

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ
УНИВЕРСИТЕТ "МИФИ"»**

Институт Интеллектуальных Кибернетических Систем
Кафедра №42 "Криптология и кибербезопасность"
Дисциплина «Параллельное программирование»

Отчет к лабораторной работе № 4
«Технология OpenMP. Особенности настройки»

Выполнил:

студент группы Б22-505
Титов Дмитрий Иванович

Принял:

Куприяшин Михаил Андреевич

Москва
2024 год

Цель работы

Изучить основные настройки среды OpenMP и связанные с ними возможности.

1. Рабочая среда

Процессор - amd Ryzen 7 7840HS (2023) 8 ядер и 16 потоков

Оперативная память - 16GB DDR5

ОС - Ubuntu 24.04.1 LTS 64-бит

Среда разработки - VS Code, CMake 3.28.3, GCC 13.2.0

Версия OpenMP - 4.5 201511

Ноутбук был на зарядке

2. Выполнение заданий 1-7

2.1. При помощи переменной предпроцессора OPENMP определить дату принятия используемого стандарта OpenMP. Вывести на экран версию стандарта и дату принятия.

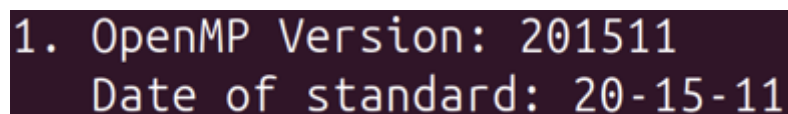
Описание: В данном фрагменте кода используется макрос `_OPENMP` для вывода версии стандарта OpenMP и даты его принятия в формате год-месяц-день. Если OpenMP не поддерживается, выводится соответствующее сообщение.

Фрагмент кода:

```
#ifdef _OPENMP
    printf("1. OpenMP Version: %d\n", _OPENMP);
    printf("  Date of standard: %d-%d-%d\n",
        _OPENMP / 10000,
        (_OPENMP % 10000) / 100,
        _OPENMP % 100);
#else
    printf("1. OpenMP is not supported on this system.\n");
#endif
```

Листинг 2.1. Фрагмент task1-7.c

Результат:



```
1. OpenMP Version: 201511
  Date of standard: 20-15-11
```

Рис. 2.1. Вывод программы task1-7.c для задания 1

Здесь 201511 указывает на стандарт OpenMP 4.5, утверждённый в ноябре 2015 года.

2.2. Использовать функции `omp_get_num_procs()` и `omp_get_max_threads()` для определения числа доступных процессоров и потоков. Вывести результат.

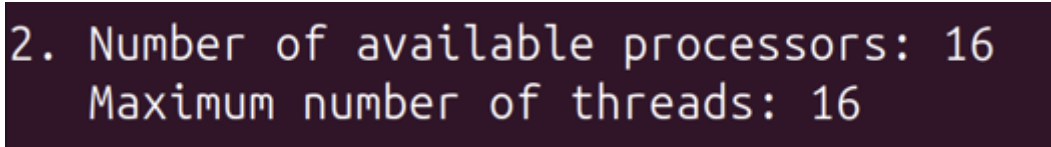
Описание: Функции `omp_get_num_procs()` и `omp_get_max_threads()` используются для определения количества доступных процессоров и максимального числа потоков, которые могут быть использованы в текущей среде.

Фрагмент кода:

```
int num_procs = omp_get_num_procs();
int max_threads = omp_get_max_threads();
printf("\n2. Number of available processors: %d\n", num_procs);
printf("  Maximum number of threads: %d\n", max_threads);
```

Листинг 2.2. Фрагмент `tas1-7.c`

Результат:



```
2. Number of available processors: 16
Maximum number of threads: 16
```

Рис. 2.2. Вывод программы `task1-7.c` для задания 2

Система имеет 16 логических процессоров, а OpenMP поддерживает до 16 потоков одновременно.

2.3. Выяснить и описать назначение опции `dynamic`. Определить её состояние при помощи функции `omp_get_dynamic()`. Вывести результат.

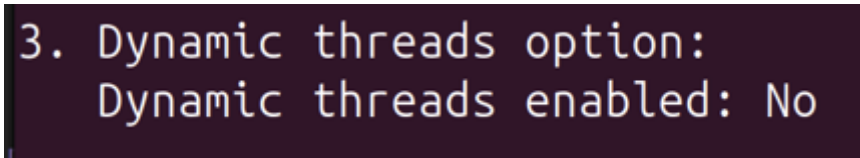
Описание: Опция `dynamic` позволяет динамически изменять количество потоков в параллельных областях во время выполнения программы. Для определения состояния опции используется функция `omp_get_dynamic()`, которая возвращает 1, если динамические потоки включены, и 0 в противном случае.

