3.1.1. Вычисление числа π

Описание алгоритма

Этот алгоритм использует многопоточность для приближенного вычисления числа π с помощью метода Монте-Карло. Основная идея заключается в генерации случайных точек в квадрате и подсчете числа точек, попадающих внутрь четверти круга, чтобы оценить значение π. Алгоритм выполняется в несколько потоков, что позволяет ускорить вычисления. Ниже описаны ключевые шаги алгоритма.

1. **Глобальные переменные**:
   - `global_hits`: хранит общее количество точек, попадающих в круг.
   - `total_attempts`: общее количество попыток (сгенерированных точек).
   - `mutex_lock`: мьютекс для обеспечения потокобезопасности при обновлении общего счетчика `global_hits`.
2. **Функция** `calculate_pi`:
   - Инициализирует генератор случайных чисел с уникальным значением, основанным на идентификаторе потока.
   - В цикле генерирует заданное количество случайных точек (координаты `x` и `y`).
   - Проверяет, попадает ли каждая точка в четверть круга (условие: x^2+y^2≤1). Если точка попадает, увеличивает счетчик `local_hits`.
   - После завершения подсчетов блокирует мьютекс и добавляет `local_hits` к общему числу попаданий `global_hits`, затем разблокирует мьютекс.
3. **Функция** `create_threads`:
   - Создает заданное количество потоков, передавая им указатель на количество попыток, которые каждый поток должен выполнить.
4. **Функция** `join_threads`:

- Ожидает завершения всех созданных потоков, обеспечивая, что основной поток не завершится, пока не завершатся все дочерние.

5. **Функция** `initialize_mutex`:
   - Инициализирует мьютекс для обеспечения потокобезопасного доступа к переменной `global_hits`.
6. **Функция** `destroy_mutex`:
   - Уничтожает мьютекс после завершения работы потоков.
7. **Функция** `estimate_pi`:
   - Вычисляет приближенное значение числа π по заданной формуле.
   - Выводит полученное значение на экран.
8. **Функция** `log_results_to_file`:
   - Записывает результаты выполнения (число потоков, количество попыток и время выполнения) в файл `results.txt`. Это облегчает дальнейший анализ результатов.
9. **Основная программа** (`main`):
   - Проверяет, что переданы корректные аргументы (число потоков и количество попыток).
   - Инициализирует потоки и запускает их выполнение.
   - После завершения потоков, вызывает функции для оценки числа π и записи результатов в файл.
   - Замеряет и выводит время выполнения программы.

Оценка времени и эффективности работы

После неоднократного запуска программы с различными значениями у меня получились следующие результаты:

Кол-во потоков, кол-во точек, время выполнения программы

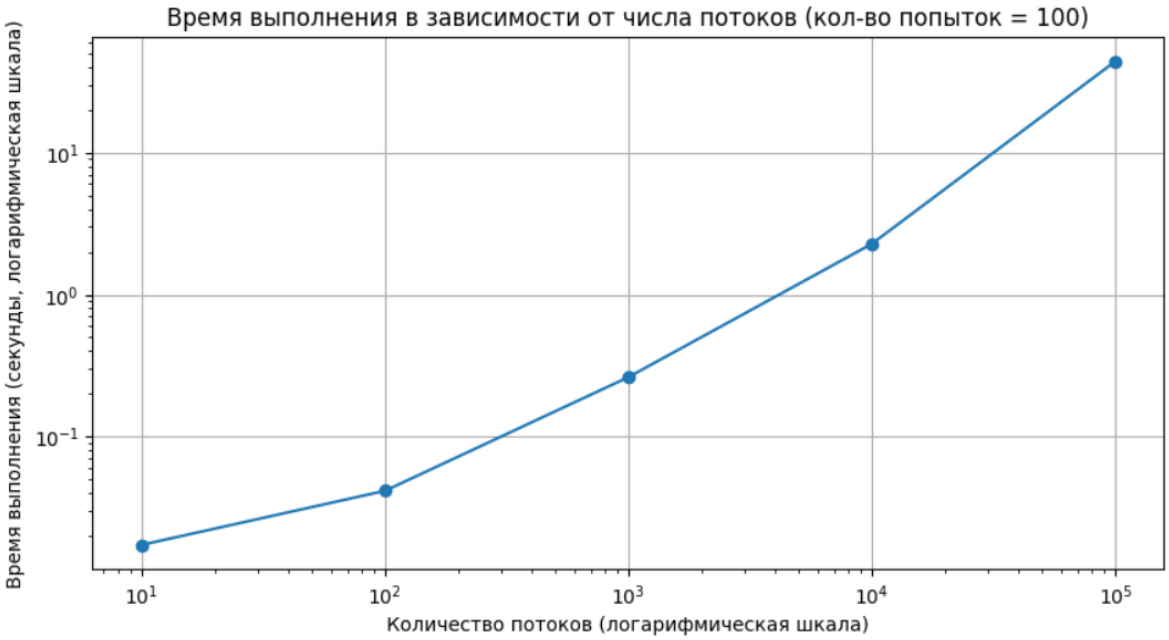
10, 100, 0.017357
100, 100, 0.041676
1000, 100, 0.262887

