БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра дискретной математики и алгоритмики

Азявчиков Алексей Павлович

ИСПОЛЬЗОВАНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ДЛЯ УСКОРЕНИЯ МАССОВЫХ ОПЕРАЦИЙ С ОДНОРОДНЫМИ ДАННЫМИ

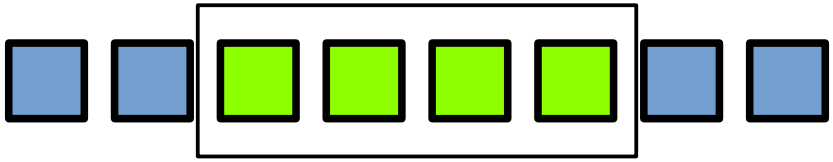
Научный руководитель:
старший преподаватель
И. Д. Лукьянов

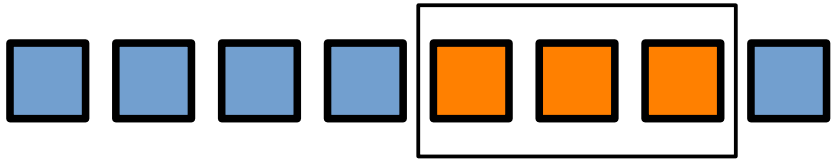
Основные определения

- **Поток -** это наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.
- **Гонка данных (Data race)** - это ситуация, когда два или более потока одновременно обращаются к одному и тому же общему ресурсу и при этом хотя бы один из них выполняет операцию записи.
- **Работой** многопоточного вычисления называется общее время выполнения всего вычисления на одном процессоре, исполняющим лишь один поток в каждый момент времени.
- **Интервалом** многопоточного вычисления называется наибольшее время выполнения одного потока на идеальном процессоре, т.е. процессоре, исполняющим потенциально бесконечно большое число потоков в каждый момент времени.

Постановка задачи

Целью курсовой работы является реализация на языке C++ шаблонной структуры над массивом однородных данных, предоставляющей пользователю интерфейс для модификиции данных на отрезке и получения состояния отрезка, которая при этом использует параллельные алгоритмы, чтобы ускорить выполнение данных операций по сравнению с последовательной версией





Постановка задачи

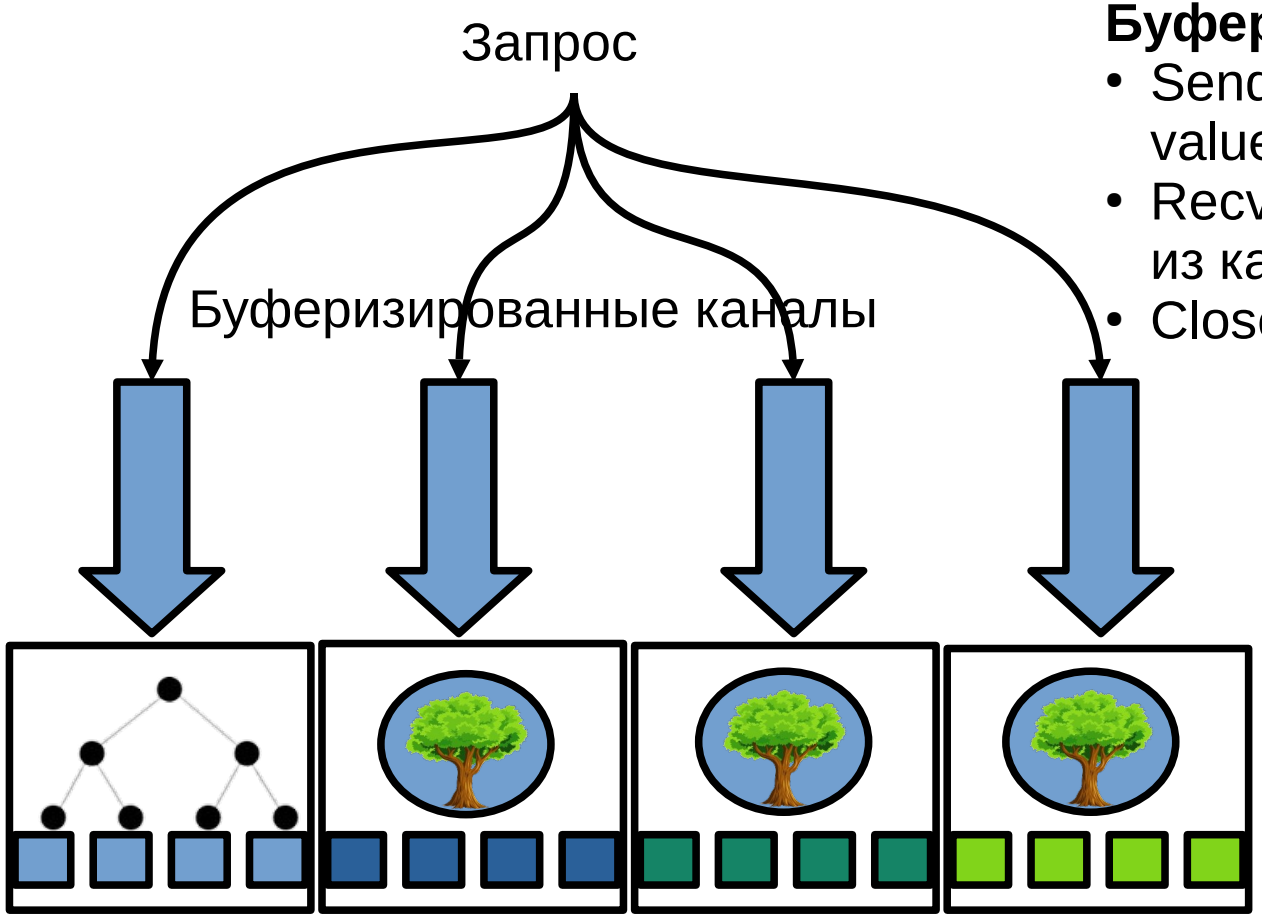
Интерфейс предоставляемый пользователю – это возможность передать в шаблон структуры политику модификации однородных данных на отрезке, которая должна реализовывать следующие функции:

- **GetState(a, b)**: вернуть состояние отрезка, состоящего из элементов a и b. Данная операция должна быть ассоциативной;
- **GetNullState()**: вернуть ноль относительно операции взятия состояния;
- **GetModifiedState(init_state, values_count, modifier)**: вернуть модифицированное состояние отрезка состоящего из values_count элементов и имеющего до модификации состояние init_state, в соответствие со значением modifier. Результат должен быть аналогичен многократному вызову GetState.

Области применения параллельных алгоритмов

- Базы данных
- Машинное обучение и анализ данных
- Вычислительная физика и научные расчеты
- Графические приложения и игры
- И т.п.

Первый вариант реализации



Буферизированный канал:

- Send(value) — отправить value в канал;
- Recv() — получить значание из канала;
- Close() — закрыть канал.

