**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №2**

**по дисциплине «Архитектура параллельных вычислительных систем»**

Тема: «СОРТИРОВКА НА СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ»

|  |  |  |
| --- | --- | --- |
| Студенты гр. 0302 |  | Бикеев М.Р. |
|  |  | Штейнберг Э.Э. |
|  |  | Камоликов В.А. |
|  |  | Билоблоцкий В.В. |
| Преподаватель |  | Костичев С.В. |

Санкт-Петербург

2024

# ВВЕДЕНИЕ

*Цель работы*: Реализация программы «Сортировка простыми вставками (InsertSort)» (вариант 4)

Используемая библиотека – OpenMP.

*Конфигурация системы.*

Программное обеспечение:

* Windows 11 Pro 23H2 22631.4169
* Visual Studio 2022 (v143)

Аппаратное окружение:

* Процессор с тактовой частотой 3.30 GHz (4.2 GHz)
* Количество ядер – 6, потоков – 12.

# Реализация программы

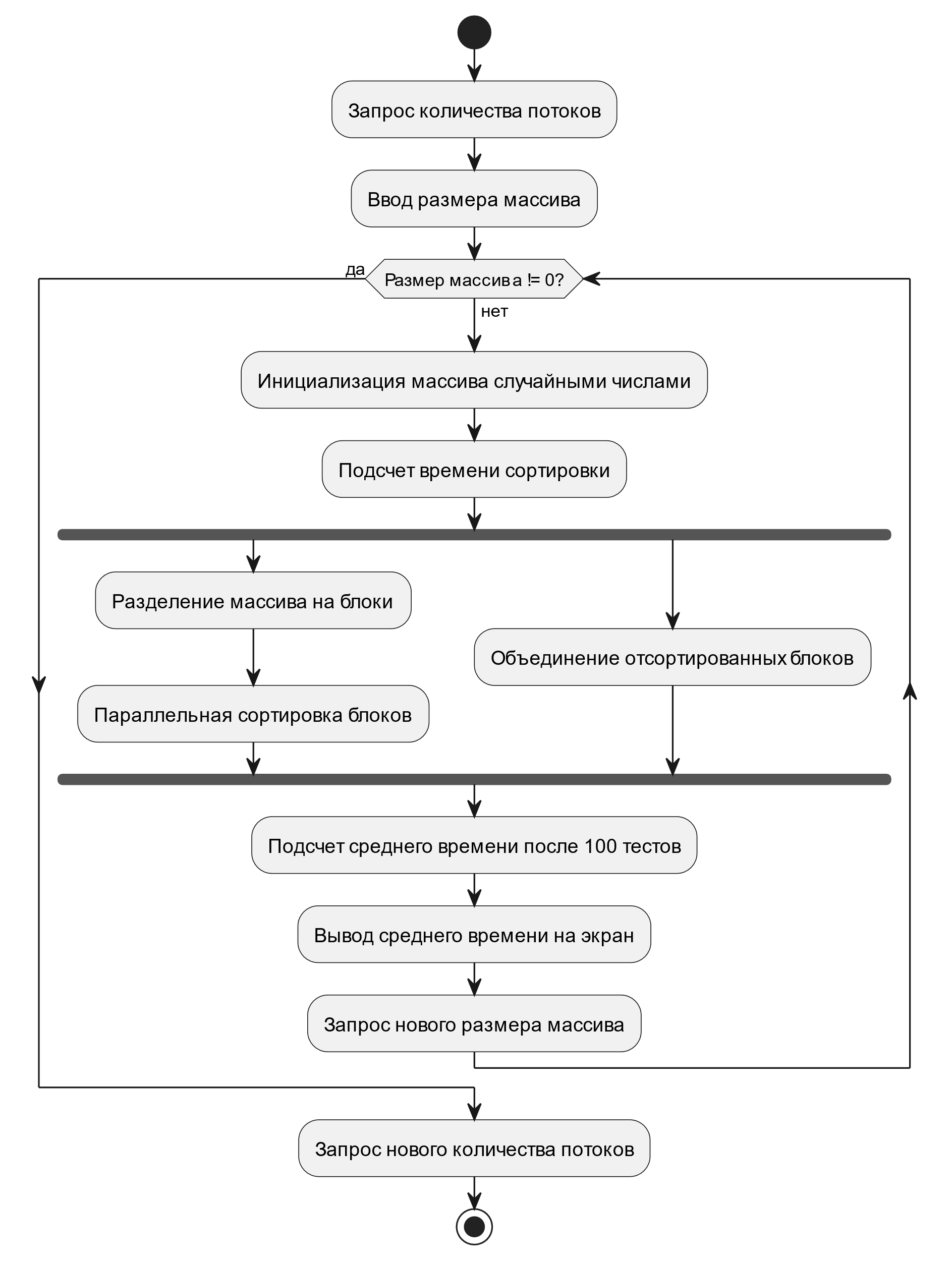
## Описание метода

Алгоритм программы:

* Пользователь вводит количество потоков и размер массива N.
* *Инициализация и заполнение массива:* Массив размером N заполняется случайными числами.
* Массив делится на блоки по количеству потоков.
* *Параллельная сортировка методом вставок:* Каждый блок сортируется методом вставок в своем потоке с использованием OpenMP.
* *После сортировки блоки объединяются*, обеспечивая правильный порядок элементов.
* Сортировка повторяется 100 раз, и вычисляется среднее время.
* *Повторный ввод или смена потоков:* Ввод нового размера массива или смена количества потоков, если введено значение 0.
* *Вывод результатов:* Программа выводит среднее время выполнения для каждого количества потоков и размера массива.
