Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет имени
Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Робототехника и комплексная автоматизация»
КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ №2

по дисциплине «Разработка программных систем»

Студент:	Турунов Дмитрий Николаевич
Группа:	РК6-63Б
Тип задания:	Лабораторная работа
Тема:	Многопоточное программирование
Вариант:	8

Студент _____ Турунов Д.Н.
подпись, дата Фамилия, И.О.

Преподаватель _____ Козов А.В.
подпись, дата Фамилия, И.О.

Москва, 2024

Содержание

Задание

Разработать, используя средства многопотокового программирования, параллельную программу решения двумерной нестационарной краевой задачи методом конечных разностей с использованием явной вычислительной схемы. Объект моделирования - прямоугольная пластина постоянной толщины. Возможны граничные условия первого и второго рода в различных узлах расчетной сетки. Количество потоков, временной интервал моделирования и количество узлов расчетной сетки - параметры программы. Программа должна демонстрировать ускорение по сравнению с последовательным вариантом. Предусмотреть визуализацию результатов посредством утилиты gnuplot.

Распределение поля температур по пластине описывается уравнением теплопроводности:

$$\frac{dT}{dt} = aT\left[\frac{d^2T}{dx^2} + \frac{d^2T}{dy^2}\right] + gT \tag{1}$$

$$aT = \frac{lambda}{CT^*p} \tag{2}$$

где aT - коэффициент теплопроводности;
$lambda$ - коэффициент теплопроводности среды;
CT - удельная теплоемкость единицы массы;
p - плотность среды;

$$gT = \frac{GT}{CT^*p} \tag{3}$$

где gT - приведенная скорость взаимного превращения тепловой энергии в другие виды энергии, в нашем случае $gT = 0$.
В явной вычислительной схеме МКР для аппроксимации производной температуры по времени, в узле, принадлежащем i-ому и j-ому пространственным слоям, и k-ому временному, используется «разница вперед» (4):

$$\frac{dT}{dt \mid ijk} = \frac{T_{ij}^{k+1} - T_{ij}^{k}}{h_t} \tag{4}$$

где h_t– шаг дискретизации по оси времени.
Для аппроксимации второй производной температуры по пространственной координате x используется «центральная разница» 5, аналогично для координаты y 6:

$$\frac{d^2T}{dx^2 \mid ijk} = \frac{T_{i+1j}^{k} - 2 * T_{ij}^{k} + T_{i-1j}^{k}}{h_x^2} \tag{5}$$

$$\frac{d^2T}{dy^2 \mid ijk} = \frac{T_{ij+1}^{k} - 2 * T_{ij}^{k} + T_{ij-1}^{k}}{h_y^2} \tag{6}$$

где h_x,h_y - шаг дискретизации по пространственной координате.
Тогда алгебраизированное уравнение (7) теплопроводности для узла, принадлежащего

i-ому и j-ому пространственным слоям, и k-ому временному $(gT = 0)$:

$$\frac{T_{ij}^{k+1} - T_{ij}^{k}}{h_t} = aT\left(\frac{T_{i+1j}^{k} - 2 * T_{ij}^{k} + T_{i-1j}^{k}}{h_x^2} + \frac{T_{ij+!}^{k} - 2 * T_{ij}^{k} + T_{ij-1}^{k}}{h_y^2}\right) \tag{7}$$

Такой вид уравнения позволяет в явном виде выразить единственную неизвестную (8):

$$T_{ij}^{k+1} = aTh_t\left[\frac{T_{i+1j}^{k} - 2 * T_{ij}^{k} + T_{i-1j}^{k}}{h_x^2} + \frac{T_{ij+1}^{k} - 2 * T_{ij}^{k} + T_{ij-1}^{k}}{h_y^2}\right] + T_{ij}^{k} \tag{8}$$

Содержание отчета

- Текст задания на лаб. работу
- Описание структуры программы и реализованных способов взаимодействия потоков управления (с рисунком).
- Описание основных используемых структур данных
- Блок-схема программы согласно ГОСТ и пояснения к ней
- Примеры результатов работы программы
- Текст программы с исчерпывающими комментариями

Описание структуры программы и использованных структур данных

- `Thread_param` - структура, содержащая параметры для каждого потока, включая идентификатор потока, размеры области расчета, временной шаг, а также индексы начала и конца обрабатываемой области.