Первый вариант реализации

Структура запроса к потоку:

```
struct Query {
  std::optional<M> modifier{std::nullopt};
  size_t l{0}, r{0};
  Response response{};
};
struct Response {
  std::shared_ptr<std::promise<T>> result{nullptr};
  std::shared_ptr<std::atomic<T>> state{nullptr};
  std::shared_ptr<std::atomic<int>> counter{nullptr};
};
```

Данная реализация возвращает пользователю объект std::future<T>, делая таким образом вызов метода GetTreeState неблокирующим

Первый вариант реализации

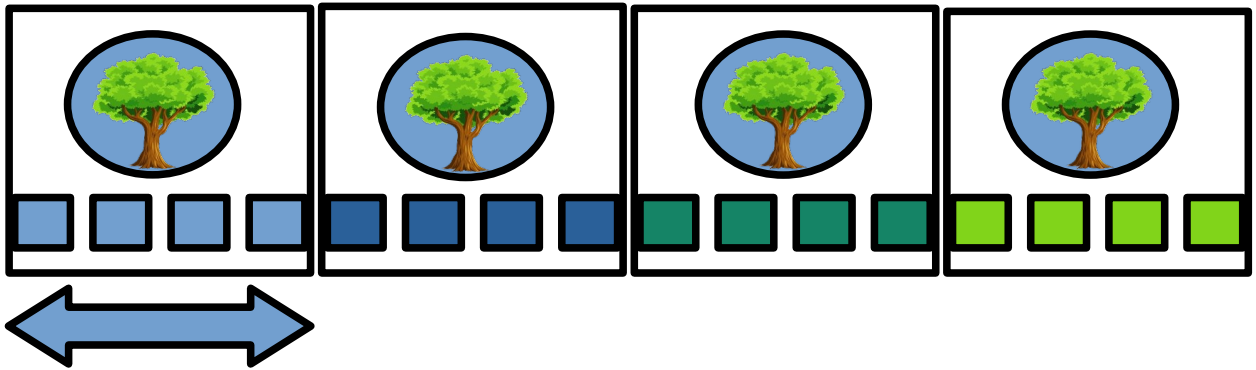
Общий паттерн работы с атомарными переменными на примере response.state:

```
for (auto old_state = response.state->load();;) {
  // state modification:
  auto new_state = this->GetState( first_state: old_state,  second_state: sub_state);
  if (response.state->compare_exchange_strong(old_state, new_state)) {
    break;
  }
}
```

Оценки

Введем обозначения:

- **n** – размер массива;
- **t** – число потоков;
- **buff_size** – размер буфера канала



O(n / t)

ModifyTree

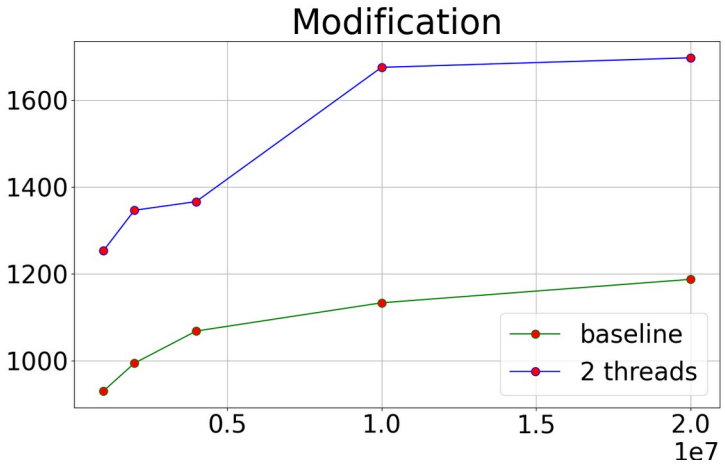
- Интервал O(t + log(n / t))
- Работа O(t + t * log(n / t))

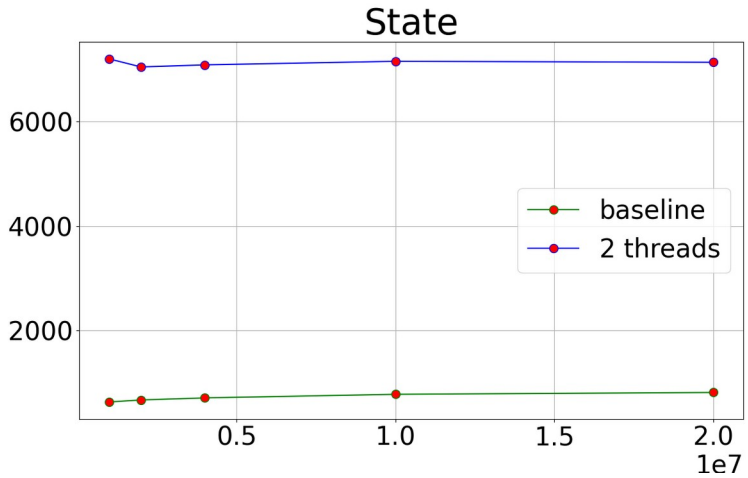
GetTreeState

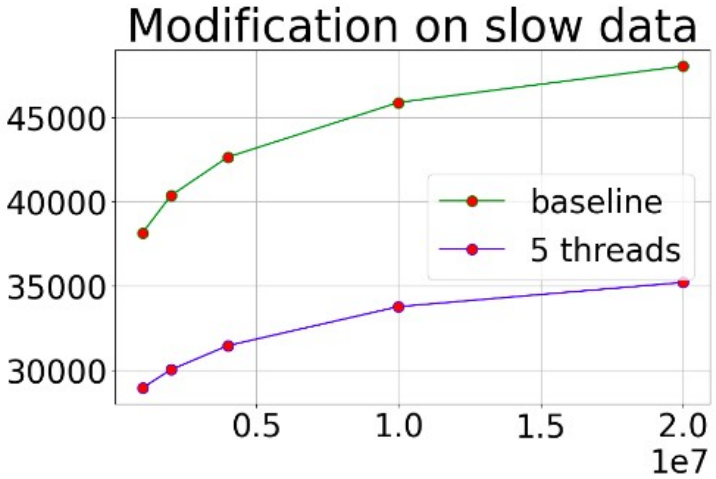
- Интервал O(t + log(n / t))
- Работа O(t + t * log(n / t))

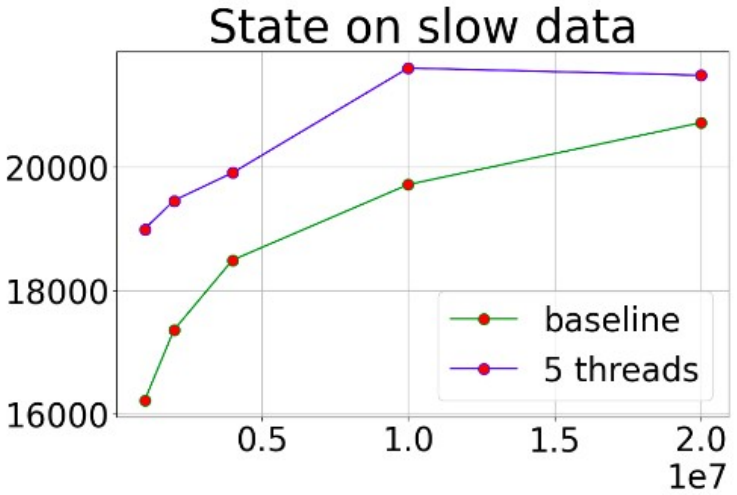
Память: O(n + t * buff_size)

Тестирование (первая версия)





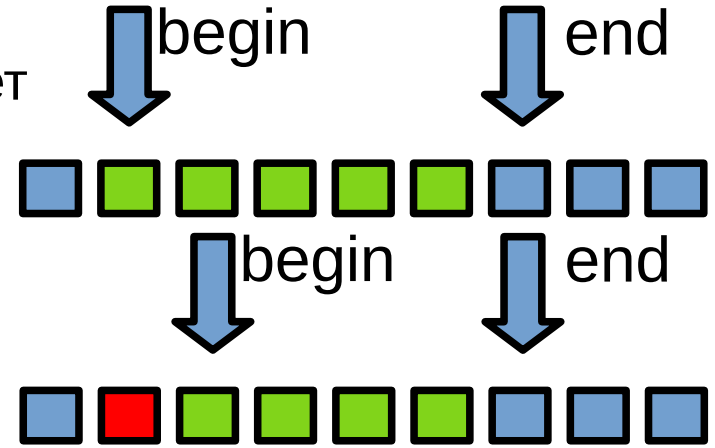




Второй вариант реализации

Быстрая буферизированная очередь:

- Enqueue если очередь заполнена возвращает false, а иначе вставляет элемент и возвращает true;
- Dequeue если очередь пуста возвращает false, а иначе достает элемент и возвращает true.



