МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра дискретной математики и алгоритмики

ИСПОЛЬЗОВАНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ДЛЯ УСКОРЕНИЯ МАССОВЫХ ОПЕРАЦИЙ С ОДНОРОДНЫМИ ДАННЫМИ

Курсовая работа

Азявчикова Алексея Павловича
студента 3-го курса
специальности 1-31 03 04
«Информатика»

Научный руководитель:
старший преподаватель
И. Д. Лукьянов

Минск, 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном информационном обществе, где объемы данных постоянно возрастают, эффективная обработка и анализ информации становятся все более важными задачами для разработчиков программного обеспечения. В этом контексте параллельные алгоритмы, т.е. алгоритмы, которые могут быть реализованы по частям на множестве различных вычислительных устройств, выделяются как один из методов ускорения операций с данными. А области их применения в данный момент активно расширяются за счет того, что параллельные компьютеры, т.е. компьютеры с несколькими устройствами обработки данных, становятся все более распространенными и охватывают широкий диапазон цен и производительности.

Целью данной курсовой работы является разработка параллельной версии дерева отрезков для ускорения обработки массовых операций с однородными данными, покрытие её тестами и оценка временных характеристик основных операций над нею.

ГЛАВА 1
ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ПОНЯТИЯ

1.1 Основные понятия

Определение 1. *Поток* - это наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Определение 2. *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.

Определение 3. *Работой* многопоточного вычисления называется общее время выполнения всего вычисления на одном процессоре, исполняющим лишь один поток в каждый момент времени. [3]

Определение 4. *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. [3]

Определение 5. *Атомарные операции* — операции, выполняющиеся как единое целое либо не выполняющиеся вовсе. Т. е. это операция, во время исполнения которой данные, читаемые/изменяемые этой операцией не могут быть изменены другой операцией.

1.2 Постановка задачи

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

Другими словами, структура должна поддерживать следующие виды операций:

1. $ModifyTree(L, R, Modifier)$: модифицировать данные на отрезке $[L, R)$ в соответствие со значением $Modifier$ (далее будем называть этот тип запроса запросом модификации);

2. $GetTreeState(L, R)$: вернуть состояние отрезка $[L, R)$ (далее будем называть этот тип запроса запросом состояния).

Данные операции должны быть реализованы поверх переданной пользователем в шаблон политики модификации данных и вычисления состояния на отрезке, удовлетворяющей следующему интерфейсу:

1. $GetState(first\_state, second\_state)$: вернуть состояние отрезка, состоящего из двух элементов: $first\_state$, $second\_state$. Данная операция должна быть ассоциативной;

2. $GetNullState()$: вернуть ноль относительно операции взятия состояния (такой элемент должен существовать). Иными словами, должно выполняться: $\forall S : GetState(S, GetNullState()) = GetState(GetNullState(), S) = S$;

3. $GetModifiedState(init\_state, values\_count, modifier)$: вернуть модифицированное состояние отрезка состоящего из $values\_count$ элементов и имеющего до модификации состояние $init\_state$, в соответствие со значением $modifier$. Данная функция должна выдавать результат аналогичный тому, который был бы получен в результате многократного применения к $init\_state$ операции $GetState$ для каждого элемента отрезка ($values\_count$ раз).

Введем обозначения: n - число элементов массива, t - число используемых потоков.

1.3 Области применения параллельных алгоритмов

Примеры областей применения параллельных алгоритмов:

• Базы данных. С увеличением объемов данных в современных информационных системах, параллельные алгоритмы используются для ускорения запросов к базе данных, путем одновременной обработки нескольких строк таблиц базы на различных вычислительных устройствах, что в последствии влияет на производительность тех приложений, которые совершают запросы к базе;

• Машинное обучение и анализ данных. В области машинного обучения и анализа данных параллельные алгоритмы используются для обучения моделей на больших наборах данных, таких как наборы изображений, текстов и звуков. Выполнение параллельных вычислений позволяет ускорить процесс обучения и анализа, что важно для получения быстрых и точных результатов.

• Вычислительная физика и научные расчеты. В области физики, химии, астрономии и других научных дисциплин, параллельные алгоритмы применяются для моделирования сложных процессов, расчетов структурной оптимизации и анализа больших объемов экспериментальных данных.

• Сетевое программирование и распределенные системы [5]: в сетевом программировании и разработке распределенных систем, параллельные алгоритмы применяются для эффективной обработки запросов и обмена данными между различными узлами сети.

