Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету

"Операционные системы"

№2

Студент: Филиппов А. М.

Преподаватель: Миронов Е.С.

Группа: М8О-201Б-24

Дата:

Оценка:

Подпись:

Москва, 2025

Оглавление

Цель работы

Целью является приобретение практических навыков в:

- Управление потоками в ОС
- Обеспечение синхронизации между потоками

Постановка задачи

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

4. Отсортировать массив целых чисел при помощи TimSort

Общие сведения о программе

Программа компилируется из одного файла main.c. В данном файле используется заголовочный файл lostream. В программе используются системные вызовы для работы с потоками из заголовочного файла thread.h.

Общий алгоритм решения

Данная программа сортирует массив arr размеро, который вводится с консоли, используя сортировку TimSort с помощью многопоточности. Количество потоков, работающих в один момент времени, можно задать через ключ запуска программы. Если не задано, то количество потоков будет равно 4 (значение по умолчанию). Программа разбивает массив на части и каждая часть сортируется в отдельном потоке. Затем отсортированные части объединяются с помощью функции "merge".

Реализация

gauss_solver.c

```
#include <math.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>

#define EPSILON 1e-9
#define MAX_MATRIX_SIZE 5000

typedef struct {
double **matrix;
int size;
int thread_id;
int num_threads;
pthread_barrier_t *barrier;
volatile int *error_flag;
} thread_data_t;

typedef struct {
double sequential_time;
double parallel_time;
int num_threads;
int matrix_size;
double speedup;
double efficiency;
} performance_stats_t;

static performance_stats_t stats;

double **allocate_matrix(int rows, int cols) {
double **mat = (double **)malloc(rows * sizeof(double *));
if (mat == NULL) {
return NULL;
}

for (int i = 0; i < rows; i++) {
mat[i] = (double *)malloc(cols * sizeof(double));
if (mat[i] == NULL) {
for (int j = 0; j < i; j++) {
free(mat[j]);
```

```
}
free(mat);
return NULL;
}
}
return mat;
}

void free_matrix(double **mat, int rows) {
if (mat == NULL)
return;
for (int i = 0; i < rows; i++) {
free(mat[i]);
}
free(mat);
}

void copy_matrix(double **dest, double **src, int rows, int cols) {
for (int i = 0; i < rows; i++) {
memcpy(dest[i], src[i], cols * sizeof(double));
}
}

void print_matrix(double **mat, int rows, int cols, const char *title) {
if (rows > 10 || cols > 10) {
printf("%s: Matrix %dx%d (too large to print)\n", title, rows, cols);
return;
}

printf("\n%s (%dx%d):\n", title, rows, cols);
for (int i = 0; i < rows; i++) {
printf(" [");
for (int j = 0; j < cols; j++) {
if (j == cols - 1) {
printf(" | %8.4f", mat[i][j]);
} else {
printf("%8.4f ", mat[i][j]);
}
}
printf("]\n");
}
printf("\n");
}

void generate_test_matrix(double **matrix, int size) {
for (int i = 0; i < size; i++) {
double sum = 0.0;
for (int j = 0; j < size; j++) {
```

```
if (i == j) {
matrix[i][j] = 10.0 + i * 2.0;
} else {
matrix[i][j] = 1.0;
}
sum += matrix[i][j];
}
matrix[i][size] = sum + i * 0.5;
}
}

double get_time_ms(void) {
struct timeval tv;
gettimeofday(&tv, NULL);
return tv.tv_sec * 1000.0 + tv.tv_usec / 1000.0;
}

int gauss_sequential(double **matrix, int size) {
for (int k = 0; k < size; k++) {
int max_row = k;
double max_val = fabs(matrix[k][k]);
for (int i = k + 1; i < size; i++) {
if (fabs(matrix[i][k]) > max_val) {
max_val = fabs(matrix[i][k]);
max_row = i;
}
}

if (fabs(matrix[max_row][k]) < EPSILON) {
fprintf(stderr, "Error: Matrix is singular at row %d\n", k);
return -1;
}

if (max_row != k) {
double *temp = matrix[k];
matrix[k] = matrix[max_row];
matrix[max_row] = temp;
}

for (int i = k + 1; i < size; i++) {
double factor = matrix[i][k] / matrix[k][k];
for (int j = k; j < size + 1; j++) {
matrix[i][j] -= factor * matrix[k][j];
}
}
}

for (int i = size - 1; i >= 0; i--) {
```

```
for (int j = i + 1; j < size; j++) {
matrix[i][size] -= matrix[i][j] * matrix[j][size];
}
if (fabs(matrix[i][i]) < EPSILON) {
fprintf(stderr, "Error: Division by zero at row %d\n", i);
return -1;
}
matrix[i][size] /= matrix[i][i];
}

return 0;
}

void *thread_forward_elimination(void *arg) {
thread_data_t *data = (thread_data_t *)arg;
double **matrix = data->matrix;
int size = data->size;
int thread_id = data->thread_id;
int num_threads = data->num_threads;
pthread_barrier_t *barrier = data->barrier;
volatile int *error_flag = data->error_flag;

for (int k = 0; k < size; k++) {
if (*error_flag)
pthread_exit(NULL);

if (thread_id == 0) {
int max_row = k;
double max_val = fabs(matrix[k][k]);
for (int i = k + 1; i < size; i++) {
if (fabs(matrix[i][k]) > max_val) {
max_val = fabs(matrix[i][k]);
max_row = i;
}
}

if (fabs(matrix[max_row][k]) < EPSILON) {
fprintf(stderr, "Error: Matrix is singular at row %d\n", k);
*error_flag = 1;
pthread_barrier_wait(barrier);
pthread_exit(NULL);
}

if (max_row != k) {
double *temp = matrix[k];
matrix[k] = matrix[max_row];
matrix[max_row] = temp;
}
```

```
}

pthread_barrier_wait(barrier);

if (*error_flag)
pthread_exit(NULL);

double pivot = matrix[k][k];
double *pivot_row = matrix[k];

int rows_to_process = size - k - 1;
if (rows_to_process <= 0)
continue;

int active_threads =
(rows_to_process < num_threads) ? rows_to_process : num_threads;
if (thread_id >= active_threads) {
pthread_barrier_wait(barrier</spa
```