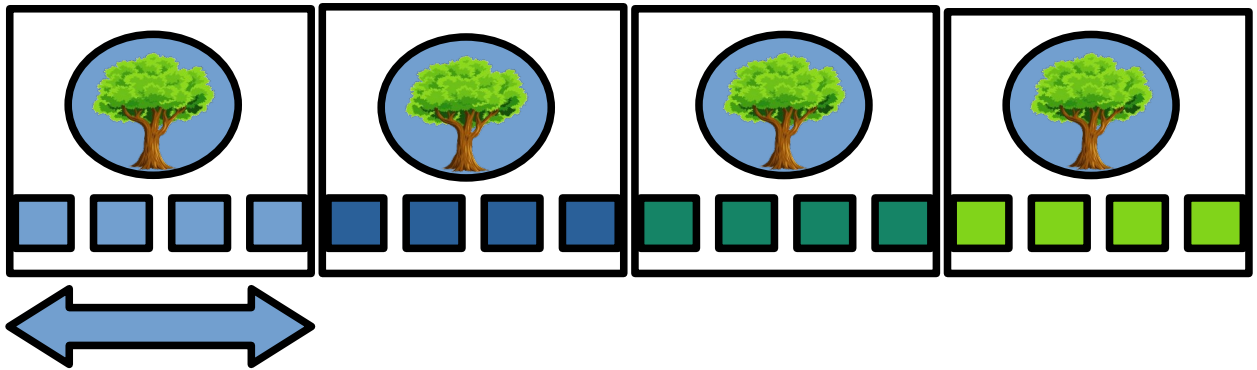
Идея: заменить каналы с их долгими системными вызовами на неблокирующие очереди. Недостатком такого подхода станет дополнительная нагрузка на процессор.

```
while (!channel->Dequeue( &: query)) {
  if (stop_threads_.test()) {
    return;
  }
}
```

Оценки

Введем обозначения:

- **buff_size** – размер буфера быстрой очереди



ModifyTree

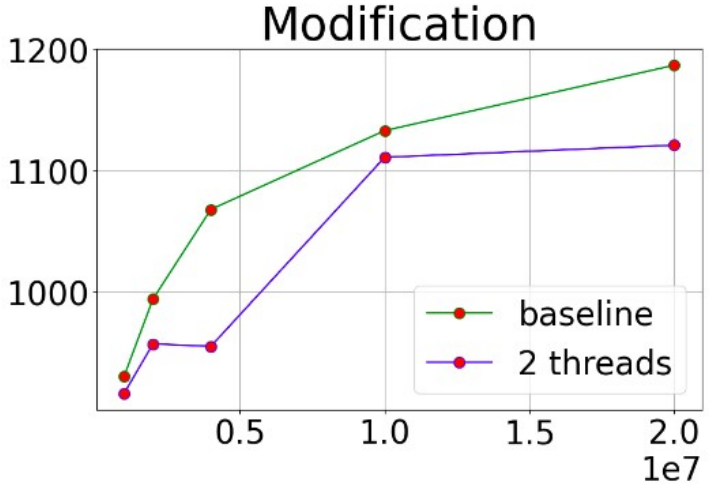
- Интервал O(t + log(n / t))
- Работа O(t + t * log(n / t))

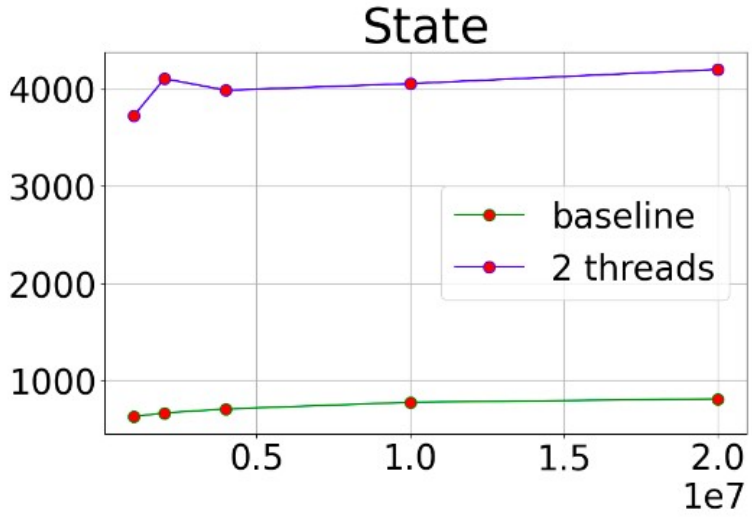
GetTreeState

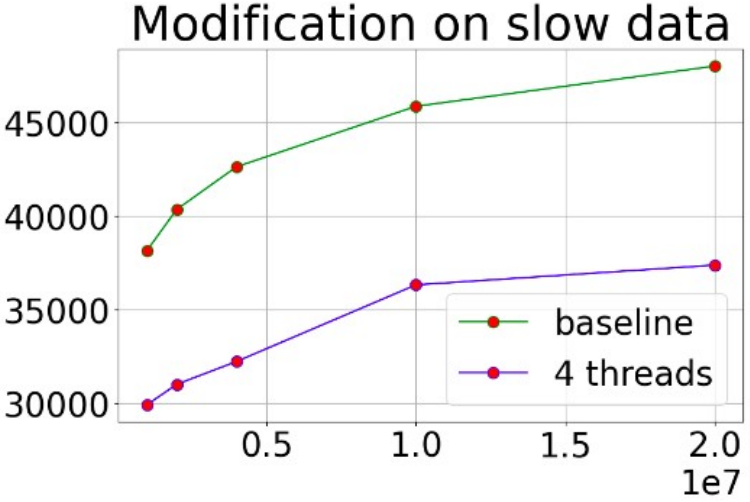
- Интервал O(t + log(n / t))
- Работа O(t + t * log(n / t))

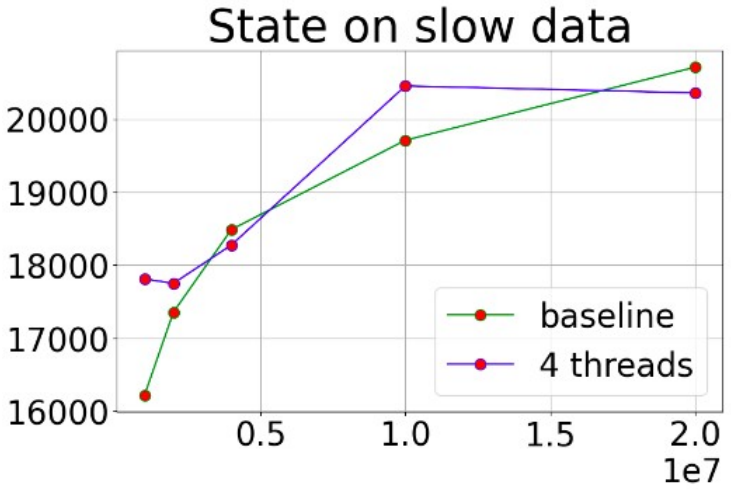
Память: O(n + t * buff_size)

Тестирование (вторая версия)