• Графические приложения и игры. В области разработки графических приложений и игр, параллельные алгоритмы применяются для обеспечения плавной отрисовки графики, выполнения физических расчетов и обработки пользовательских взаимодействий.

ГЛАВА 2

ПЕРВЫЙ ВАРИАНТ РЕАЛИЗАЦИИ ПАРАЛЛЕЛЬНОЙ СТРУКТУРЫ ДАННЫХ

2.1 Вспомогательная структура данных буферизированный канал

Для удобной работы с несколькими потоками был реализован примитив синхронизации буферизированный канал (см. код в приложениях), предоставляющий пользователю следующий интерфейс:

• Send(value) — отправить value в канал. При этом если буфер канала заполнен, то отправляющий поток будет заблокирован до тех пор, пока не появится свободное место в буфере;

• Recv() — получить ранее отправленное значение из канала. При этом если буфер канала пуст, то поток будет заблокирован до тех пор, пока в буфере не появятся значения или пока канал не будет закрыт. Во втором случае будет возвращен пустой std::optional;

• Close() — закрыть канал. При этом новые попытки отправить значение в канал будут вызывать ошибку, однако если на момент закрытия в буфере канала оставались значения, то вызов функции Recv() не вызовет ошибки и вернет значение из буфера.

Данная структура отвечает на запросы Send, Recv и Close за $O(1)$ времени и потребляет $O(buff\_size)$ памяти, где $buff\_size$ - размер внутреннего буфера, задаваемый пользователем.

2.2 Описание

Пусть задана политика модификации данных, далее именуемая просто policy. И задан массив однородных данных array. (Предполагается, что $n \geqslant t$)

Разобьем исходный массив на подмассивы размера $\lfloor n/t \rfloor$, а оставшиеся элементы (если n не делится нацело на t) положим в последний массив. Далее построим для каждого подмассива дерево отрезков [4] и сохраним указатели на все деревья в отдельный массив. Создадим также $\lfloor n/t \rfloor$ буферизированных каналов, хранящих в себе запросы модификации и состояния описанные общей структурой Query (см. приложения), каждый из которых будет отвечать за передачу запросов модификации или состояния к одному подмассиву. Теперь создадим t потоков, каждый из которых будет отвечать за модификации одного подмассива. Внутри одного потока будем вызывать у соответствующего буферизированного канала метод Recv() до тех пор, пока канал не будет закрыт (это

можно понять по возвращенному пустому std::optional) для того, чтобы узнать о новом запросе к подмассиву. Получив же новый запрос, в зависимости от его типа (модификации или состояния) поток вызовет у соответствующего дерева необходимый метод и, при необходимости (если произошел запрос состояния), изменит значение переменной responce.state, воспользовавшись функцией policy.GetState. Также поток увеличит атомарный счетчик responce.counter, и в случае, если до увеличения его значение на единицу отличалось от t, положит в respoce.result результат запроса ко всей структуре, лежащей в переменной respoce.state. Стоит отметить, что ни один из потоков не использует иных примитивов синхронизации (например, std::mutex) потому как каждый поток работает с одним конкретным деревом, а следовательно не может случиться двух запросов к одному дереву из разных потоков.

Теперь же, в момент получения от пользователя запроса модификации или состояния, нам остаётся положить в каждый из каналов запросы соответствующие выбранным подмассивам, а потоки, получив эти запросы, обработают их и положат ответ в std::promice, если это был запрос состояния. Таким образом, можно реализовать неблокирующие операции над массивом однородных данных.

Корректность данного алгоритма следует из корректности работы дерева отрезков и того факта, что каждый поток, обрабатывающий запросы, работает с одним, привязанным к нему деревом, из чего следует отсутствие data race-ов.

2.3 Оценки

2.3.1 Временная сложность

• Работа (см. определения) ModifyTree составляет $O(t + t * log(n/t))$ так как всего совершается t запросов к деревьям отрезков над массивами размера $O(n/t)$ (каждый работает за $O(log(n/t))$ по свойствам дерева отрезков), а также выдается t запросов потокам. А интервал $ModifyTree$ составляет $O(t + log(n/t))$ так как каждый поток совершает запрос к одному дереву отрезков над массивом размера $O(n/t)$, а самый медленный ещё должен дождаться выдачи ему запроса, что произойдет за $O(t)$ времени.