* Локальные результаты умножения (части матрицы C) собираются в главный процесс с помощью MPI\_Gather, выводится время выполнения MPI\_Wtime, и программа завершает работу, вызывая MPI\_Finalize.

## Блок схема алгоритмов

На Рисунке 1 показана блок схема выполнения приложения.

  
Рисунок 1 – Блок схема алгоритма программы

# Результаты работы программы

## Примеры запуска программы

При заполнении данных пользователь может указать как размер массива, так и количество потоков для параллелизма вычислений. После ввода данных пользователь получает среднее время выполнения сортировки за 100 повторений, как показано на Рисунке 2.

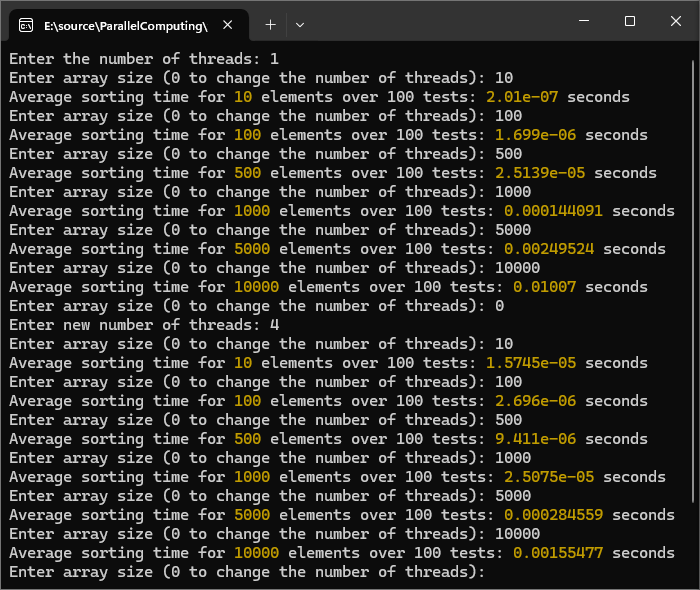
****

Рисунок 2 – Запуск и получение результатов на 4 процессах

## Сравнительная оценка эффективности программы

Рисунок 3 показывает результаты сортировки массивов с разным количеством процессов. Программа становится эффективнее при увеличении числа процессов, особенно на больших массивах, но эффективность падает на малых размерах.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер массива** | **1 процесс** | **2 процесса** | **4 процесса** | **12 процессов** |
| 1 | 0 | 0 | 0 | 0 |
| 10 | 2,01E-07 | 1,02E-05 | 1,57E-05 | 4,93E-05 |
| 100 | 1,70E-06 | 1,97E-06 | 2,70E-06 | 9,13E-06 |
| 500 | 2,51E-05 | 1,94E-05 | 9,41E-06 | 2,28E-05 |
| 1000 | 1,44E-04 | 4,54E-05 | 2,51E-05 | 3,11E-05 |
| 5000 | 0,00249524 | 9,96E-04 | 0,0002846 | 1,87E-04 |
| 10000 | 0,01007 | 0,0033551 | 0,0015548 | 5,06E-04 |