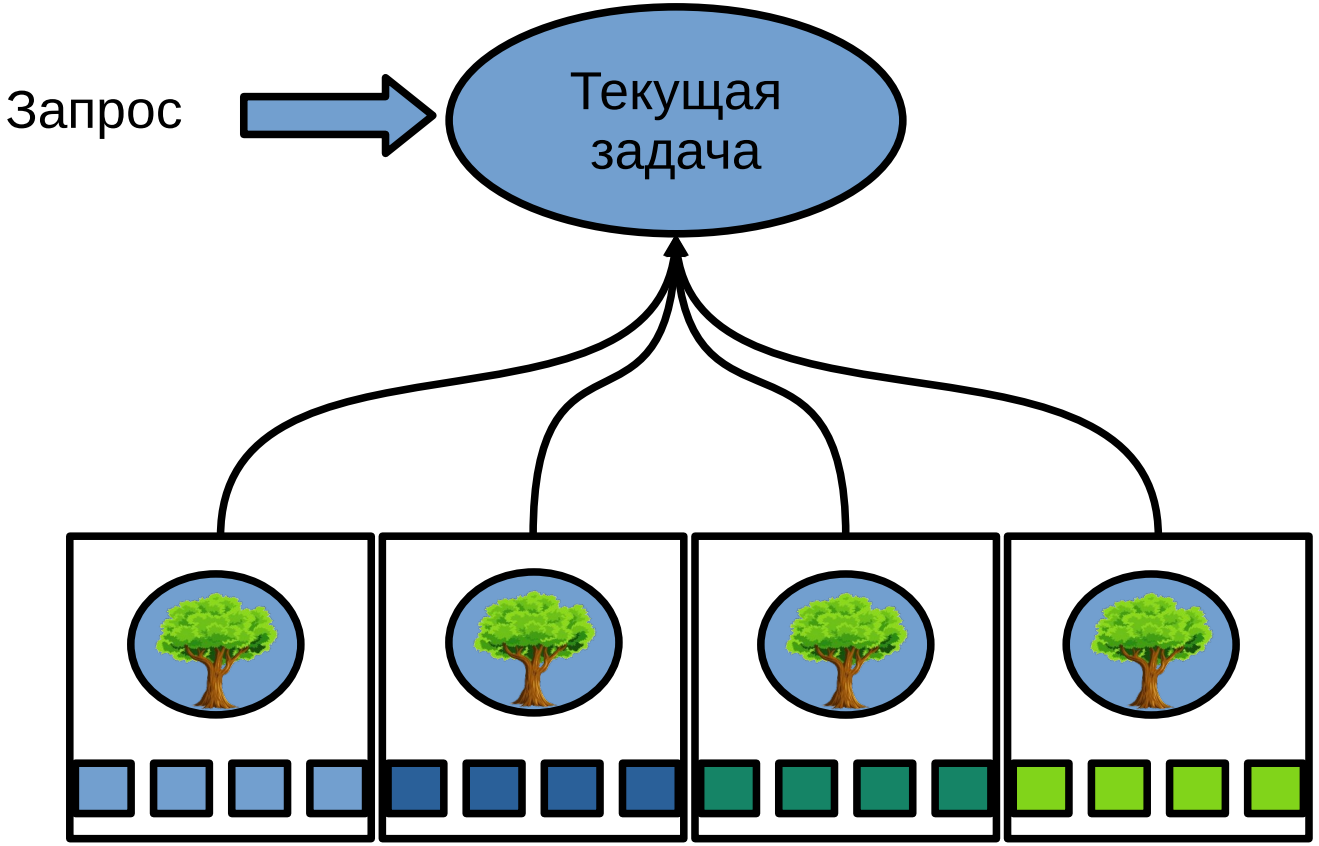




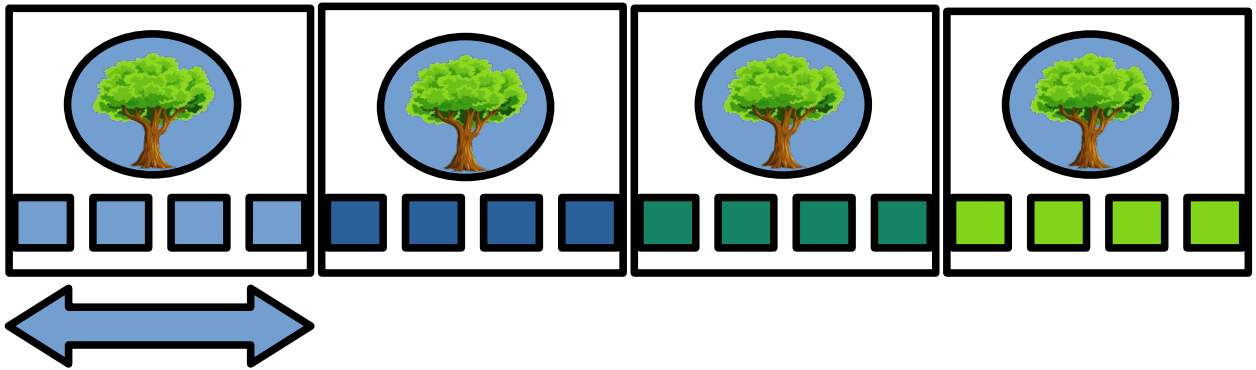
Третий вариант реализации

В данной реализации потоки сами определяют появление новых задач.

Особенностью данного подхода является необходимость дожидаться завершения предыдущей поставленной задачи.



Оценки

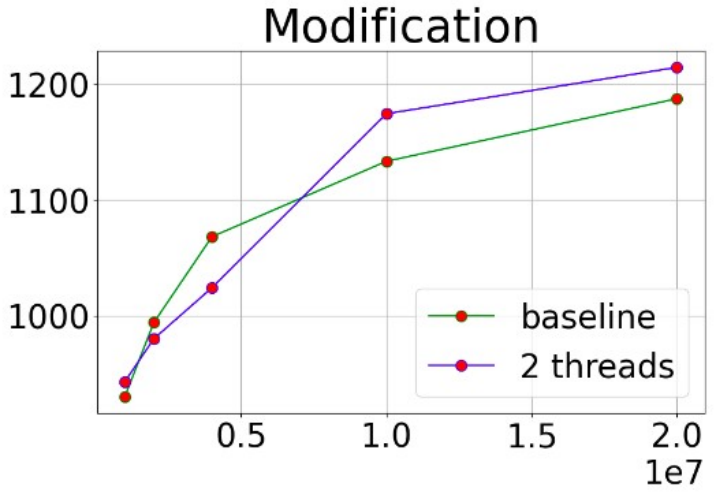


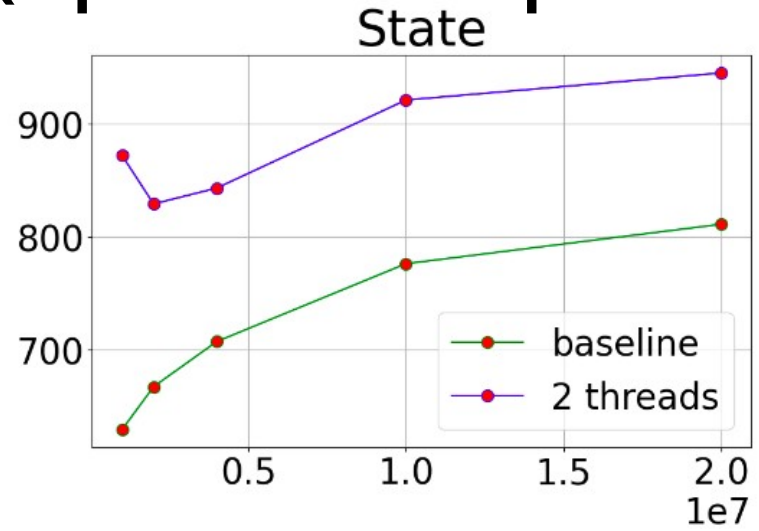
ModifyTree
- Интервал O(1 + log(n / t))
- Работа O(t + t * log(n / t))

