ОТЧЁТ ПО ЭКСПЕРИМЕНТУ (операция редукции)

Постановка задачи

В задании исследуется, как различные механизмы взаимного исключения в OpenMP влияют на скорость выполнения операции суммирования.Рассматриваются четыре подхода: атомарные операции (`atomic`), критические секции (`critical`), блокировки (`lock`) и встроенная редукция через параметр `reduction` директивы `for`.

Суммирование выполняется над большим числом элементов для нескольких значений `n` (от 10^5 до $2 \cdot 10^7$) и разным количеством потоков (1, 2, 4, 5, 8), после чего сравниваются время работы и ускорение относительно последовательного варианта.

Описание реализации

В качестве базового ядра используется функция, которая для индекса `i` выполняет небольшое фиксированное количество тригонометрических операций и возвращает вещественное значение:

inline double kernel(int i) { double x = 0.0; int reps = 20 + (i % 30); for (int k = 0; k < reps; ++k) { x += std::sin(0.0001 * (i + k)); x -= std::cos(0.0002 * (i + 3 * k)); } return x; }

Последовательная сумма используется только для проверки корректности:

double seq_sum(long long n) { double acc = 0.0; for (long long i = 0; i < n; ++i) { acc += kernel(static_cast(i)); } return acc; }

Параллельная часть отличается лишь способом защиты общей переменной `sum`:

- **atomic**: каждый поток накапливает частичную сумму и добавляет её к общей переменной через `#pragma omp atomic`.
- **critical**: добавление частичных результатов обёрнуто в `#pragma omp critical`.
- **lock**: используется явный `omp_lock_t` и вызовы `omp_set_lock` / `omp_unset_lock`.
- **reduction**: применяется `#pragma omp parallel for reduction(+ : sum)`, что позволяет компилятору сгенерировать эффективную схему слияния частичных сумм.

Результаты измерений

Все графики лежат в папке `plots/`.

Графики времени исполнения (time)

- `n = 100000`: reduce_n100000_time_by_method.jpg

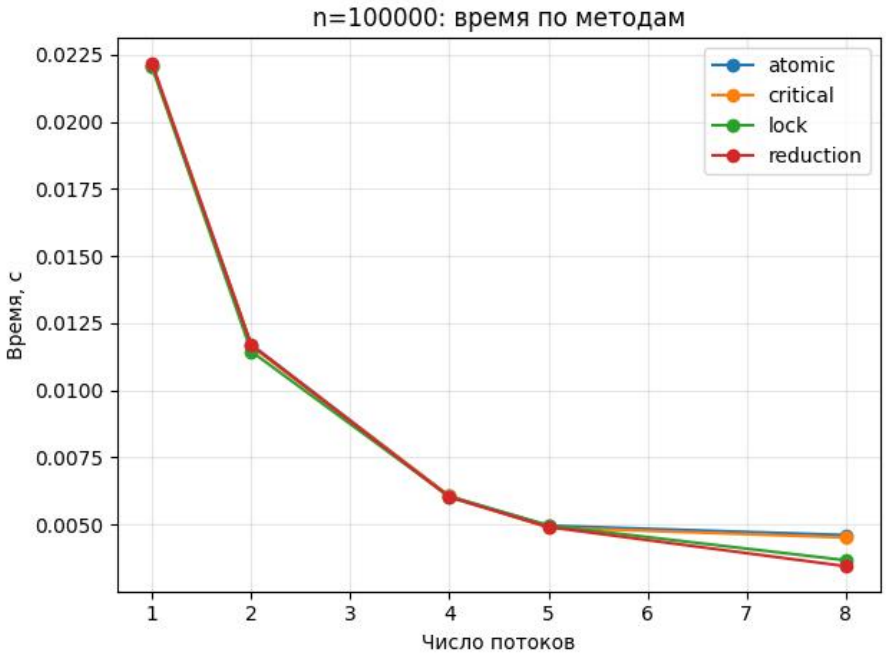

Figure 1: n=100000: время по методам

- `n = 1000000`: reduce_n1000000_time_by_method.jpg
- `n = 5000000`: reduce_n5000000_time_by_method.jpg
- `n = 20000000`: reduce_n20000000_time_by_method.jpg