Фрагмент кода:

```
int dynamic_state = omp_get_dynamic();
printf("\n3. Dynamic threads option:\n");
printf("  Dynamic threads enabled: %s\n", dynamic_state ? "Yes" : "No");
```

Листинг 2.3. Фрагмент lab4task1-7.c

Результат:



```
3. Dynamic threads option:
  Dynamic threads enabled: No
```

Рис. 2.3. Вывод программы lab4task1-7.c для задания 3

Опция динамического создания потоков отключена (No).

2.4. Определить разрешение таймера при помощи `omp_get_wtick()`. Вывести результат с указанием единицы измерения.

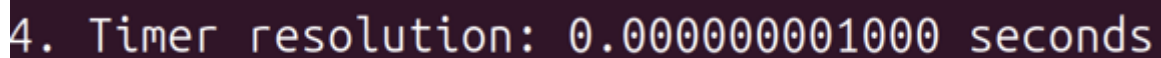
Описание: Разрешение таймера в OpenMP определяется функцией `omp_get_wtick()`. Эта функция возвращает минимальное время, которое может быть измерено таймером, в секундах.

Фрагмент кода:

```
double timer_resolution = omp_get_wtick();  
printf("\n4. Timer resolution: %.12f seconds\n", timer_resolution);
```

Листинг 2.4. Фрагмент lab4task1-7.c

Результат:



```
4. Timer resolution: 0.000000001000 seconds
```

Рис. 2.4. Вывод программы lab4task1-7.c для задания 4

Разрешение таймера OpenMP составляет **1 наносекунду** (или 0.000000001 секунд).

2.5. Уточнить особенности работы со вложенными параллельными областями в OpenMP. Определить текущие настройки среды при помощи функций `omp_get_nested()` и `omp_get_max_active_levels()` и вывести на экран.

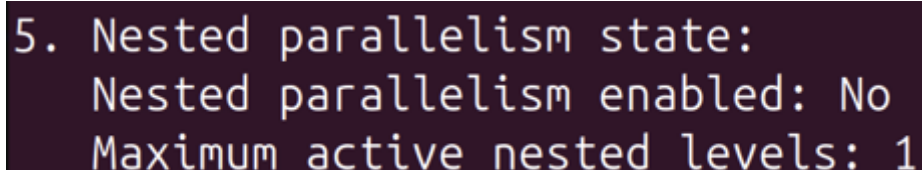
Описание: Функции `omp_get_nested()` и `omp_get_max_active_levels()` используются для проверки состояния вложенных параллельных областей. Первая возвращает 1, если вложенные области разрешены, и 0 в противном случае. Вторая указывает максимальное количество активных вложенных уровней.

Фрагмент кода:

```
int nested_state = omp_get_nested();
int max_active_levels = omp_get_max_active_levels();
printf("\n5. Nested parallelism state:\n");
printf("  Nested parallelism enabled: %s\n", nested_state ? "Yes" : "No");
printf("  Maximum active nested levels: %d\n", max_active_levels);
```

Листинг 2.5. Фрагмент lab4task1-7.c

Результат:

The image shows a terminal window with a dark background and light-colored text. It displays the output of a program, which is a numbered list of three items: '5. Nested parallelism state:', 'Nested parallelism enabled: No', and 'Maximum active nested levels: 1'.

```
5. Nested parallelism state:
  Nested parallelism enabled: No
  Maximum active nested levels: 1
```

Рис. 2.5. Вывод программы lab4task1-7.c для задания 5

Вложенный параллелизм отключён (No), и **допускается только один уровень параллельных областей.**

2.6. Уточнить особенности распределения нагрузки в среде OpenMP. Получить текущие настройки среды с использованием функции `omp_get_schedule()` и вывести их на экран.

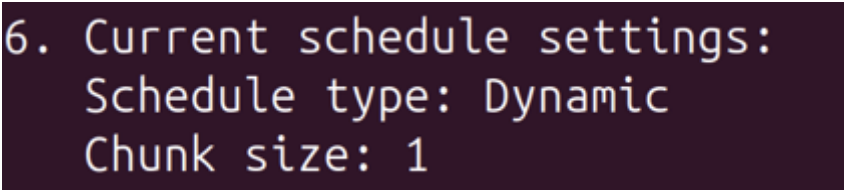
Описание: Функция `omp_get_schedule()` возвращает текущий тип распределения нагрузки (`static`, `dynamic`, `guided`, `auto`) и размер чанка. Это позволяет оптимизировать распределение работы между потоками.