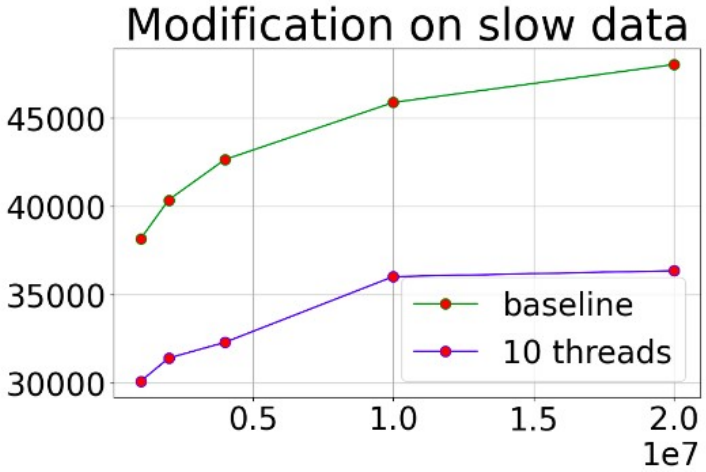
GetTreeState
- Интервал O(1 + log(n / t))
- Работа O(t + t * log(n / t))

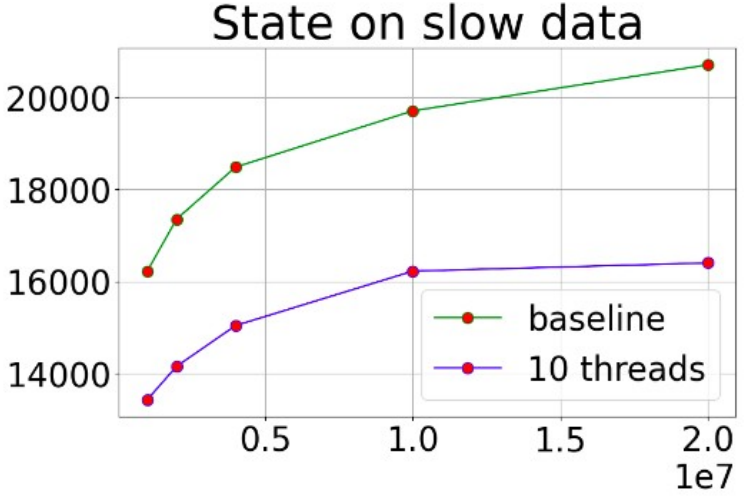
Память: O(n + t)

Тестирование (третья версия)

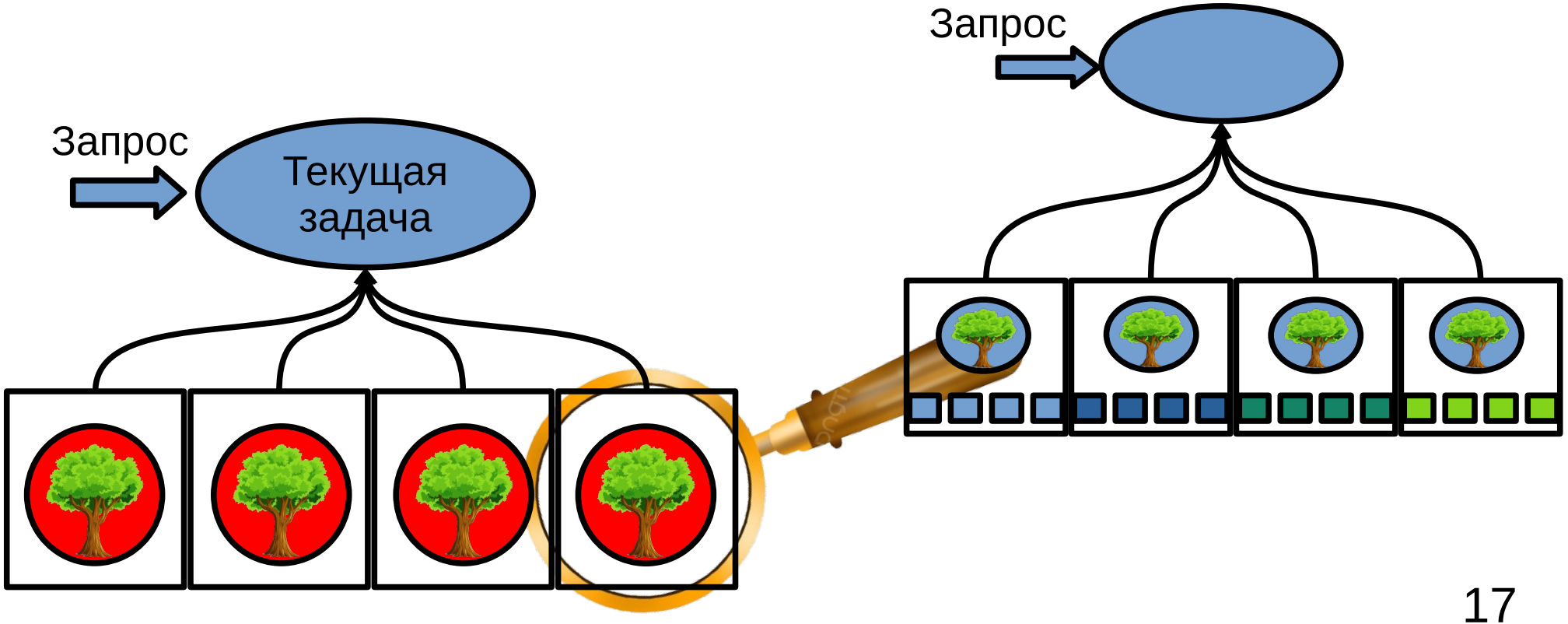








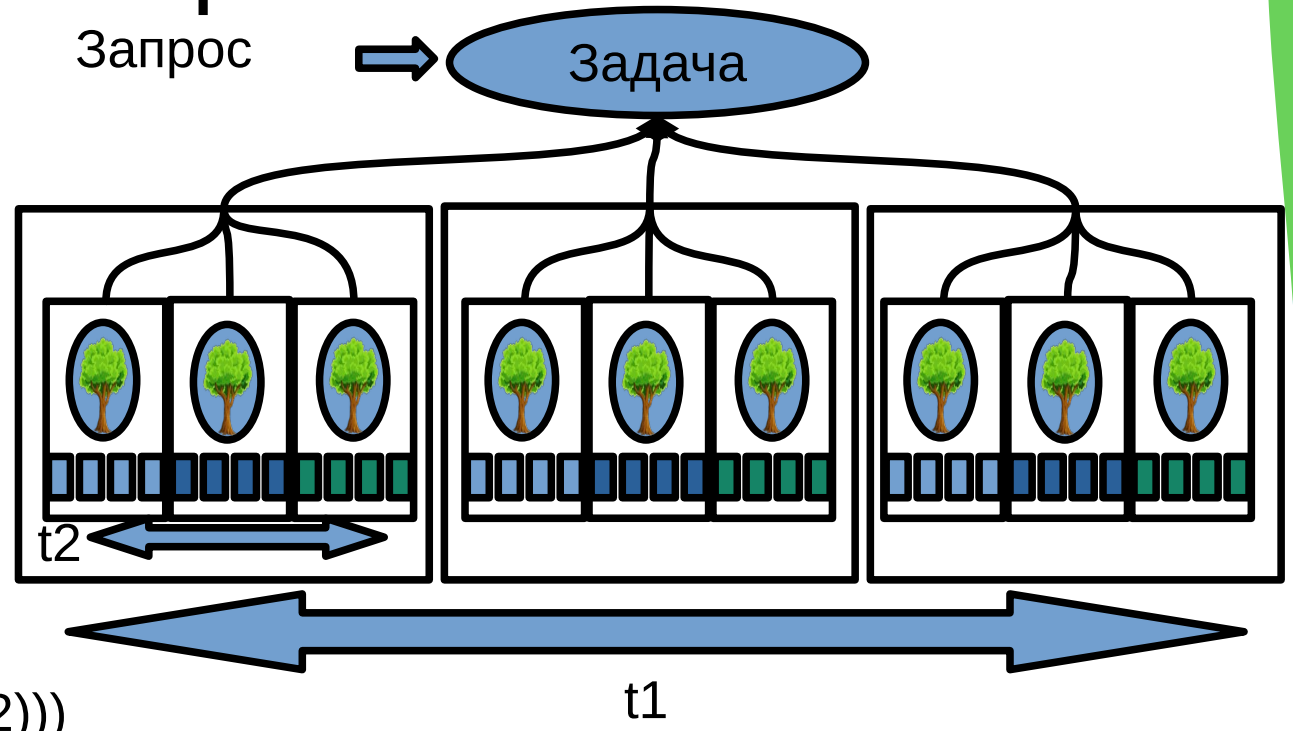
Модифицированный третий вариант реализации



Оценки

Введем обозначения:

- **t1** – число потоков в исходной структуре;
- **t2** – число потоков в дочерних структурах.