• Работа GetTreeState составляет $O(t+t*log(n/t))$, а интервал GetTreeState составляет $O(t + log(n/t))$ (рассуждения аналогичны ModifyTree).

2.3.2 Затраты памяти

Затраты памяти составляют $O(n+t+buff\_size)$, так как каждое из t поддеревьев занимает $O(n/t)$ памяти, так как оперирует с массивом размера $O(n/t)$, а также расходуется $O(buff\_size)$ памяти на каждый буферизированный канал.

ГЛАВА 3

ВТОРОЙ ВАРИАНТ РЕАЛИЗАЦИИ ПАРАЛЛЕЛЬНОЙ ПАРАЛЛЕЛЬНОЙ СТРУКТУРЫ ДАННЫХ

3.1 Вспомогательная структура данных буферизированная очередь

Для удобной работы с несколькими потоками был реализован примитив синхронизации буферизированная очередь (см. код в приложениях), предоставляющий пользователю следующий интерфейс:

- 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.

Данная структура отвечает на запросы Enqueue и Dequeue за $O(1)$ времени и потребляет $O(\text{buff\_size})$ памяти, где buff_size - размер внутреннего буфера, задаваемый пользователем и являющимся при этом степенью двойки (это обязательное требование к размеру).

Отличием данной структуры от буферизированного канала являются неблокирующие вызовы Enqueue и Dequeue, позволяющие сэкономить время на системных вызовах внутри структуры, тем самым обеспечив лучшие временные показатели.

3.2 Описание

Данная реализация аналогично предыдущей версии разбивает исходный массив на подмассивы в соответствие с числом используемых потоков, однако задани потокам выдаются уже с помощью буферизированных очередей. В данной версии потоки после своего создания и до уведомления их о необходимости завершения работы через созданную переменную типа std::atomic_flag постоянно проверяют соответствующую очередь на заполненность, вызывая метод Dequeue. И в момент, когда метод вернет true, поток, получивший новый запрос, начинает его исполнение по аналогичным предыдущей реализации правилам.

Основное отличие данной реализации в том, что потоки никогда не "засыпают" (вызывая метод wait у соответствующего std::condition_variable), как это было раньше, что приводит к дополнительной нагрузке на процессор, однако позволяет улучшить временные характеристики структуры, как мы убедимся далее.

3.3 Оценки

3.3.1 Временная сложность

• Работа ModifyTree составляет $O(t + t * log(n/t))$ так как всего совершается t запросов к деревьям отрезков над массивами размера $O(n/t)$ (каждый работает за $O(log(n/t))$ по свойствам дерева отрезков), а также выдается t запросов потокам. А интервал $ModifyTree$ составляет $O(t + log(n/t))$ так как каждый поток совершает запрос к одному дереву отрезков над массивом размера $O(n/t)$, а самый медленный ещё должен дождаться выдачи ему запроса, что произойдет за $O(t)$ времени.

• Работа GetTreeState составляет $O(t * log(n/t))$, а интервал GetTreeState составляет $O(log(n/t))$ (рассуждения аналогичны ModifyTree).

3.3.2 Затраты памяти

Затраты памяти составляют $O(n+t+buff\_size)$, так как каждое из t поддеревьев занимает $O(n/t)$ памяти, так как оперирует с массивом размера $O(n/t)$, а также расходуется $O(buff\_size)$ памяти на каждую буферизированную очередь.

ГЛАВА 4

ТРЕТИЙ ВАРИАНТ РЕАЛИЗАЦИИ ПАРАЛЛЕЛЬНОЙ СТРУКТУРЫ ДАННЫХ

4.1 Описание

В данной реализации мы также разобьем исходный массив на подмассивы, по каждому из которых построим дерево отрезков, закрепив за ним поток-обработчик запросов.

Однако в отличие от предыдущих реализаций данная версия полностью отказывается от попыток выдачи задач потокам обработчикам. Вместо этого потоки обработчики сами определяют наличие новой задачи. Это происходит за счет введения новой атомарной переменной: идентификатора исполняемой задачи (std::atomic<int64_t> [1]). Потоки обработчики сканируют этот индекс до тех пор, пока он не изменится (что будет свидетельствовать о выдаче новой задачи) или пока не будет выставлен флаг (std::atomic_flag [1]), обозначающий, что потокам стоит прекратить свою работу. В момент же изменения идентификатора, поток обработчик, заметивший это изменение, смотрит на общую для всех потоков атомарную переменную запроса и определяет, попадает ли подмассив под запрос, и, если попадает, выполняет запрос и обновляет общую атомарную переменную состояния, а в конце всех проделанных операций вне зависимости от того, попадал ли подмассив под запрос, увеличивает атомарный счетчик числа потоков, выполнивших свою задачу. Поток, увеличивший этот счетчик до числа используемых потоков выставляет атомарный флаг, свидетельствующий о том, что запрос к структуре выполнен.