Фрагмент кода:

```
omp_sched_t schedule_type;
int chunk_size;
omp_get_schedule(&schedule_type, &chunk_size);
printf("\n6. Current schedule settings:\n");
printf("  Schedule type: ");
switch (schedule_type) {
    case omp_sched_static: printf("Static\n"); break;
    case omp_sched_dynamic: printf("Dynamic\n"); break;
    case omp_sched_guided: printf("Guided\n"); break;
    case omp_sched_auto:   printf("Auto\n"); break;
    default:               printf("Unknown\n"); break;
}
printf("  Chunk size: %d\n", chunk_size);
```

Листинг 2.6. Фрагмент lab4task1-7.c

Результат:



```
6. Current schedule settings:
  Schedule type: Dynamic
  Chunk size: 1
```

Рис. 2.6. Вывод программы lab4task1-7.c для задания 6

Тип расписания: **Dynamic** (динамическое распределение), а размер чанка равен **1**.

2.7. Разработать пример вычислительного алгоритма, использующего механизм явных блокировок (omp set lock()). Обосновать необходимость использования блокировки.

Описание: В данном примере используется явная блокировка (omp_set_lock() и omp_unset_lock()) для защиты общего ресурса от одновременного доступа несколькими потоками. Блокировка гарантирует, что только один поток может обновлять ресурс в любой момент времени.

Фрагмент кода:

```
omp_lock_t lock;
omp_init_lock(&lock);

int shared_resource = 0;

#pragma omp parallel num_threads(4)
{
    int thread_id = omp_get_thread_num();
    omp_set_lock(&lock); // Захватываем блокировку
    printf("Thread %d: Lock acquired, updating shared resource.\n", thread_id);
    shared_resource += 1; // Обновление общего ресурса
    printf("Thread %d: Updated shared resource to %d.\n", thread_id, shared_resource);
    omp_unset_lock(&lock); // Освобождаем блокировку
}

omp_destroy_lock(&lock);
printf("Final value of shared resource: %d\n", shared_resource);
```

Листинг 2.7. Фрагмент lab4task1-7.c

Обоснование необходимости блокировки: Без использования блокировки несколько потоков могли бы одновременно читать и изменять общий ресурс, что привело бы к некорректным результатам (условие гонки). Блокировка предотвращает этот сценарий.

Результат:

```
7. Example of computation using explicit locks:  
Thread 0: Lock acquired, updating shared resource.  
Thread 0: Updated shared resource to 1.  
Thread 2: Lock acquired, updating shared resource.  
Thread 2: Updated shared resource to 2.  
Thread 1: Lock acquired, updating shared resource.  
Thread 1: Updated shared resource to 3.  
Thread 3: Lock acquired, updating shared resource.  
Thread 3: Updated shared resource to 4.  
Final value of shared resource: 4
```

Рис. 2.7. Вывод программы lab4task1-7.c для задания 7

Каждый из четырёх потоков последовательно захватывает блокировку, обновляет общий ресурс, а затем освобождает блокировку. Итоговое значение общего ресурса: 4.

3. Задание 8

Для одного из алгоритмов, реализованных в предыдущих лабораторных работах, повторить вычислительный эксперимент для разных типов разделения нагрузки и размеров фрагмента (опция `schedule` директивы `parallel`). Сравнить результаты, объяснить наличие/отсутствие разницы.

Для данного задания был выбран алгоритм параллельного поиска элемента в массиве (лабораторная работа #2). Для каждого из режимов `static`, `dynamic` и `guided` программа должна сравнить среднее время выполнения, ускорения и эффективности для разных чанков (5, 10, 25, 50), записать результаты в соответствующие файлы, затем построить графики.