ModifyTree

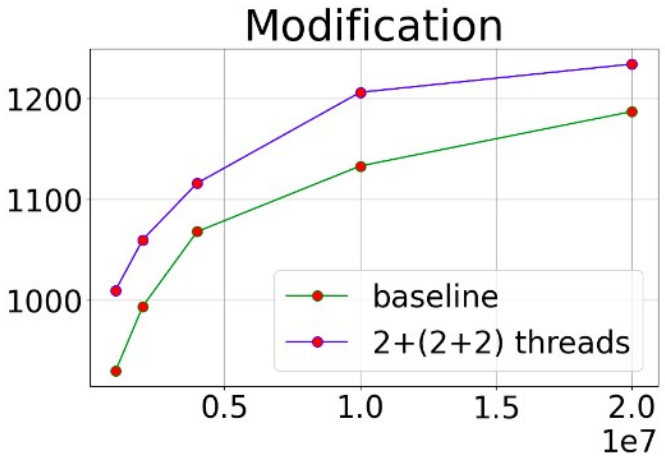
- Интервал O(log(n / (t1*t2)))
- Работа O(t1*t2*log(n/(t1 * t2)))

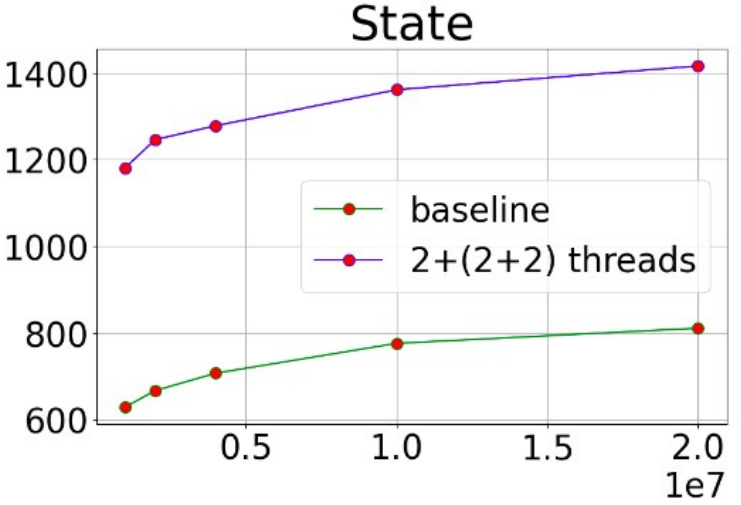
GetTreeState

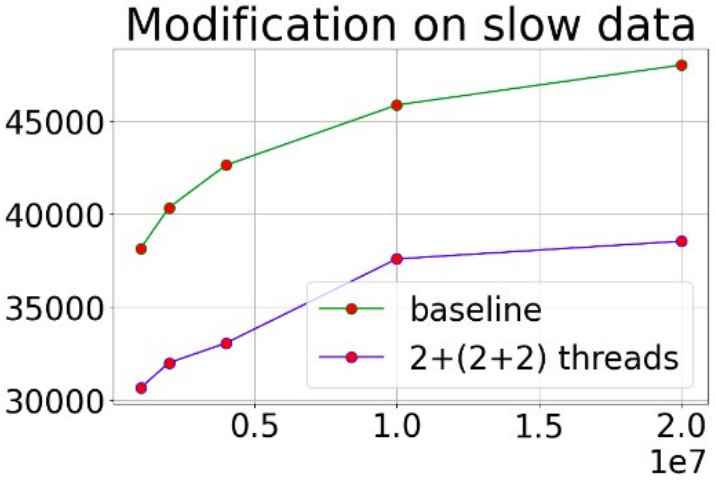
- Интервал O(log(n / (t1*t2)))
- Работа O(t1*t2*log(n/(t1 * t2)))

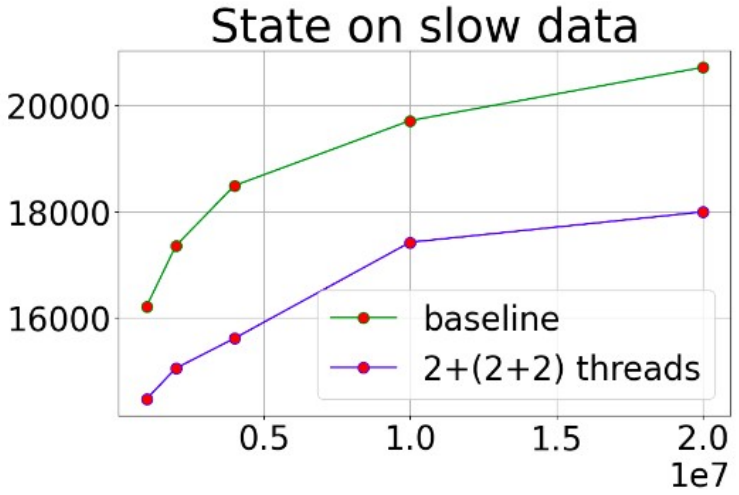
Память: O(n + t1 * t2)

Тестирование (третья* версия)