Таким образом, потоку, выдающему задачи требуется дождаться выполнения предыдущего запроса, обновить переменную запроса и увеличить идентификатор исполняемой задачи. И, если требуется (если это был запрос состояния), дождаться завершения запроса.

4.2 Оценки

4.2.1 Временная сложность

• Работа ModifyTree составляет $O(t * log(n/t))$ так как всего совершается t запросов к деревьям отрезков над массивами размера $O(n/t)$ (каждый работает за $O(log(n/t))$ по свойствам дерева отрезков). А интервал $ModifyTree$ составляет $O(log(n/t))$ так как каждый поток совершает запрос к одному дереву отрезков над массивом размера $O(n/t)$.

• Работа GetTreeState составляет $O(t * log(n/t))$, а интервал - $O(log(n/t))$ (рассуждения аналогичны ModifyTree).

4.2.2 Затраты памяти

Затраты памяти составляют $O(n + t)$, так как каждое из t поддеревьев занимает $O(n/t)$ памяти, так как оперирует с массивом размера $O(n/t)$, а также расходуется $O(1)$ памяти на общие для всех потоков переменные флагов и счетчиков, а также на переменную запроса.

4.3 Модификация структуры

Идея данной реализации заключается в том, чтобы при разбиении исходного массива на подмассивы строить по ним не обычные деревья отрезков, а многопоточные. В качестве таких деревьев и будет использован третий вариант реализации.

Далее введем обозначения: t1 - число потоков в исходной структуре, t2 - число потоков в дочерних структурах.

4.3.1 Временная сложность

• Работа ModifyTree составляет $O(t1 * t2 * log(n/(t1 * t2)))$ так как всего совершается $t1 * t2$ запросов к деревьям отрезков над массивами размера $O(n/(t1 * t2))$. А интервал $ModifyTree$ составляет $O(log(n/(t1 * t2)))$ так как каждый поток совершает запрос к одному дереву отрезков над массивом размера $O(n/(t1 * t2))$.

• Работа GetTreeState составляет $O(t1 * t2 * log(n/(t1 * t2)))$, а интервал - $O(log(n/(t1 * t2)))$ (рассуждения аналогичны ModifyTree).

4.3.2 Затраты памяти

Затраты памяти составляют $O(n + t1 * t2)$, так как каждое из $t1 * t2$ поддеревьев занимает $O(n/(t1 * t2))$ памяти, а также расходуется $O(1)$ памяти на общие для всех потоков переменные флагов и счетчиков, а также на переменную запроса.

ГЛАВА 5

ЧЕТВЕРТЫЙ ВАРИАНТ РЕАЛИЗАЦИИ ПАРАЛЛЕЛЬНОЙ ПАРАЛЛЕЛЬНОЙ СТРУКТУРЫ ДАННЫХ

5.1 Описание

Данная реализация аналогично предыдущей версии разбивает исходный массив на подмассивы в соответствие с числом используемых потоков, однако использует всего 3 потока в своей реализации и опирается на тот факт, что дерево отрезков может обработать запрос за $O(1)$ в случае, если этот запрос затрагивает весь массив.

Идея в следующем: если какой-то запрос затрагивает более 3 подмассивов, то теоретическую оценку в логарифм дают крайние подмассивы, а подмассивы между ними обрабатываются за $O(1)$. Поэтому можно попробовать выдать тяжелые краевые задачи двум потокам (первому и третьему), а оставшемуся потоку (второму) - легкие задачи по модификации оставшихся поддеревьев.