- `pthread_mutex_t` и `pthread_barrier_t` - используются для синхронизации работы потоков и защиты критических секций.

- `solver` - функция, представляющая собой тело потока, в котором выполняется расчет температуры в заданных границах.

- `boundary` - функция для расчета граничных условий.

- `into_file` - функция для записи результатов расчета в файл (опционально).

Процесс работы программы:

1. При запуске программы считываются аргументы командной строки, определяющие количество потоков, временной шаг и размеры расчетной области.

2. Выделяется память для хранения текущего и предыдущего слоев температур, а также инициализируются мьютексы и барьеры.

3. Для каждого потока инициализируются его параметры, включая область ответственности по индексам. Начальные условия записываются в массив предыдущего слоя температур.

4. Создаются потоки, каждый из которых начинает выполнение функции `solver`, рассчитывая температуру в своей области.

5. Внутри каждого потока после завершения расчета на текущем временном шаге потоки синхронизируются с помощью барьера. Затем один из потоков переключает указатели на текущий и предыдущий слои данных, после чего все потоки продолжают расчеты уже с обновленными данными.

6. После выполнения всех расчетов потоки завершают работу. Основной поток программы ожидает завершения всех потоков, после чего освобождает выделенные ресурсы и выводит время выполнения.

На рисунке 1 представлена визуализация работы потоков программы.

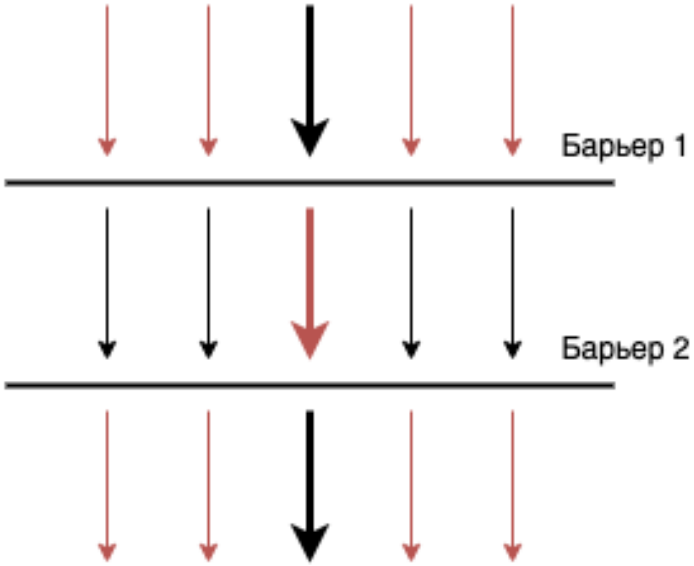

Рис. 1. Визуализации работы потоков. Черные потоки - активные, жирным обозначен главный поток

Блок-схема

На рисунке 2 представлена блок-схема программы.