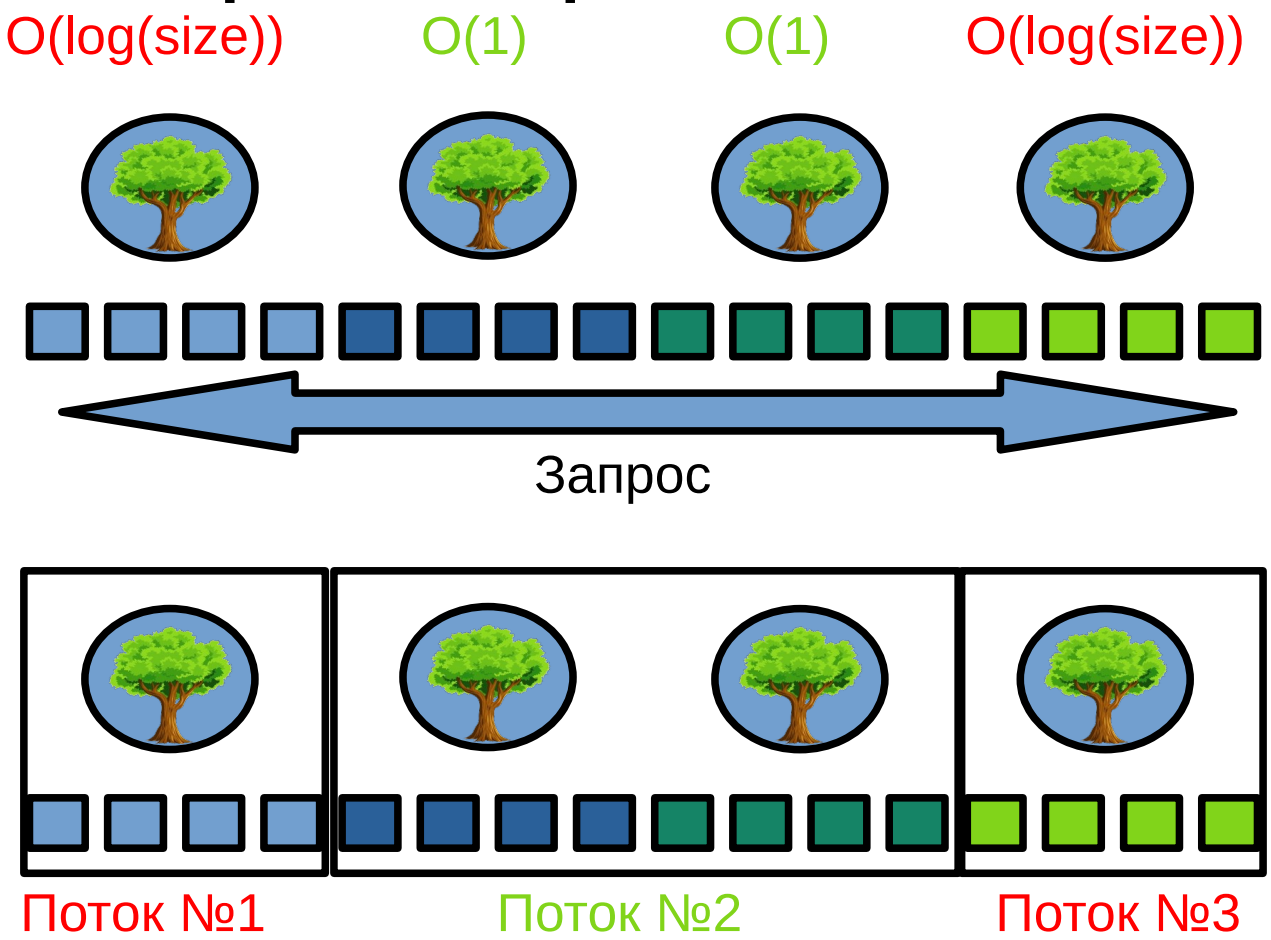






Четвертый вариант реализации

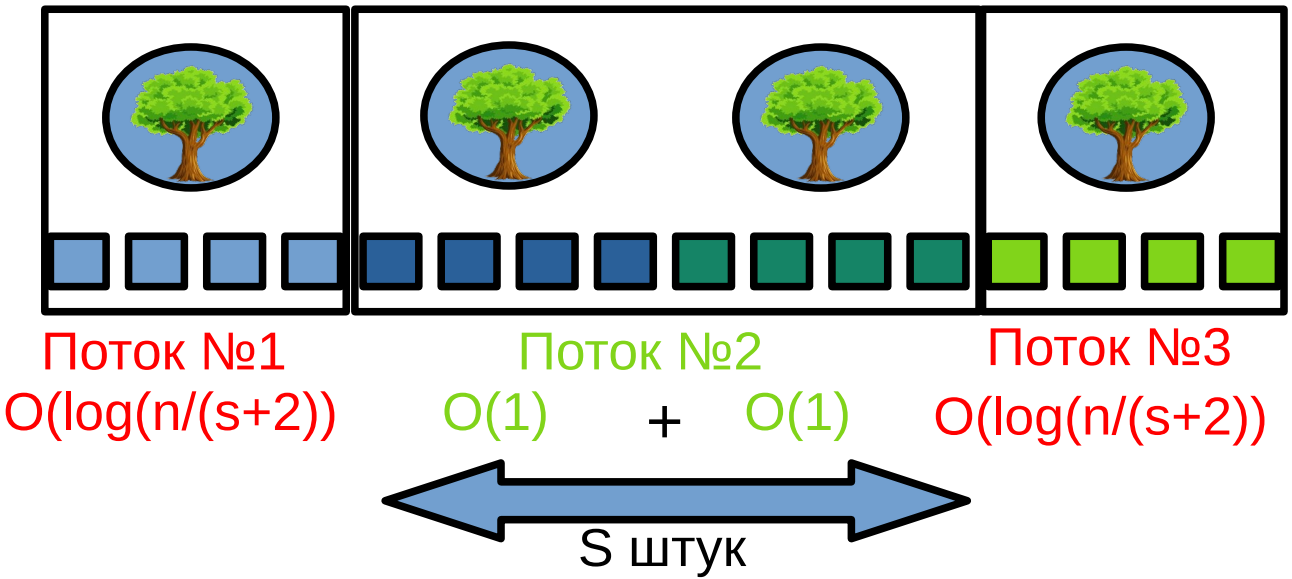
Идея: выдать несколько легких запросов одному потоку



Оценки

Введем обозначение:

- S – (число подмассивов, на которые был разбит исходный массив) - 2.



ModifyTree

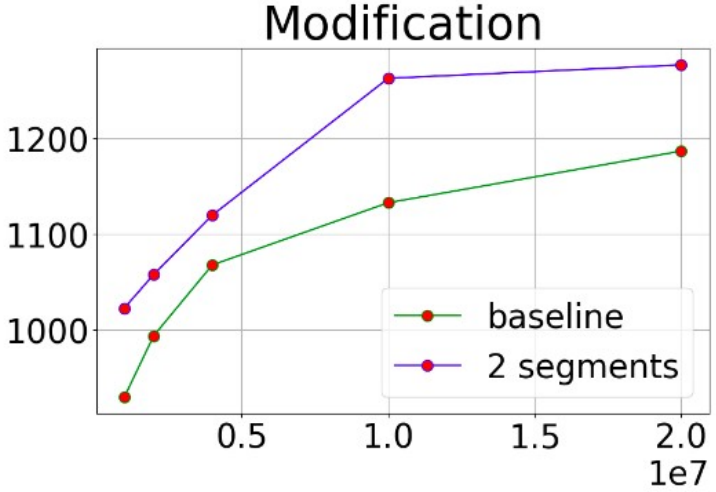
- Интервал O(max(log(n / (s+2), s))
- Работа O(s + 2*log(n/(s+2)))

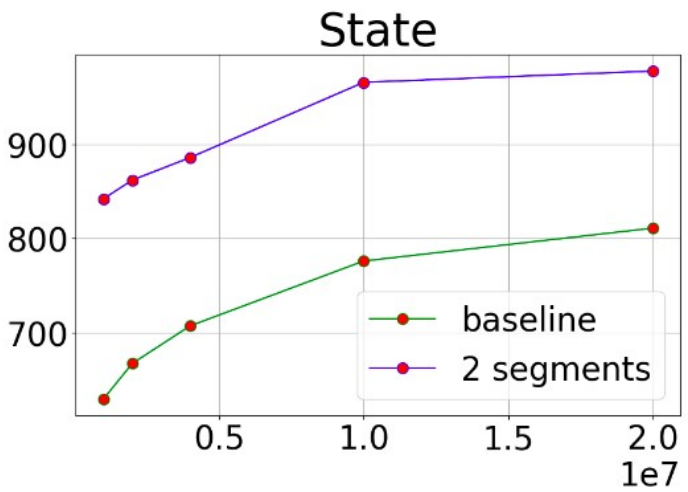
GetTreeState

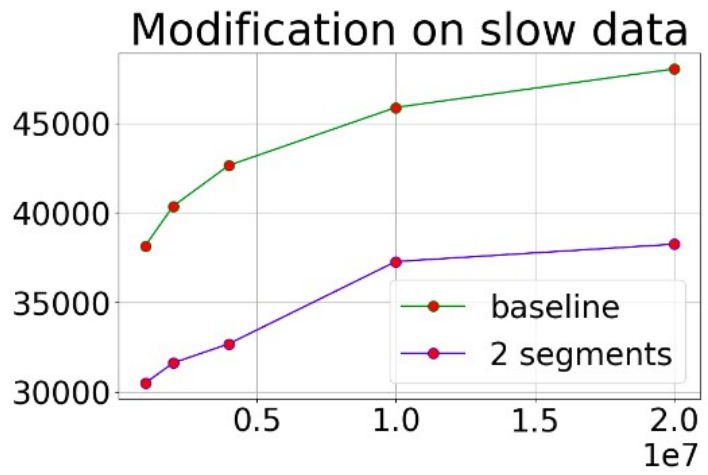
- Интервал O(max(log(n / (s+2), s))
- Работа O(s + 2*log(n/(s+2)))