Рисунок 3 – Отношение выполняемого времени в зависимости от числа процессов и размера массива

Ключевые моменты:

* Многопоточность неэффективна для маленьких массивов.
* С увеличением размера массива (от 500 элементов и выше) многопоточность начинает приносить значительные улучшения производительности. Например, для массива в 10000 элементов использование 12 потоков сокращает время выполнения в 20 раз по сравнению с одним потоком.

# Текст программы

Для данной лабораторной работы было разработано консольное приложение включающая в себя 1 файл программного кода.

Файл 1 - *2labSorting.cpp*

#include <iostream>

#include <omp.h>

#include <vector>

#include <cstdlib> // Для функции rand() и srand()

#include <ctime> // Для функции time()

#include <algorithm> // Для std::min

// Функция для сортировки методом вставок

void insertionSort(std::vector<int>& arr, int start, int end) {

for (int i = start + 1; i <= end; i++) {

int key = arr[i];

int j = i - 1;

while (j >= start && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

// Функция для слияния двух отсортированных блоков

void merge(std::vector<int>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

std::vector<int> L(n1), R(n2);

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int i = 0; i < n2; i++)

R[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Функция для многократного слияния блоков

void mergeAllBlocks(std::vector<int>& arr, int num\_threads, int block\_size) {

int n = arr.size();

while (block\_size < n) {

#pragma omp parallel for

for (int i = 0; i < n; i += 2 \* block\_size) {

int left = i;

int mid = std::min(i + block\_size - 1, n - 1);

int right = std::min(i + 2 \* block\_size - 1, n - 1);

if (mid < right) {

merge(arr, left, mid, right);

}

}

block\_size \*= 2; // Увеличиваем размер блоков в два раза

}

}

int main() {

int num\_threads;

std::cout << "Enter the number of threads: ";

std::cin >> num\_threads;

omp\_set\_num\_threads(num\_threads);

int n;

while (true) {

std::cout << "Enter array size (0 to change the number of threads): ";

std::cin >> n;

if (n == 0) {

std::cout << "Enter new number of threads: ";

std::cin >> num\_threads;

omp\_set\_num\_threads(num\_threads);

continue;

}

const int num\_tests = 100; // Количество повторений для тестирования

double total\_time = 0.0; // Переменная для хранения суммарного времени

// Выполнение 100 тестов

for (int test = 0; test < num\_tests; test++) {

srand(static\_cast<unsigned>(time(0)));

std::vector<int> arr(n);

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

arr[i] = rand() % (n + 1);

}

double start\_time = omp\_get\_wtime();

// Определение размера блока

int block\_size = (n + num\_threads - 1) / num\_threads;

// Параллельная сортировка блоков

#pragma omp parallel

{

int thread\_id = omp\_get\_thread\_num();

int start = thread\_id \* block\_size;

int end = std::min(start + block\_size - 1, n - 1);

if (start < n) {

insertionSort(arr, start, end);

}

}

// Слияние отсортированных блоков

mergeAllBlocks(arr, num\_threads, block\_size);

double end\_time = omp\_get\_wtime();

total\_time += (end\_time - start\_time); // Суммируем время каждого теста

}

// Вычисление среднего времени выполнения

double avg\_time = total\_time / num\_tests;

std::cout << "Average sorting time for " <<

"\033[33m" << n << "\033[0m"

<< " elements over " << num\_tests << " tests: " <<

"\033[33m" << avg\_time << "\033[0m"

<< " seconds" << std::endl;

}

return 0;

}

# Выводы

В процессе выполнения лабораторной работы была разработана программа для сортировки простыми вставками с использованием параллельных вычислений с помощью библиотеки OpenMP.

Эффективность программы по сортировке методом простых вставок возрастает с увеличением числа процессов, но многопоточность полезна только на больших объемах данных.