10000, 100, 2.278856
100000, 100, 43.845609
10, 1000, 0.008789
10, 10000, 0.010802
10, 100000, 0.016118
10, 1000000, 0.015984
10, 10000000, 0.107255

```
import matplotlib.pyplot as plt

# Данные для графика с увеличением числа потоков (при фиксирован
num_threads = [10, 100, 1000, 10000, 100000]
execution_times = [0.017357, 0.041676, 0.262887, 2.278856, 43.84

plt.figure(figsize=(10, 5))
plt.plot(num_threads, execution_times, marker='o')
plt.xscale('log')
plt.yscale('log')
plt.title('Время выполнения в зависимости от числа потоков (кол
plt.xlabel('Количество потоков (логарифмическая шкала)')
plt.ylabel('Время выполнения (секунды, логарифмическая шкала)')
plt.grid(True)
plt.show()
```

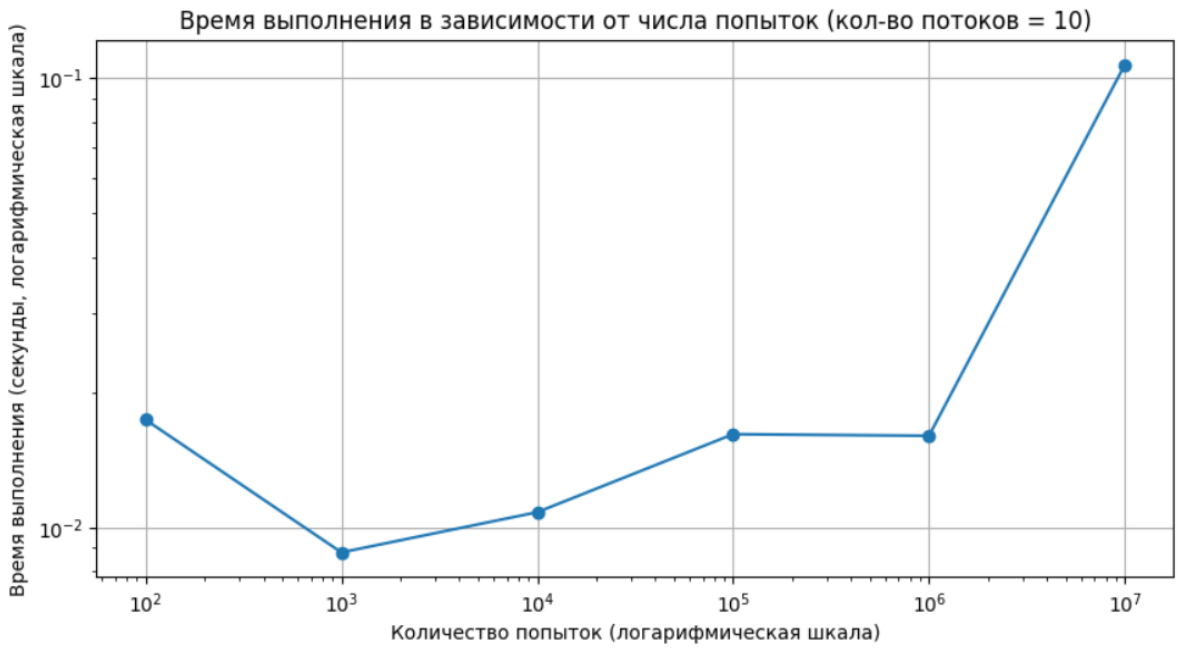


Как видно из графика, время выполнения растет при увеличении количества потоков и делает это нелинейно. Здесь могут быть следующие причины:

1. **Контенция мьютексов**
   В программе используется мьютекс для синхронизации доступа к общей переменной
   `global_hits`. Когда несколько потоков одновременно пытаются обновить эту переменную, они должны ожидать освобождения мьютекса. Это приводит к контенции, которая увеличивает общее время выполнения. С увеличением количества потоков контенция становится более выраженной, и время ожидания потоков возрастает.
2. **Накладные расходы на управление потоками**
   При создании и управлении потоками возникают дополнительные накладные расходы. Каждый новый поток требует ресурсов на инициализацию и переключение контекста, что может превышать выгоду от распараллеливания. При увеличении числа потоков эти накладные расходы становятся более заметными.

```
# Данные для графика с увеличением числа попыток
num_attempts = [100, 1000, 10000, 100000, 1000000, 10000000]
execution_times_attempts = [0.017357, 0.008789, 0.010802, 0.016
```

```
plt.figure(figsize=(10, 5))
plt.plot(num_attempts, execution_times_attempts, marker='o')
plt.xscale('log')
plt.yscale('log')
plt.title('Время выполнения в зависимости от числа попыток (кол-
plt.xlabel('Количество попыток (логарифмическая шкала)')