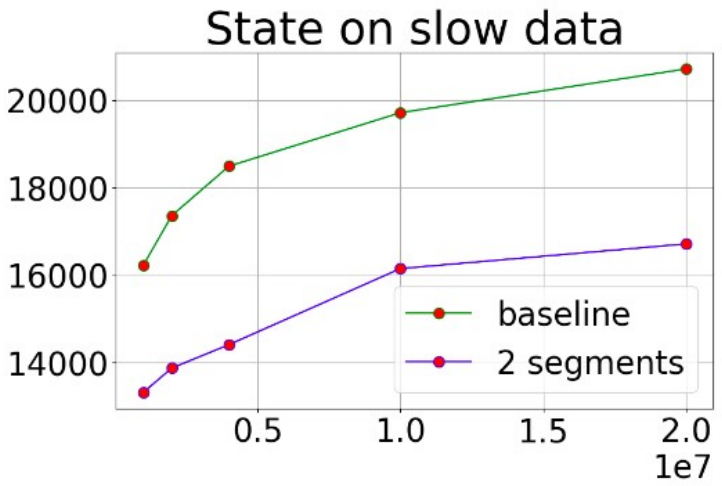
Память: O(n + 3)

Тестирование (четвертая версия)

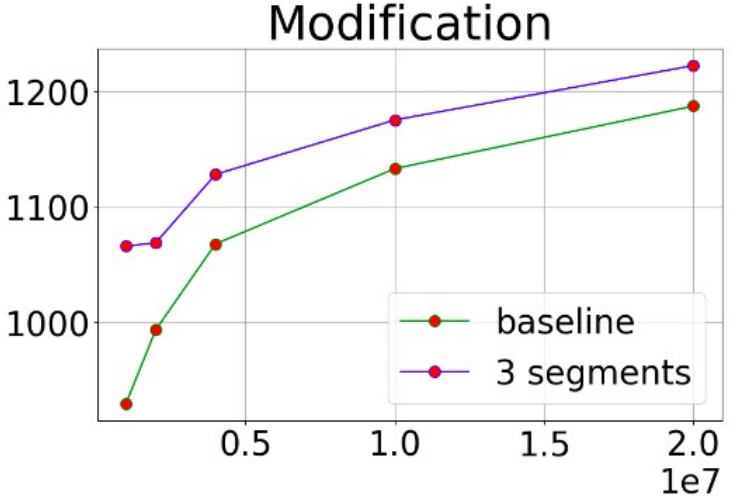


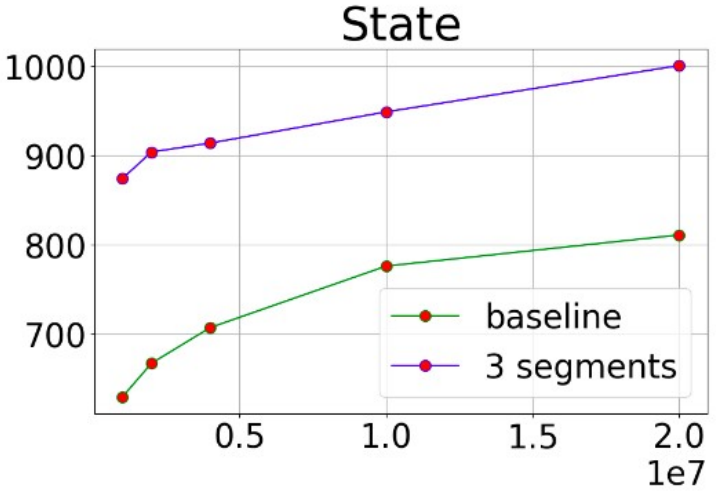


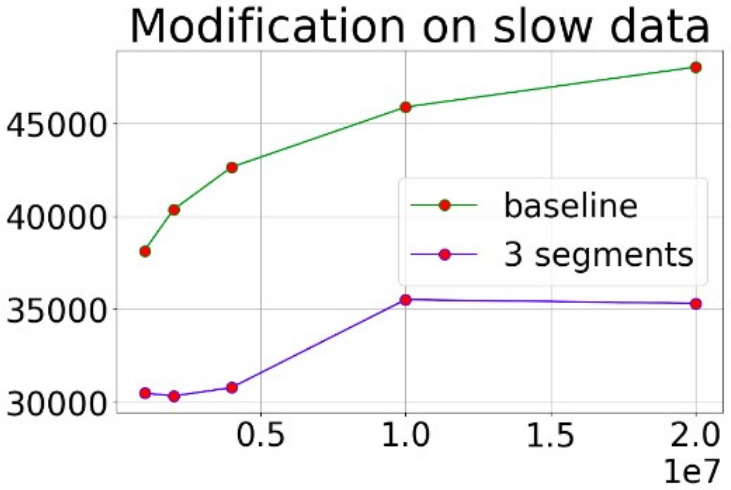


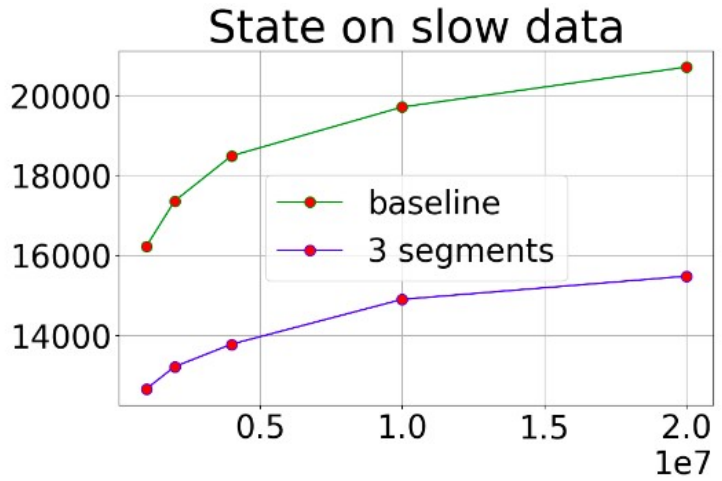


Тестирование (четвертая версия)









Итоги

Сравнение с эталоном лучших временных показателей разных версий на массиве быстрых данных размера 2*10^7

Версия	Modify	State
1 (t = 2)	+49%	+782%
2 (t = 2)	-2%	+419%
3 (t = 2)	2%	17%
3* (t =2+(2+2))	8%	75%
4 (s = 2)	12%	10%
4 (s= 3)	2%	24%

Сравнение с эталоном лучших временных показателей разных версий на массиве медленных данных размера 2*10^7

Версия	Modify	State
1 (t = 5)	-26%	4%
2 (t = 4)	-22%	-1%
3 (t =4 или 10)	-24%	-20%
3* (t =2+(2+2))	-19%	-13%
4 (s = 2)	-20%	-19%
4 (s= 3)	-26%	-25%

Заключение

В процессе выполнения курсовой работы были получены следующие результаты:

- Изучены основные особенности работы многопоточного приложения;
- Разработаны 5 вариантов требуемой структуры;
- Протестирована корректность работы разработанных структур;
- Проведены оценки временных характеристик разработанных структур.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Rivest R. L. Stein C. Cormen T. H., Leiserson C. E. Introduction to algorithms, 2022.
[2] ThreadSanitizer documentation [Electronic resource]. Mode of access: https://clang.llvm.org/docs/ThreadSanitizer.html. Date of access: 16.05.2024.
[3] C++ documentation [Electronic resource]. Mode of access: https://en.cppreference.com/w/. Date of access: 16.05.2024.
[4] Тель Ж. Введение в распределенные алгоритмы, 2009.

Спасибо за внимание!