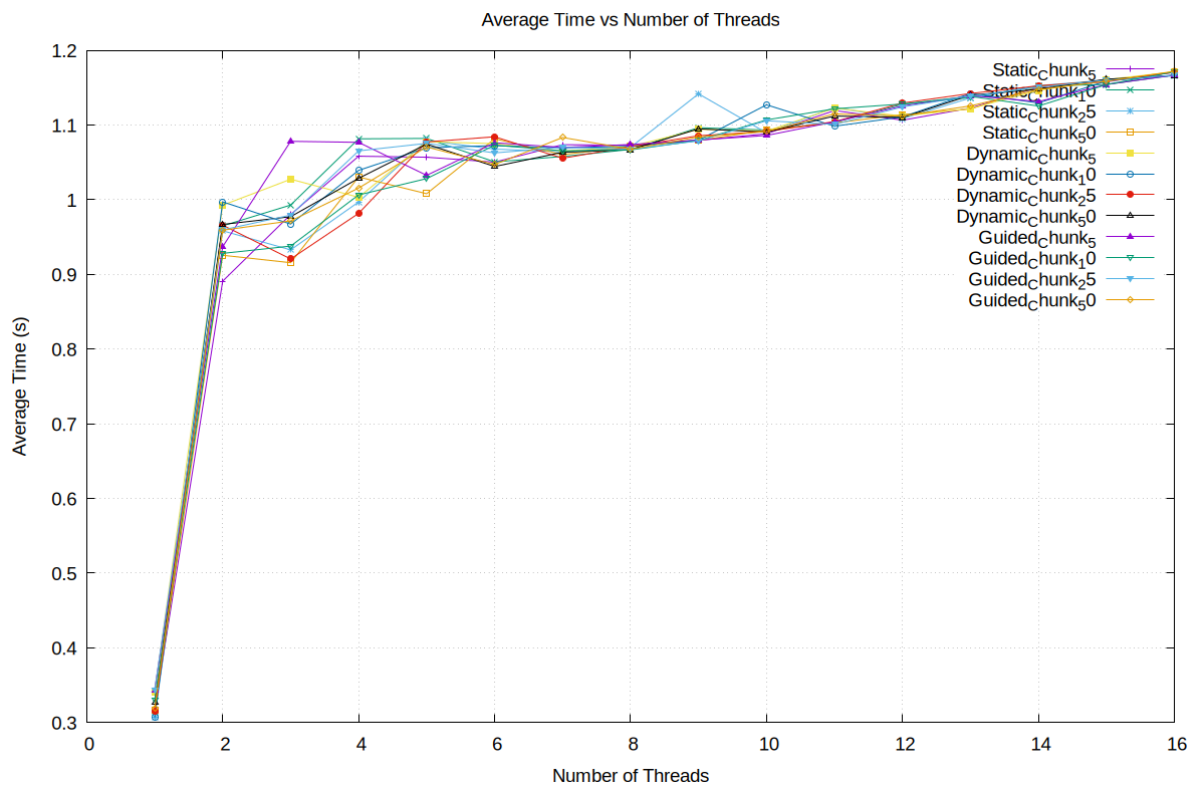


Рис. 3.1. График зависимости среднего времени от числа потоков для разных чанков

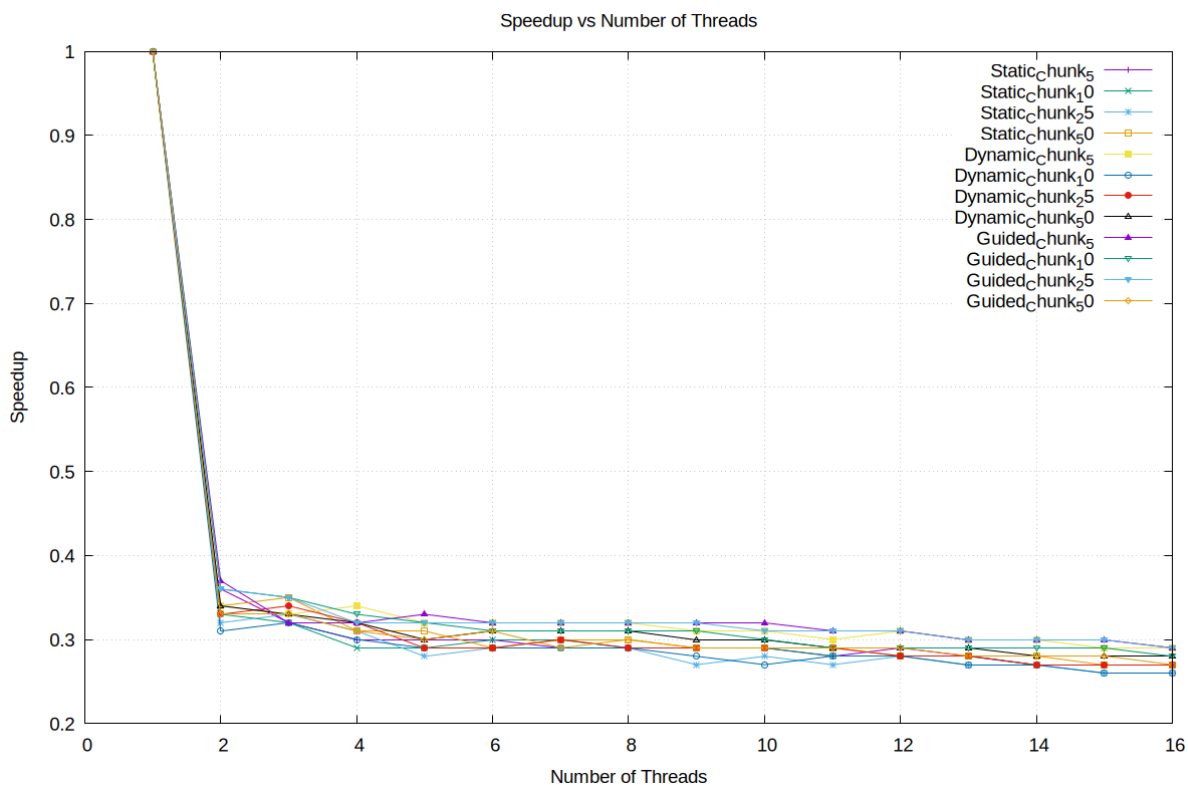


Рис. 3.2. График зависимости ускорения от числа потоков для разных чанков

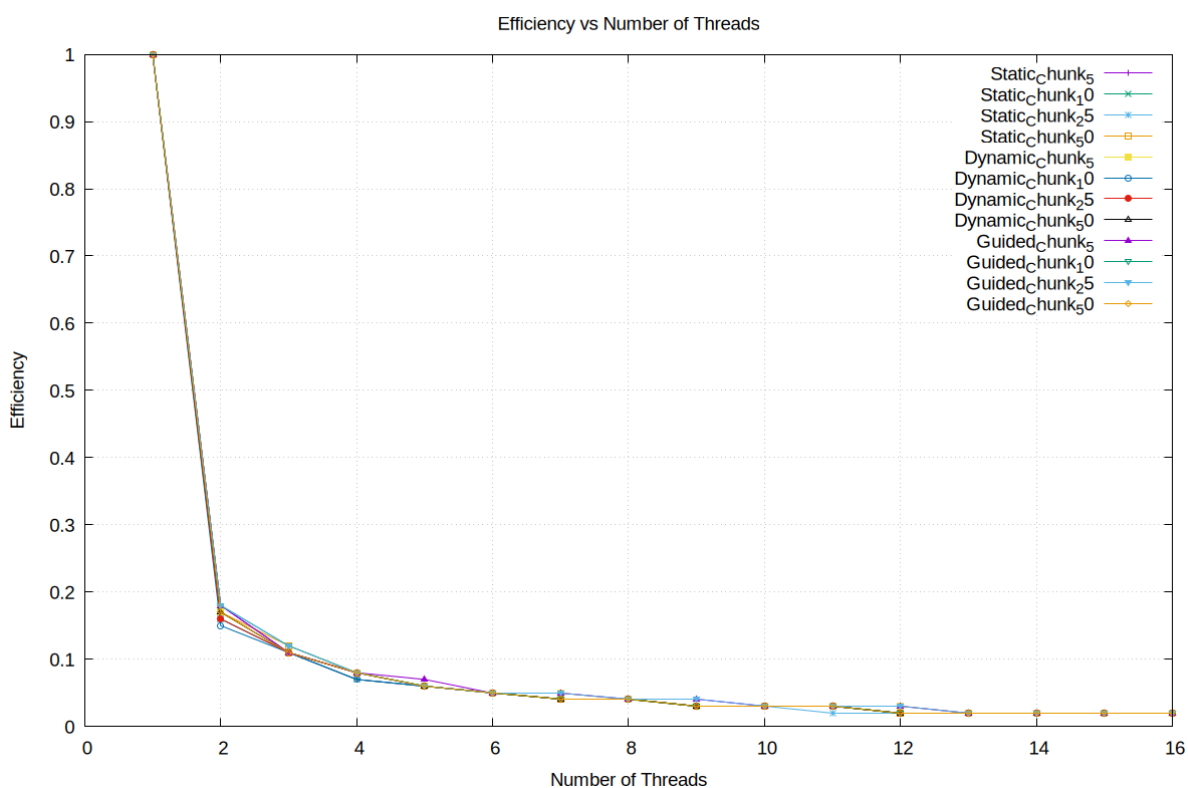


Рис. 3.3. График зависимости эффективности от числа потоков для разных чанков

Анализ и выводы

1. График изменения времени выполнения (Avg Time)

Во всех случаях время выполнения с увеличением числа потоков возрастает, что является необычным результатом. Ожидаемо, при увеличении числа потоков время работы должно сокращаться, но в данном случае, наоборот, наблюдается рост или плато.