Пример работы

./gauss_solver 1 -t

```
=== Performance Analysis ===
Matrix Size | Threads | Sequential (ms) | Parallel (ms) | Speedup | Efficiency
---------------------------------------------------------------------------
Size: 250 | Threads:  1 | Sequential:    19.65 ms | Parallel:    21.80 ms | Speedup:  0.902 |
Efficiency:  0.902
Size: 500 | Threads:  1 | Sequential:   144.20 ms | Parallel:   122.74 ms | Speedup:  1.175 |
Efficiency:  1.175
Size: 1000 | Threads:  1 | Sequential:  1081.16 ms | Parallel:  1007.16 ms | Speedup:  1.073
| Efficiency:  1.073
Size: 1500 | Threads:  1 | Sequential:  3785.61 ms | Parallel:  3431.34 ms | Speedup:  1.103
| Efficiency:  1.103
Size: 2000 | Threads:  1 | Sequential:  9355.43 ms | Parallel:  8415.50 ms | Speedup:  1.112
| Efficiency:  1.112
```

./gauss_solver 2 -t

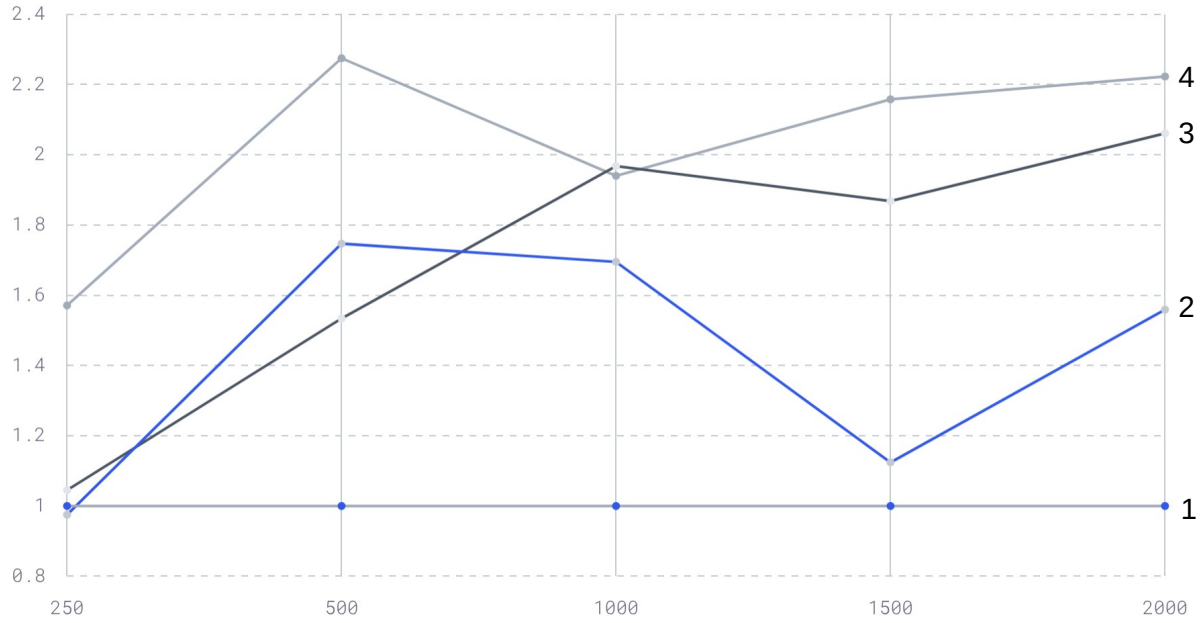
```
=== Performance Analysis ===
Matrix Size | Threads | Sequential (ms) | Parallel (ms) | Speedup | Efficiency
---------------------------------------------------------------------------
Size: 250 | Threads:  2 | Sequential:    22.96 ms | Parallel:    21.97 ms | Speedup:  1.045 |
Efficiency:  0.522
Size: 500 | Threads:  2 | Sequential:   148.72 ms | Parallel:    96.95 ms | Speedup:  1.534 |
```

Efficiency: 0.767
Size: 1000 | Threads: 2 | Sequential: 1098.15 ms | Parallel: 558.02 ms | Speedup: 1.968 | Efficiency: 0.984
Size: 1500 | Threads: 2 | Sequential: 3853.23 ms | Parallel: 2063.15 ms | Speedup: 1.868 | Efficiency: 0.934
Size: 2000 | Threads: 2 | Sequential: 9191.42 ms | Parallel: 4459.64 ms | Speedup: 2.061 | Efficiency: 1.031

./gauss_solver 4 -t

=== Performance Analysis ===
Matrix Size | Threads | Sequential (ms) | Parallel (ms) | Speedup | Efficiency
---
Size: 250 | Threads: 4 | Sequential: 30.21 ms | Parallel: 19.24 ms | Speedup: 1.571 | Efficiency: 0.393