Далее введем обозначение s - (число подмассивов, на которые был разбит исходный массив) минус 2. Т.е. это то количество поддеревьев, которые придется обработать второму потоку в худшем случае

5.2 Оценки

5.2.1 Временная сложность

• Работа ModifyTree составляет $O(s+2*log(n/(s+2)))$ так как всего совершается 2 долгих запроса к деревьям отрезков над массивами размера $O(n/(s+2))$ (каждый работает за $O(log(n/(s+2))$ по свойствам дерева отрезков), а также выдается s легких запросов второму потоку, работающих за $O(1)$. А интервал $ModifyTree$ составляет $O(max(log(n/(s+2), s))$ - по определению, так как два потока отработают за $log(n/(s+2)$, а оставшийся - за $O(s)$.

• Работа GetTreeState составляет $O(s+2*log(n/(s+2)))$, а интервал GetTreeState составляет $O(log(n/(s+2))$ (рассуждения аналогичны ModifyTree).

5.2.2 Затраты памяти

Затраты памяти составляют $O(n)$, так как каждое из $s+2$ поддеревьев занимает $O(n/(s+2))$ памяти.

ГЛАВА 6
ТЕСТИРОВАНИЕ РЕАЛИЗОВАННЫХ СТРУКТУР

6.1 Тестирование корректности работы

Тестирование корректности работы реализованных структур проводилось с помощью рандомизированных тестов и потокового санитайзера для C++ [2]. Также некоторое количество тестов было написано вручную. Для тестирования использовалась библиотека gtest.

6.2 Временные оценки

Для измерения времени работы структур использовались тесты, работающие с типом данных int и операцией присваивания в качестве операции модификации, и операцией суммы в качестве операции состояния.

Для того, чтобы протестировать работу структуры с данные, операции с которыми проводятся медленнее, чем с int (что может происходить, например, когда данные лежат на жестком диске или на удаленном сервере) была написана политика модификации, замедляющая все операции с int на 500 наносекунд. Такие тесты в дальнейшем будем называть тестами на медленных данных.

Тестирование проводилось на машине, позволяющей одновременно исполнять не более 12 потоков.

6.2.1 Временные оценки эталонной однопоточной структуры

Таблица 6.2.1 – Среднее время работы ModifyTree в однопоточном дереве в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
1	930	994	1068	1133	1187

Таблица 6.2.2 – Среднее время работы GetTreeState в однопоточном дереве в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
1	629	667	707	776	811

Таблица 6.2.3 – Среднее время работы ModifyTree в однопоточном дереве в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
1	38146	40356	42638	45875	48027

Таблица 6.2.4 – Среднее время работы GetTreeState в однопоточном дереве в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
1	16214	17355	18487	19710	20712

6.2.2 Временные оценки первой структуры

Таблица 6.2.5 – Среднее время работы ModifyTree в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	1253	1346	1366	1675	1697
3	1861	1834	1878	1735	1618
4	2698	2589	2591	2612	2528
5	3451	3579	3599	3422	3332

Таблица 6.2.6 – Среднее время работы GetTreeState в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	7201	7048	7088	7156	7137
3	7765	7888	7932	8278	8369
4	9472	8982	9004	9111	9694
5	10365	10045	10058	10293	9992

Как видно из приведенных выше измерений, данная реализация не дает выигрыша по времени по сравнению с классической реализацией, а при увеличении числа потоков даже ухудшает свои временные показатели. Это связано с тем, что написанный ранее примитив синхронизации буферизированный канал - это очень дорогой инструмент для выдачи задач потокам, так как внутри он использует системные вызовы, чтобы блокировать и пробуждать потоки в момент поступления новых запросов.

Таблица 6.2.7 – Среднее время работы ModifyTree в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	31429	32493	33600	37996	38952
3	34809	35790	38589	42228	43380
4	30533	31478	32565	37093	38073
5	28974	30048	31476	33786	35220
6	33073	35131	36669	40406	42540
7	33635	35777	36922	41226	42570

Таблица 6.2.8 – Среднее время работы GetTreeState в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	19084	19748	20288	22033	22846
3	20313	21282	21866	24899	25736
4	18988	19545	20592	22325	22333
5	18989	19451	19900	21596	21480
6	19680	20934	21634	24563	24680
7	19721	20575	21359	24626	25487

Как видно из приведенных выше измерений, данная реализация дает выигрыш по времени по сравнению с классической реализацией при использовании 5 потоков на запросах модификации с медленными данными (время работы уменьшилось примерно на 26%). Однако увеличение числа потоков, как и ранее, приводит к замедлению структуры.