Для всех планировок (static, dynamic, guided) с увеличением числа потоков время выполнения сначала сокращается (для 2-3 потоков), но затем стабилизируется или увеличивается. Например, в планировке **dynamic** с размером блока 5:

- Для 1 потока: 0.340740 с.
- Для 2 потоков: 0.992607 с (в 2.91 раза хуже).
- Для 3 потоков: 1.027400 с (еще хуже).

Аналогичная тенденция прослеживается и для других планировок и размеров блоков. Это указывает на **неэффективность параллелизации**.

2. Ускорение (Speedup)

Ускорение, которое должно увеличиваться с ростом числа потоков, также не демонстрирует ожидаемого поведения:

- В планировке **dynamic** с размером блока 5:
 - Для 1 потока: Speedup = 1.00.
 - Для 2 потоков: Speedup = 0.34.
 - Для 3 потоков: Speedup = 0.33.Это явный признак того, что использование многопоточности в данном случае не даёт ожидаемого эффекта.

Для всех сочетаний планировок, ускорение сначала уменьшается и стабилизируется, что указывает на:

- **Перегрузка системы** (слишком много потоков относительно объема работы, или системные ресурсы не справляются с повышенной нагрузкой).
- **Низкая эффективность параллелизации** задачи.

3. Эффективность (Efficiency)

Эффективность работы системы с параллельными потоками везде также снижена:

- В **dynamic** с размером блока 5:
 - Для 1 потока: Efficiency = 1.00.
 - Для 2 потоков: Efficiency = 0.17.
 - Для 3 потоков: Efficiency = 0.11.

Эффективность стремится к нулю по мере увеличения количества потоков, что является ещё одним свидетельством того, что многопоточность здесь не работает должным образом.

4. Планировки

- **Static** планировка показывает довольно схожие результаты с **dynamic** и **guided** планировками, при этом в некоторых случаях она работает чуть быстрее при меньшем числе потоков, но затем поведение аналогично: время увеличивается, ускорение и эффективность падают.
- **Dynamic** и **Guided** планировки показывают аналогичные результаты, с небольшими различиями в начальном времени работы, но общий тренд одинаков.

5. Размеры блоков (Chunk Size)

Изменение размера блоков тоже не даёт значительного улучшения:

- Размер блока влияет на распределение работы между потоками, но результаты по времени и эффективности остаются в целом одинаковыми.
- При увеличении размера блока до 50, различия между результатами незначительны, что подтверждает, что параллелизация не была эффективно использована для данной задачи.

Возможные причины плохих результатов

1. **Параллельность не подходит для данной задачи:** Данный алгоритм — поиск целевого значения в массиве, где после нахождения первого целевого значения нужно завершить выполнение всех потоков. Это создаёт проблемы с синхронизацией:
 - **Флаг "found"** (найден ли элемент) используется глобально, что приводит к блокировке всех потоков после первого успешного поиска. Это снижает параллельную производительность.
 - Использование конструкции `#pragma omp critical` для обновления глобального индекса приводит к дополнительным накладным расходам, что препятствует эффективному распределению работы между потоками.
2. **Перегрузка из-за высокой контекстной синхронизации:** Постоянная синхронизация между потоками (через переменные типа `found`, использование `#pragma omp critical` и атомарных операций) снижает выгоду от многозадачности.
3. **Размер задачи слишком мал для использования множества потоков:** Для задачи с одним целевым значением в массиве использование множества потоков может быть нецелесообразным, так как количество операций, которое каждый поток может выполнить, слишком мало, чтобы оправдать дополнительные накладные расходы на управление потоками.

4. **Перегрузка памяти:** Важно учитывать, что многопоточность требует дополнительных системных ресурсов, таких как память, что также может влиять на производительность.

Заключение

Результаты показывают, что алгоритм поиска целевого значения в массиве плохо подходит для параллелизации с использованием OpenMP, особенно при большом числе потоков. Основные причины — это избыточная синхронизация и глобальные флаги, которые снижают производительность при использовании многопоточности. В таких случаях можно ожидать, что увеличение числа потоков приведёт не к ускорению, а к ухудшению производительности из-за накладных расходов на управление потоками.

Для улучшения параллелизации можно:

- Использовать более эффективные методы синхронизации или полностью изменить подход к поиску.
- Разработать алгоритм, который минимизирует блокировки и использует локальные результаты для каждого потока.

Таким образом, данный алгоритм неэффективен для параллельной реализации с использованием OpenMP.