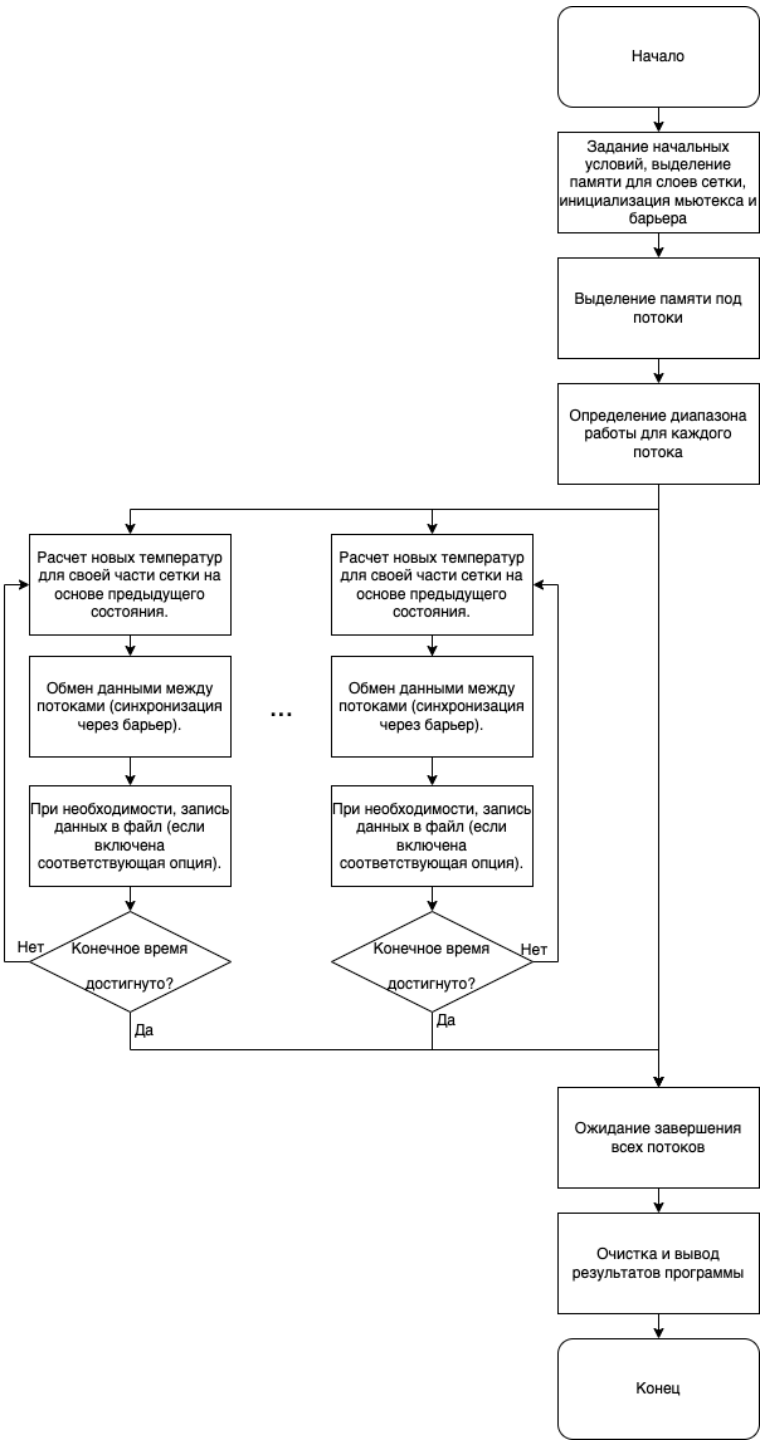

Рис. 2. Блок-схема алгоритма работы программы

Примеры работы программы

На рисунке 3 представлен кадр из визуализации работы программы для сетки 8x8.

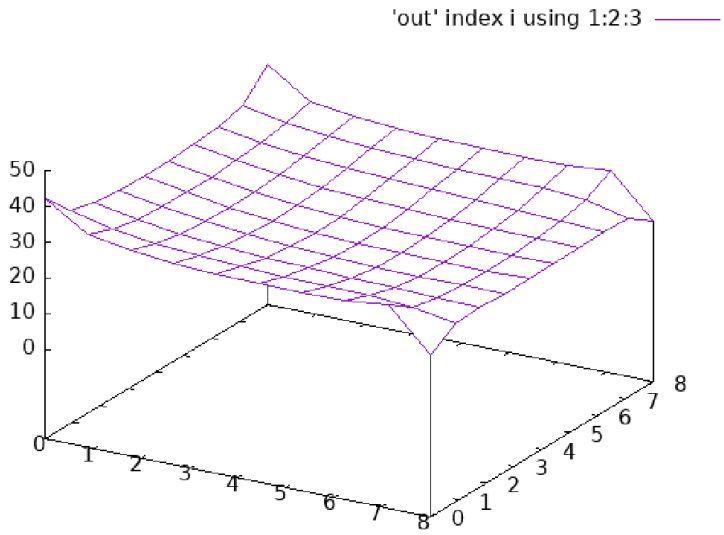

Рис. 3. Пример работы программы для сетки 8x8

На рисунке 4 представлен кадр из визуализации работы программы для сетки 16x16.