plt.ylabel('Время выполнения (секунды, логарифмическая шкала)')
plt.grid(True)
plt.show()
```



По графику выше можно сказать, что при увеличении количества попыток время выполнения не растет, или растет несущественно (по крайней мере, на небольших объемах. Кратный рост произошел только на 10 миллионах). Это может быть связано с несколькими факторами:

1. **Параллельная обработка**: Поскольку количество потоков фиксировано, они распределяют работу по попыткам равномерно. Каждый поток выполняет свои вычисления параллельно, что позволяет более эффективно использовать процессорное время. Увеличение числа

попыток приводит к увеличению нагрузки на потоки, но прирост времени выполнения остается минимальным благодаря параллелизму.

2. **Постоянные накладные расходы на создание потоков**: При фиксированном числе потоков накладные расходы на создание и управление потоками остаются неизменными. Это означает, что время выполнения не увеличивается значительно при добавлении новых попыток, так как накладные расходы не растут с увеличением числа попыток.

Выводы

Использование многопоточности в данной задаче продемонстрировало значительное ускорение вычислений. Распределение попыток между несколькими потоками позволяет использовать ресурсы процессора более эффективно, что особенно заметно при больших значениях числа попыток.

Время выполнения программы не пропорционально увеличивается с ростом числа потоков, что связано с конкурентным доступом к общим ресурсам (в частности, к переменной для подсчета попаданий в круг). Нелинейный рост времени выполнения при увеличении числа потоков подчеркивает важность управления ресурсами, такими как мьютексы, для предотвращения блокировок и потерь производительности.

При фиксированном числе потоков время выполнения программы несущественно меняется с ростом количества попыток. Это свидетельствует о том, что для данной реализации программа успешно использует параллельную обработку, обеспечивая эффективное распределение задач и минимизируя накладные расходы.