4. Приложение

4.1. Листинг заданий 1-7

```
#include <stdio.h>
#include <omp.h>
#include <time.h>

void compute_with_locks(); // Объявление функции для пункта 7

int main() {
    // 1. Определение версии OpenMP и даты стандарта
    #ifdef _OPENMP
        printf("1. OpenMP Version: %d\n", _OPENMP);
        printf("   Date of standard: %d-%d-%d\n",
            _OPENMP / 10000,
            (_OPENMP % 10000) / 100,
            _OPENMP % 100);
    #else
```

```

    printf("1. OpenMP is not supported on this system.\n");
#endif

// 2. Определение числа процессоров и максимального числа потоков
int num_procs = omp_get_num_procs();
int max_threads = omp_get_max_threads();
printf("\n2. Number of available processors: %d\n", num_procs);
printf("   Maximum number of threads: %d\n", max_threads);

// 3. Опция dynamic и её состояние
int dynamic_state = omp_get_dynamic();
printf("\n3. Dynamic threads option:\n");
printf("   Dynamic threads enabled: %s\n", dynamic_state ? "Yes" : "No");

// 4. Разрешение таймера
double timer_resolution = omp_get_wtick();
printf("\n4. Timer resolution: %.12f seconds\n", timer_resolution);

// 5. Вложенные параллельные области
int nested_state = omp_get_nested();
int max_active_levels = omp_get_max_active_levels();
printf("\n5. Nested parallelism state:\n");
printf("   Nested parallelism enabled: %s\n", nested_state ? "Yes" : "No");
printf("   Maximum active nested levels: %d\n", max_active_levels);

// 6. Текущие настройки распределения нагрузки
omp_sched_t schedule_type;
int chunk_size;
omp_get_schedule(&schedule_type, &chunk_size);
printf("\n6. Current schedule settings:\n");
printf("   Schedule type: ");
switch (schedule_type) {
    case omp_sched_static: printf("Static\n"); break;
    case omp_sched_dynamic: printf("Dynamic\n"); break;
    case omp_sched_guided: printf("Guided\n"); break;
    case omp_sched_auto: printf("Auto\n"); break;
    default: printf("Unknown\n"); break;
}
printf("   Chunk size: %d\n", chunk_size);

// 7. Пример с блокировками
printf("\n7. Example of computation using explicit locks:\n");
compute_with_locks();

```



```

    return 0;
}

void compute_with_locks() {
    omp_lock_t lock;
    omp_init_lock(&lock);

    int shared_resource = 0;

    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        omp_set_lock(&lock); // Захватываем блокировку
        printf("Thread %d: Lock acquired, updating shared resource.\n", thread_id);
        shared_resource += 1; // Обновление общего ресурса
        printf("Thread %d: Updated shared resource to %d.\n", thread_id, shared_resource);
        omp_unset_lock(&lock); // Освобождаем блокировку
    }

    omp_destroy_lock(&lock);
    printf("Final value of shared resource: %d\n", shared_resource);
}

```

Листинг 4.1. Код файла lab4task1-7.c

4.2. Листинг задания 8

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char** argv)
{
    const int count = 100000000;    ///< Number of elements in each array
    const int seeds[10] = {920214, 123456, 654321, 789101, 112131,
                          141516, 171819, 202122, 232425, 262728}; ///< Array seeds
    const int target = 16;          ///< Target value to search for
    const int max_threads = 16;     ///< Maximum number of threads to test
    const int num_arrays = 10;      ///< Number of random arrays to generate

    // Allocate memory for storing multiple arrays
    int** arrays = (int**)malloc(num_arrays * sizeof(int*));
    for (int i = 0; i < num_arrays; i++) {
        arrays[i] = (int*)malloc(count * sizeof(int));
        srand(seeds[i]); ///< Initialize random seed for array generation
        for (int j = 0; j < count; j++) {
            arrays[i][j] = rand(); ///< Fill array with random values
        }
    }

    // Scheduling policies and chunk sizes
    char* schedules[3] = {"static", "dynamic", "guided"}; ///< Schedule types
    int chunks[] = {5, 10, 25, 50};          ///< Different chunk sizes

    double linear_avg_time = 0.0; ///< Linear time with threads = 1

    // ** Unified Schedule Calculation for static, dynamic, and guided **
    for (int s = 0; s < 3; s++) { // Loop over scheduling policies
        for (int chunk = 0; chunk < 4; chunk++) { // Loop over chunk sizes
            char filename[100];
            sprintf(filename, "/home/dt/ParProg/lab4/task8/performance_%s_chunk_%d.txt",
                schedules[s], chunks[chunk]);

            FILE* file = fopen(filename, "w");
            if (!file) {
                printf("Failed to open file for %s schedule with chunk size %d\n", schedules[s],
                    chunks[chunk]);
                return 1;
            }
        }
    }
}
```