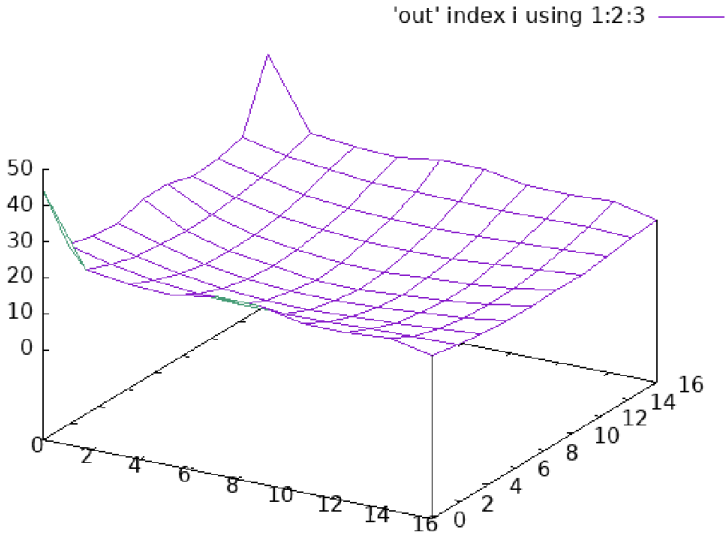

Рис. 4. Пример работы программы для сетки 16x16

Текст программы

На листинге 1 представлен код программы.

```
// Include necessary headers: standard I/O, standard lib, pthreads for
// threading, unistd for various constants, and sys/time for measuring execution
// time
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
// Define constants for the diffusion equation parameters and boundary
// conditions
#define COEF 1
#define dx 2
#define dy 2
#define MTIME 50
#define TOP 0.6
#define BOTTOM 20
#define LEFT 0.4
#define RIGHT 40

// Define a structure for thread parameters, including thread ID, grid
// dimensions, time step, and indexes for the portion of the grid each thread is
// responsible for
typedef struct {
  pthread_t tid;
  int n, m;
  double dt;
  int firstIndexStart, firstIndexEnd, secondIndexStart, secondIndexEnd;
} Thread_param;
// Initialize a mutex and a barrier for thread synchronization
pthread_mutex_t mutx;
pthread_barrier_t barr;
// Declare pointers for storing the previous and current states of the
// temperature grid
Thread_param *threads;
double *prevLayer, *currLayer;

// Optionally include a function to write the grid state to a file, used if
// WRITE_IN_FILE is defined
#ifdef WRITE_IN_FILE
void into_file(FILE *output, double *layer, int N, int M) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
      fprintf(output, "%d %d %lf\n", i, j, layer[N * j + i]);
  fprintf(output, "\n\n");
}
#endif

// Function to calculate boundary conditions based on position in the grid
```

```
49 double boundary(int i, int j, int n, int m, double dt, double T) {
50   double grad_top = 1.0;
51   double grad_left = 1.0;
52   if (i == 0)
53     return T + grad_top * dx;
54   else if (i == n - 1)
55     return BOTTOM;
56   else if (j == 0)
57     return T + grad_left * dy;
58   else if (j == m - 1)
59     return RIGHT;
60   return T; // Should never reach here.
61 }
62
63 // Main function executed by each thread to solve the heat equation over its
64 // part of the grid
65 void *solver(void *arg_p) {
66   Thread_param *param = (Thread_param *)arg_p;
67   for (double t = 0.0 + param->dt; t <= MTIME; t += param->dt) {
68     pthread_barrier_wait(&barr);
69     for (int i = param->firstIndexStart; i <= param->firstIndexEnd; i++)
70       for (int j = param->secondIndexStart; j <= param->secondIndexEnd; j++) {
71         if ((i != 0) && (i != param->n - 1) && (j != 0) &&
72             (j != param->m - 1)) {
73           double x1 = (prevLayer[param->n * j + i - 1] -
74                        2 * prevLayer[param->n * j + i] +
75                        prevLayer[param->n * j + i + 1]) /
76                       (dx * dx);
77           double x2 = (prevLayer[param->n * (j - 1) + i] -
78                        2 * prevLayer[param->n * j + i] +
79                        prevLayer[param->n * (j + 1) + i]) /
80                       (dy * dy);
81           currLayer[param->n * j + i] =
82               param->dt * COEF * (x1 + x2) + prevLayer[param->n * j + i];
83         } else {
84           currLayer[param->n * j + i] = boundary(
85               i, j, param->n, param->m, param->dt, prevLayer[param->n * j + i]);
86         }
87       }
88     pthread_barrier_wait(&barr);
89 #ifdef WRITE_IN_FILE
90     if (pthread_mutex_trylock(&mutx) == 0) {
91       double *interm = prevLayer;
92       prevLayer = currLayer;
93       currLayer = interm;
94       FILE *output = fopen("out", "a");
95       into_file(output, currLayer, param->n, param->m);
96       fclose(output);
97       pthread_mutex_unlock(&mutx);
98     }
99 #endif
100   }
```

```
101   return NULL;
102 }
103
104 int main(int argc, char *argv[]) {
105   // Check for valid command-line arguments and handle various constraints and
106   // errors
107   if (argc != 5) {
108     printf("Invalid argc\n");
109     return -1;
110   }
111   if (atoi(argv[3]) % 8 != 6  atoi(argv[4]) % 8 != 6) {