Графики ускорения (speedup)

- `n = 100000`: reduce_n100000_speedup_by_method.jpg
- `n = 1000000`: reduce_n1000000_speedup_by_method.jpg
- `n = 5000000`: reduce_n5000000_speedup_by_method.jpg
- `n = 20000000`: reduce_n20000000_speedup_by_method.jpg

Агрегированные показатели (усреднение по всем размерам `n`)

- Среднее время: reduce_mean_time_by_method.jpg
- Среднее ускорение: reduce_mean_speedup_by_method.jpg

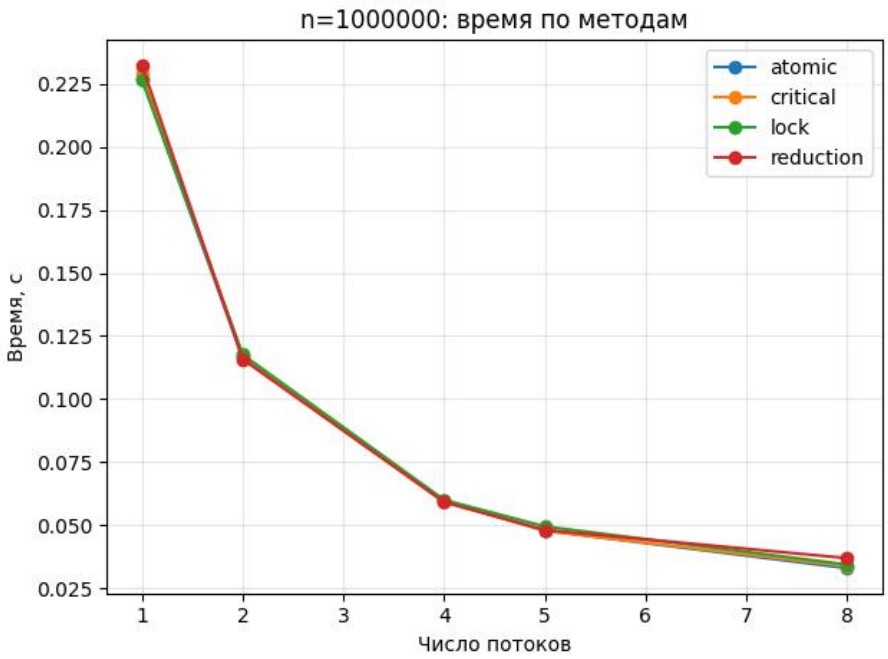

Figure 2: n=1000000: время по методам

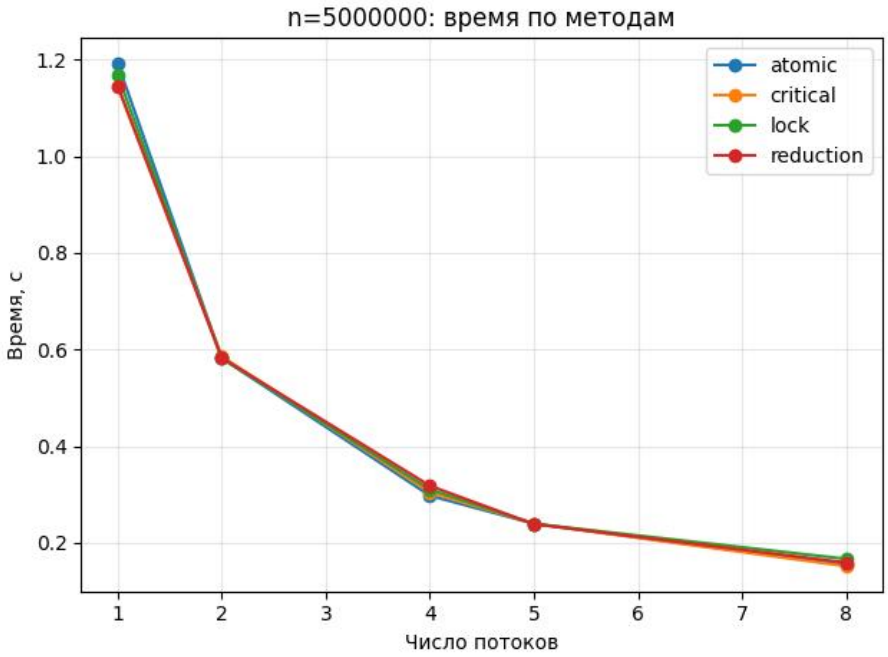

Figure 3: n=5000000: время по методам

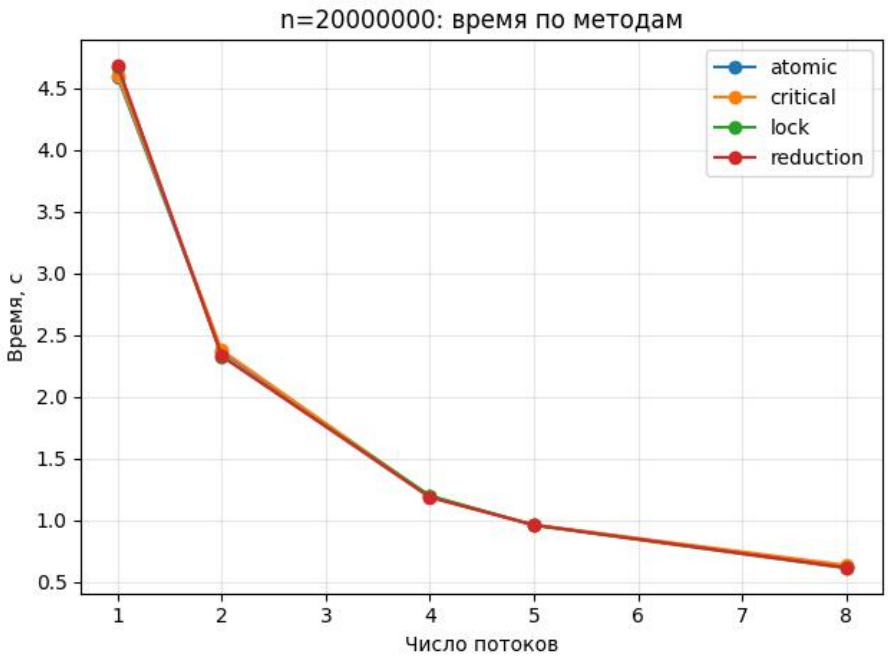

Figure 4: n=20000000: время по методам