```

fprintf(file, "# Schedule: %s, Chunk Size: %d\n", schedules[s], chunks[chunk]);
fprintf(file, "# Threads\tAvg Time (s)\tSpeedup\tEfficiency\n");

// Test thread counts from 1 to max_threads
for (int threads = 1; threads <= max_threads; threads++) {
    omp_set_num_threads(threads); ///< Set number of OpenMP threads

    double avg_time = 0.0; ///< Average time for parallel search

    for (int arr = 0; arr < num_arrays; arr++) { ///< Loop through arrays
        int index = -1; ///< Global index of target value
        int found = 0; ///< Flag to indicate if target was found

        double start_time = omp_get_wtime(); ///< Start time for parallel search

        #pragma omp parallel
        {
            int local_index = -1; ///< Local index of the target value

            // Use appropriate scheduling policy with chunk size
            #pragma omp for schedule(runtime)
            for (int i = 0; i < count; i++) {
                if (found) continue; ///< Stop if target is already found

                if (arrays[arr][i] == target) { ///< Check for target value
                    local_index = i;

                    #pragma omp atomic write
                    found = 1; ///< Set the global found flag
                }
            }

            // Update the global index in a critical section
            #pragma omp critical
            {
                if (local_index != -1 && (index == -1 || local_index < index)) {
                    index = local_index; ///< Update global index
                }
            }
        }

        avg_time += (omp_get_wtime() - start_time); ///< Total search time
    }

    avg_time /= num_arrays; ///< Average time for the current test
}

```

```

// For one thread, save the time as the reference (linear_avg_time)
if (threads == 1) {
    linear_avg_time = avg_time;
}

// Compute speedup and efficiency
double speedup = linear_avg_time / avg_time;
double efficiency = speedup / threads;

// Write results in tabular format
fprintf(file, "%d\t%.6f\t%.2f\t%.2f\n",
        threads, avg_time, speedup, efficiency);

// Print results to console for real-time monitoring
printf("Schedule: %s, Chunk: %d, Threads: %d, Avg Time: %.6f s, Speedup: %.2f,
Efficiency: %.2f\n",
        schedules[s], chunks[chunk], threads, avg_time, speedup, efficiency);
}

fclose(file);
printf("Results for %s schedule, chunk size %d saved to %s\n", schedules[s],
chunks[chunk], filename);
}
}

// Free allocated memory
for (int i = 0; i < num_arrays; i++) {
    free(arrays[i]);
}
free(arrays);

printf("All results saved successfully.\n");
return 0;
}

```

Листинг 4.2. Код файла lab4task8.c

4.3. Листинг GNUPlot-Script

```

# Set output format
set terminal pngcairo size 1200, 800 enhanced font "Arial, 14"
set style data linespoints

```

```

# Define input files and corresponding labels
files = "performance_static_chunk_5.txt performance_static_chunk_10.txt
performance_static_chunk_25.txt performance_static_chunk_50.txt
performance_dynamic_chunk_5.txt performance_dynamic_chunk_10.txt
performance_dynamic_chunk_25.txt performance_dynamic_chunk_50.txt
performance_guided_chunk_5.txt performance_guided_chunk_10.txt
performance_guided_chunk_25.txt performance_guided_chunk_50.txt"
labels = "Static_Chunk_5 Static_Chunk_10 Static_Chunk_25 Static_Chunk_50
Dynamic_Chunk_5 Dynamic_Chunk_10 Dynamic_Chunk_25 Dynamic_Chunk_50
Guided_Chunk_5 Guided_Chunk_10 Guided_Chunk_25 Guided_Chunk_50"

# Plot Avg Time vs Threads
set output "avg_time_vs_threads.png"
set title "Average Time vs Number of Threads"
set xlabel "Number of Threads"
set ylabel "Average Time (s)"
set grid
plot for [i=1:words(files)] word(files, i) using 1:2 title word(labels, i) lc i

# Plot Speedup vs Threads
set output "speedup_vs_threads.png"
set title "Speedup vs Number of Threads"
set xlabel "Number of Threads"
set ylabel "Speedup"
set grid
plot for [i=1:words(files)] word(files, i) using 1:3 title word(labels, i) lc i

# Plot Efficiency vs Threads
set output "efficiency_vs_threads.png"
set title "Efficiency vs Number of Threads"
set xlabel "Number of Threads"
set ylabel "Efficiency"
set grid
plot for [i=1:words(files)] word(files, i) using 1:4 title word(labels, i) lc i

```

Листинг 4.3. Код файла plot_performance.gnuplot