112     printf("Error: N or M mod 8 != 6\n");
113     return -2;
114   }
115   unsigned int value = (1U << 30) - 2;
116   if ((atoi(argv[3]) * atoi(argv[4])) > value) {
117     printf("Too many nodes\n");
118     return -3;
119   }
120   // Parse command-line arguments for the number of threads, time step, and grid
121   // dimensions
122   int count = atoi(argv[1]), N = atoi(argv[3]) + 2, M = atoi(argv[4]) + 2;
123   double dt = atof(argv[2]);
124   // Record start time for measuring execution time
125   struct timeval start, end;
126   gettimeofday(&start, NULL);
127   // Allocate memory for storing the grid states
128   prevLayer = calloc(N * M, sizeof(double));
129   currLayer = calloc(N * M, sizeof(double));
130   // Initialize pthread attributes, mutex, and barrier
131   pthread_attr_t attr;
132   pthread_mutex_init(&mutx, NULL);
133   pthread_barrier_init(&barr, NULL, count);
134   // Allocate memory for thread parameters and configure each thread's part of
135   // the grid
136   threads = calloc(count, sizeof(Thread_param));
137   // Initialize the grid with boundary conditions
138   for (int i = 0; i < count; i++) {
139     threads[i] = (Thread_param){.n = N,
140                                 .m = M,
141                                 .dt = dt,
142                                 .firstIndexStart = 1 + i * ((N - 2) / count),
143                                 .firstIndexEnd = (i + 1) * ((N - 2) / count),
144                                 .secondIndexStart = 0,
145                                 .secondIndexEnd = M - 1};
146     if (i == 0)
147       threads[i].firstIndexStart--;
148     if (i == count - 1)
149       threads[i].firstIndexEnd++;
150   }
151   for (int i = 0; i < N; i++)
152     for (int j = 0; j < M; j++)
```

```
153       prevLayer[N * j + i] = (i == 0  i == N - 1  j == 0  j == M - 1)
154                                  ? boundary(i, j, N, M, dt, 0.01)
155                                  : 0;
156       // Optionally write the initial grid state to a file
157 #ifdef WRITE_IN_FILE
158   FILE *output = fopen("out", "w");
159   into_file(output, prevLayer, N, M);
160   fclose(output);
161 #endif
162   // Set pthread attributes for system-wide contention scope and joinable state
163   pthread_attr_init(&attr);
164   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
165   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
166   // Create threads to start solving the heat equation
167   for (int i = 0; i < count; i++)
168     pthread_create(&threads[i].tid, &attr, solver, &threads[i]);
169   // Join threads after completion
170   for (int i = 0; i < count; i++)
171     pthread_join(threads[i].tid, NULL);
172   // Clean up: destroy mutex and barrier, print execution time, and free
173   // allocated memory
174   pthread_mutex_destroy(&mutx);
175   pthread_barrier_destroy(&barr);
176   gettimeofday(&end, NULL);
177   long seconds = (end.tv_sec - start.tv_sec);
178   long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);
179   printf("Execution time: %ld seconds, %ld microseconds\n", seconds, micros);
180   // Optionally write a configuration file for gnuplot to visualize the results
181 #ifdef WRITE_IN_FILE
182   FILE *fp = fopen("gnuplot.cfg", "w");
183   if (fp) {
184     char *message;
185     fprintf(fp,
186             "set term gif animate\nset output 'animation.gif'\nset zrange "
187             "[0:50]\nset dgrid3d\nset hidden3d\ndo for [i=0:%d] {\nsplot 'out' "
188             "index i using 1:2:3 with lines\n}",
189             (int)(MTIME / dt) + 1);
190     fclose(fp);
191     printf("gnuplot -persist gnuplot.cfg\n");
192   } else {
193     printf("Error recording gnuplot config\n");
194   }
195 #endif
196   free(prevLayer);
197   free(currLayer);
198   free(threads);
199   return 0;
200 }
```

Листинг 1. Программный код для решения уравнения теплопроводности на двумерной сетке с использованием многопоточности