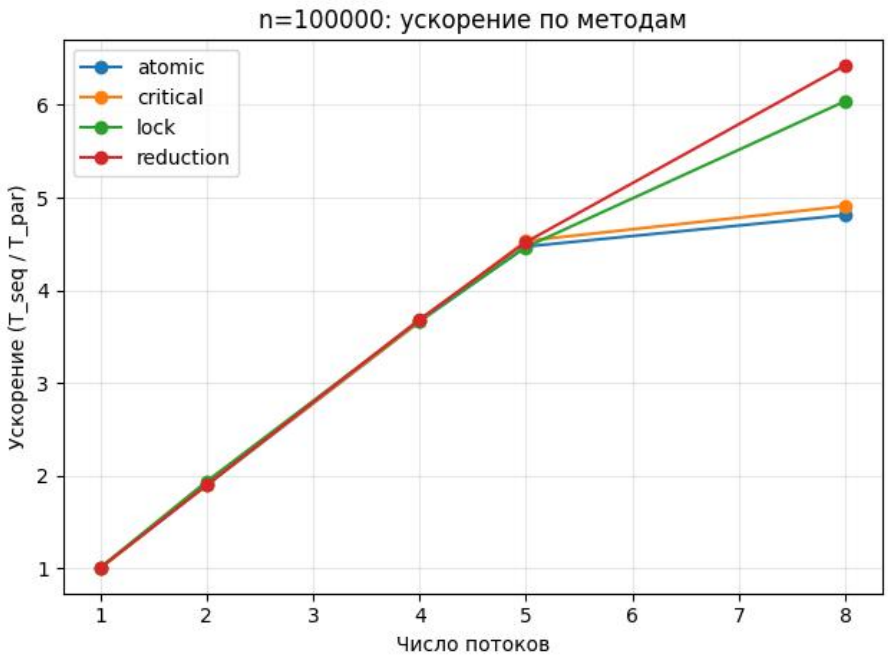

Figure 5: n=100000: ускорение по методам

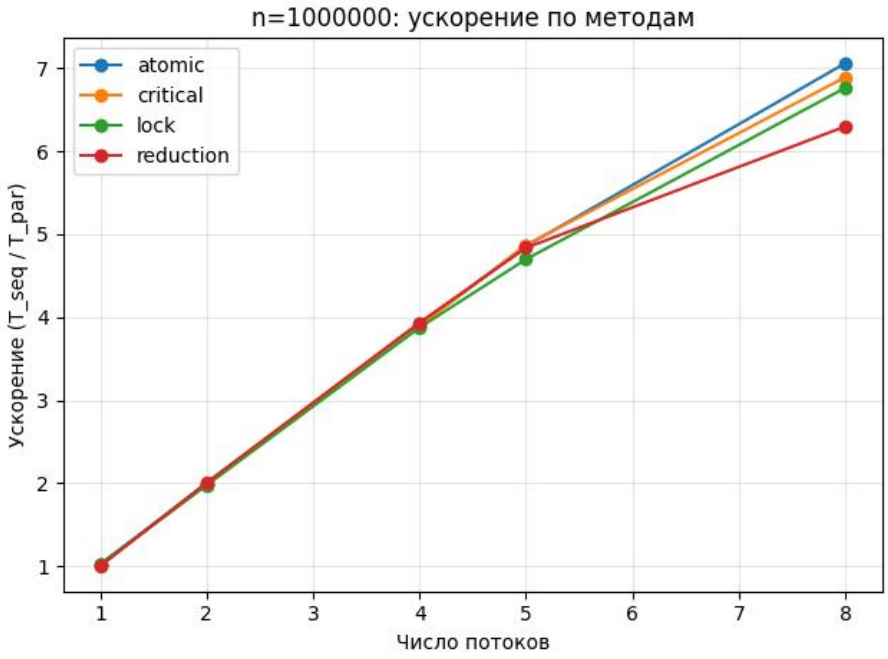

Figure 6: n=1000000: ускорение по методам

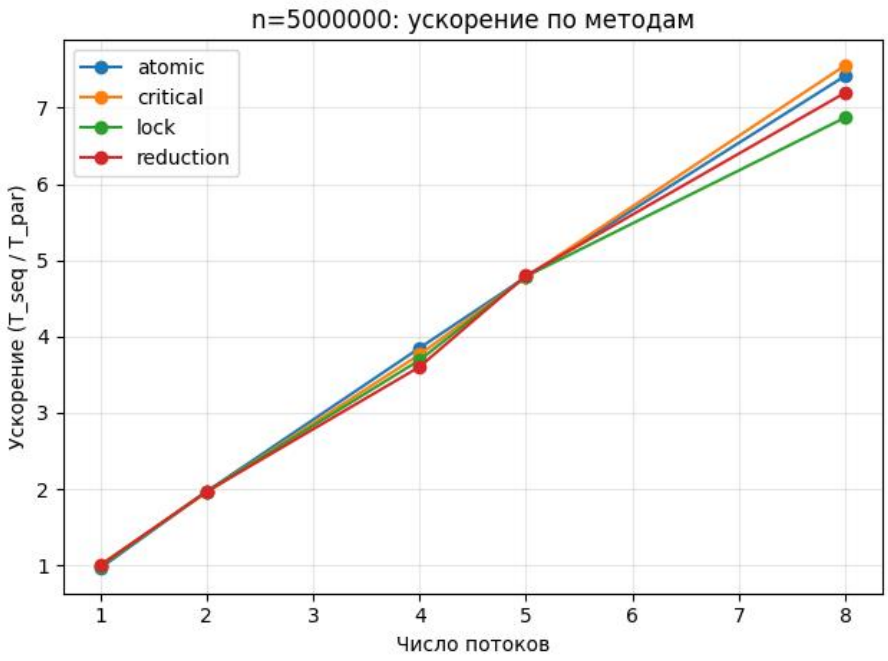

Figure 7: n=5000000: ускорение по методам

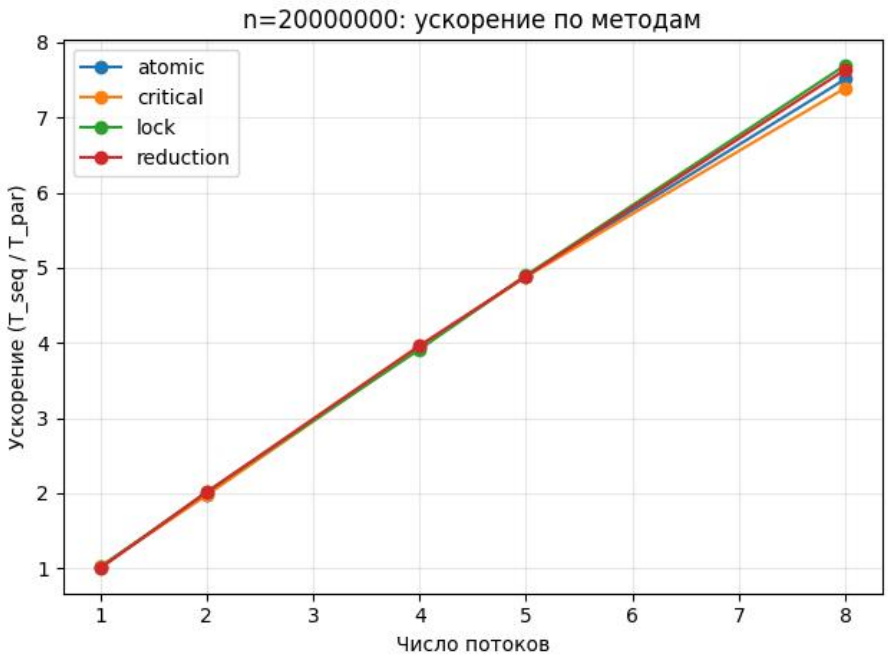

Figure 8: n=20000000: ускорение по методам

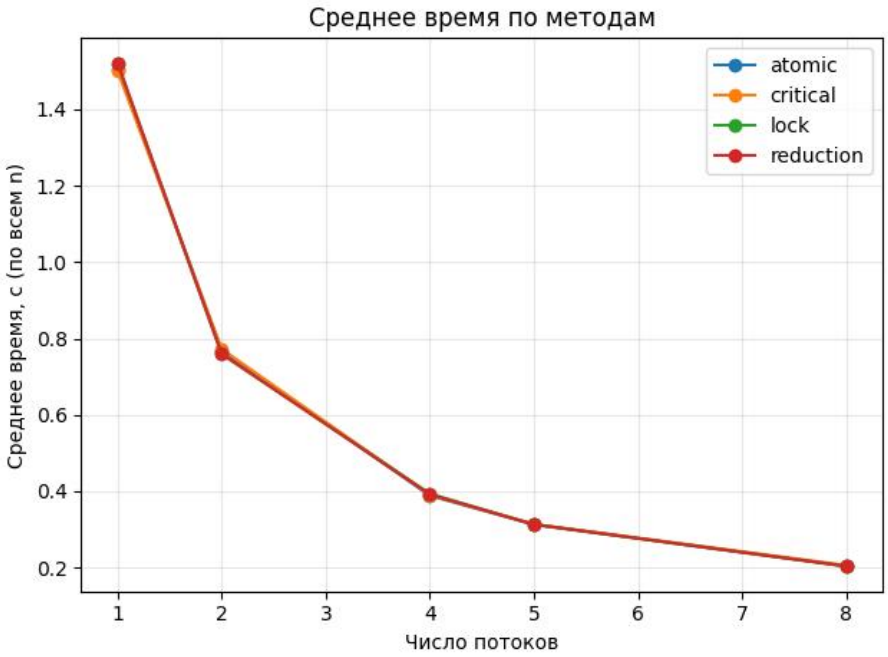

Figure 9: Среднее время по методам

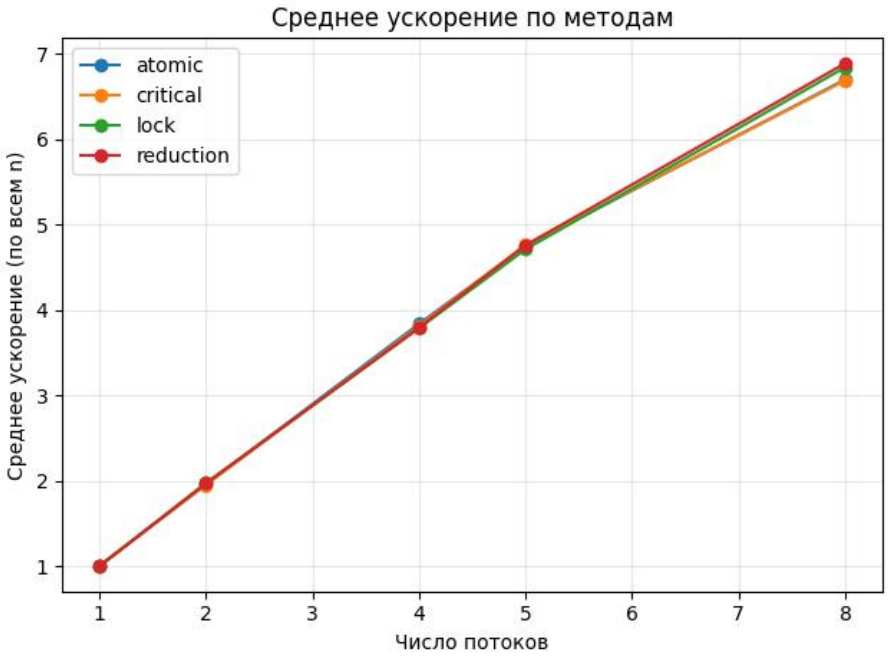

Figure 10: Среднее ускорение по методам

Анализ поведения методов

Малые и средние размеры `n`

При $n = 10^5$ и 10^6 заметно, что:

- Время работы для `atomic`, `critical` и `lock` **увеличивается** при росте числа потоков: доминирует стоимость синхронизации, а не вычислений.
- Встроенная **reduction** остаётся существенно быстрее остальных подходов и даёт единственный видимый прирост по ускорению, достигая значений порядка $S \approx 1.7\text{-}1.8$ на 4-8 потоках.

Крупные задачи

Для $n = 5 \cdot 10^6$ и $2 \cdot 10^7$:

- Время всех методов растёт пропорционально размеру задачи, но относительные различия сохраняются: `critical` и `lock` consistently оказываются самыми медленными из-за высоких накладных расходов на взаимное исключение.
- `atomic` ведёт себя лучше критической секции и замков, но остаётся заметно медленнее `reduction`, особенно на 4-8 потоках.
- Ускорение `reduction` достигает значений около $S \approx 3$ при $n = 2 \cdot 10^7$ и восьми потоках, что отражает хорошую масштабируемость этого подхода.

Усреднённые метрики

Средние по всем `n` кривые показывают общую картину:

- У `atomic`, `critical` и `lock` среднее ускорение близко к нулю: увеличение числа потоков почти не уменьшает общее время из-за серьёзных накладных расходов на синхронизацию.
- `reduction` демонстрирует стабильный рост средней производительности с увеличением числа потоков, оставаясь самым быстрым методом на всех рассматриваемых размерах задач.

Выводы

1. Явные механизмы взаимного исключения (`critical`, `lock`) плохо подходят для частых обновлений общей переменной: при увеличении числа потоков они приводят к росту времени выполнения и практически не дают ускорения.

2. Атомарные операции снимают часть накладных расходов, но даже `atomic` остаётся ощутимо медленнее встроенной редукции и масштабируется ограниченно.
3. Использование параметра `reduction` в директиве `for` оказывается наиболее эффективным вариантом: при больших `n` он обеспечивает наилучшее сочетание времени работы и ускорения, а также самую предсказуемую масштабируемость при переходе от 1 к 8 потокам.