, ts / curT / i);
    }
    printf("\n\n");

    printf("===== experiment threads = logic threads =====\n");
    clock_gettime(CLOCK_MONOTONIC, &st);
    paralel(tn);
    clock_gettime(CLOCK_MONOTONIC, &end);
    curT = getDif(st, end);
    printf("ThreadsCount %d . Time : %lf ms. Acceleration %lf . Efficiency %lf\n",
                tn, curT, ts / curT, ts / curT / tn);
    printf("\n\n");

    int bigCnt[3] = {16, 32, 1024};
    printf("===== experiment threads > logic threads =====\n");
    for(int i = 0; i < 3; ++i) {
        clock_gettime(CLOCK_MONOTONIC, &st);
        paralel(bigCnt[i]);
        clock_gettime(CLOCK_MONOTONIC, &end);
        curT = getDif(st, end);
        printf("ThreadsCount %d . Time : %lf ms. Acceleration %lf . Efficiency %lf\n",
                bigCnt[i], curT, ts / curT, ts / curT / bigCnt[i]);
    }
    printf("\n\n");
    }
    return 0;
}
```

Протокол работы программы

› ./main.exe

Total thread count 12

First matrix size (1000 * 1000), second matrix size (1000 * 1000)

===== experiment consistent =====

Time : 5272.033578 ms. Acceleration 1.000000 . Efficiency 1.000000

===== experiment Paralel threads < logic Thread =====

ThreadsCount 1 . Time : 5012.298274 ms. Acceleration 1.051820 . Efficiency 1.051820

ThreadsCount 2 . Time : 2377.711087 ms. Acceleration 2.217273 . Efficiency 1.108636

ThreadsCount 3 . Time : 1570.733470 ms. Acceleration 3.356415 . Efficiency 1.118805

ThreadsCount 4 . Time : 1193.564721 ms. Acceleration 4.417049 . Efficiency 1.104262

ThreadsCount 5 . Time : 1249.136997 ms. Acceleration 4.220541 . Efficiency 0.844108

ThreadsCount 6 . Time : 1177.161347 ms. Acceleration 4.478599 . Efficiency 0.746433

ThreadsCount 7 . Time : 1080.516376 ms. Acceleration 4.879180 . Efficiency 0.697026

ThreadsCount 8 . Time : 984.306939 ms. Acceleration 5.356087 . Efficiency 0.669511

ThreadsCount 9 . Time : 910.709603 ms. Acceleration 5.788929 . Efficiency 0.643214

ThreadsCount 10 . Time : 818.816632 ms. Acceleration 6.438601 . Efficiency 0.643860

ThreadsCount 11 . Time : 766.032724 ms. Acceleration 6.882256 . Efficiency 0.625660

===== experiment threads = logic threads =====

ThreadsCount 12 . Time : 717.189858 ms. Acceleration 7.350959 . Efficiency 0.612580

===== experiment threads > logic threads =====

ThreadsCount 16 . Time : 769.270708 ms. Acceleration 6.853288 . Efficiency 0.428330

ThreadsCount 32 . Time : 686.630179 ms. Acceleration 7.678127 . Efficiency 0.239941

ThreadsCount 1024 . Time : 754.566810 ms. Acceleration 6.986835 . Efficiency 0.006823

Вывод

В ходе данной лабораторной работы я научился работать с потоками. Разобрался с различными проблемами и нюансами при работе с ними. Также получил следующие выводы, которые я изображу в виде таблицы

Число потоков	Время исполнения (мс)	Ускорение	Эффективность
Последовательно	5272.033578	1	1
1	5012.298274	1.051820	1.0518
2	6739.873	1.99	0.99
8	984.306939	5.356087	0.669511
11	766.032724	6.882256	0.625660
12	717.189858	7.350959	0.612580
16	69.270708	6.853288	0.428330
32	686.630179	7.678127	0.239941
1024	754.566810	6.986835	0.006823

Расчеты:

- **Ускорение: T1/Tn, где T1 – время выполнения последовательно,**
  Tn – время выполнения с n потоками
- **Эффективность: Ускорение/n**

Анализ результатов:

Вывод

На основе проведенных экспериментов можно сделать следующие выводы о поведении многопоточной программы в зависимости от количества потоков:

1. Количество потоков МЕНЬШЕ логических ядер процессора (1-11 потоков)

При увеличении количества потоков от 1 до 11 наблюдается значительное улучшение производительности:

- Время выполнения последовательно уменьшается с 5012 мс до 766 мс
- Ускорение растет практически линейно от 1.05 до 6.88 раз
- Эффективность остается на высоком уровне (0.63-1.05), что свидетельствует о хорошем использовании ресурсов процессора

Это объясняется тем, что каждый поток получает выделенное ядро процессора, минимизируются накладные расходы на переключение между потоками.

2. Количество потоков РАВНО логическим ядрам процессора (12 потоков)

При 12 потоках достигается оптимальный баланс:

- Минимальное время выполнения - 717 мс
- Максимальное ускорение в этой категории - 7.35 раз
- Эффективность составляет 0.61, что является хорошим показателем

12 потоков полностью используют все логические ядра процессора, достигая пиковой производительности для данной архитектуры.

3. Количество потоков БОЛЬШЕ логических ядер процессора (16+ потоков)

При дальнейшем увеличении количества потоков наблюдается насыщение производительности:

- Время выполнения практически не улучшается (686-754 мс)
- Ускорение достигает плато около 7.68 раз
- Эффективность резко падает до 0.24-0.01

Это объясняется:

- Накладными расходами на переключение между потоками
- Конкуренцией за ресурсы процессора
- Ограничениями закона Амдала - даже при бесконечном числе потоков ускорение ограничено последовательной частью программы

Общий вывод

Программа демонстрирует классическое поведение многопоточных приложений: линейный рост производительности до количества логических ядер процессора с последующим насыщением и падением эффективности.