Size: 500 | Threads: 4 | Sequential: 152.83 ms | Parallel: 67.17 ms | Speedup: 2.275 | Efficiency: 0.569
Size: 1000 | Threads: 4 | Sequential: 1114.20 ms | Parallel: 574.33 ms | Speedup: 1.940 | Efficiency: 0.485
Size: 1500 | Threads: 4 | Sequential: 3679.77 ms | Parallel: 1705.14 ms | Speedup: 2.158 | Efficiency: 0.540
Size: 2000 | Threads: 4 | Sequential: 8648.69 ms | Parallel: 3891.06 ms | Speedup: 2.223 | Efficiency: 0.556

Ускорение при использовании 1, 2, 3, 4 потоков с параллельным вычислением в сравнении с последовательным вычислением на одном потоке:



Вывод

В результате выполнения данной лабораторной работы я научился работать с потоками. Программные потоки очень удобно использовать для многозадачности и для большей скорости работы некоторых алгоритмов. Они нужны, когда одновременно происходит несколько действий(и некоторые из них могут блокироваться). Тогда работа с несколькими потоками, которые параллельно выполняют действия, ускоряет программу. В отличии от процессов они быстрее и проще создаются. Еще одно отличие потоков от процессов состоит в том, что потоки делят между собой одно адресное пространство. Однако, это может быть как плюсом, так и минусом, так как один поток, содержащий ошибку, может испортить все остальные. В этом плане процессы безопаснее, так как более изолированы друг от друга. Но для потоков существуют примитивы синхронизации, поэтому проблема решаема. В данной лабораторной работе была продемонстрирована обработка матрицы в многопоточном режиме. В результате анализа программы можно сказать, что быстрее всего она работает при небольшом количестве потоков и очень большом объеме входных данных, что снижает значимость издержек на барьеры и недочеты оптимизации алгоритма. На системные вызовы по работе с потоками уходит часть ресурсов, из-за чего программа может работать с маленькими входными данными медленнее, чем если бы она работала в однопоточном режиме.