6.2.3 Временные оценки второй структуры

Таблица 6.2.9 – Среднее время работы ModifyTree в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	916	957	955	1111	1121
4	898	924	964	1107	1150
6	1028	1120	1153	1265	1354
8	1397	1319	1366	1950	2027
10	1653	1797	1811	1812	1901

Таблица 6.2.10 – Среднее время работы GetTreeState в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	3723	4104	3985	4053	4199
4	4350	4779	4806	5080	5347
6	6816	6727	6825	6932	6961
8	7905	8201	7607	7757	8361
10	8838	9009	9678	9451	8660

Как и предыдущая, данная структура работает медленнее однопоточной версии на быстрых данных (времени работы запросов состояния увеличилось примерно на 400%), однако она работает значительно быстрее первой версии (в сравнении с запросами состояния), предоставляя при этом тот же интерфейс неблокирующих запросов.

Таблица 6.2.11 – Среднее время работы ModifyTree в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	30865	31948	33030	37528	38754
4	29898	30999	32220	36321	37379
6	31431	34190	35290	39105	41844
8	30753	31822	33138	35663	38735
10	29191	30456	32285	34483	35725

Таблица 6.2.12 – Среднее время работы GetTreeState в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	17221	18013	18531	20440	21067
4	17810	17749	18275	20454	20359
6	18267	19259	20173	22746	24010
8	19337	20980	21260	22723	24888
10	20443	20744	21944	22066	23306

Как видно из приведенных выше измерений, на медленных данных вторая структура работает быстрее эталонной при использовании 4 потоков на запросах модификации (время работы уменьшилось примерно на 22%) и не сильно хуже на запросах состояния (время работы увеличилось примерно на 1%).

6.2.4 Временные оценки третьей структуры

Таблица 6.2.13 – Среднее время работы ModifyTree в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	925	927	981	1152	1164
4	1003	1052	1072	1215	1263
6	1166	1246	1271	1381	1456
8	1200	1200	1258	1681	1980
10	1231	1568	1565	1696	1293

Таблица 6.2.14 – Среднее время работы GetTreeState в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2	872	829	843	991	945
4	835	858	880	970	1080
6	1099	1129	1147	1187	1235
8	1222	1268	1238	1420	1427
10	1335	1418	1377	1516	1444

Данная структура работает значительно лучше предыдущих на быстрых данных (Показатели по операциям модификации отличаются в худшую сторону от эталонных не более чем на 2%, а по операциям состояния - не более чем на 18%)

Таблица 6.2.15 – Среднее время работы ModifyTree в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	31275	32575	33680	38234	39265
4	30512	31508	32571	36989	38327
6	32187	35185	36384	40388	43118
8	31325	32801	33831	36474	40033
10	30075	31397	32288	36013	36346

Таблица 6.2.16 – Среднее время работы GetTreeState в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2	13839	14362	14954	16639	17248
4	13396	13960	14530	16248	16768
6	13562	14808	15282	18030	19171
8	13075	13670	15178	17261	17807
10	13422	14152	15039	16823	16402

Из тестов видно, что по сравнению с первой или второй версией данный подход показывает себя намного лучше. Используя 4 или 10 потоков удалось добиться более чем 24% уменьшения времени по операциям модификации и 20% уменьшения времени по операциям состояния. Также данный подход лучше масштабируется на большее число потоков.

6.2.5 Временные оценки модификации третьей структуры

Таблица 6.2.17 – Среднее время работы ModifyTree в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2 + (2 + 2)	1010	1060	1116	1206	1234
2 + (3 + 3)	1198	1165	1189	1421	2226

Таблица 6.2.18 – Среднее время работы GetTreeState в наносекундах.

T | N	1000000	2000000	4000000	10000000	20000000
2 + (2 + 2)	1181	1246	1278	1361	1416
2 + (3 + 3)	1345	1499	1420	1533	1602

Таблица 6.2.19 – Среднее время работы ModifyTree в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2 + (2 + 2)	30660	32005	33064	37595	38546
2 + (3 + 3)	32533	37495	36558	42882	43463

Таблица 6.2.20 – Среднее время работы GetTreeState в наносекундах на медленных данных.

T | N	1000000	2000000	4000000	10000000	20000000
2 + (2 + 2)	14474	15053	15615	17423	17994
2 + (3 + 3)	14870	16669	17345	20655	21784

6.2.6 Временные оценки четвертой структуры

Таблица 6.2.21 – Среднее время работы ModifyTree в наносекундах.

S | N	1000000	2000000	4000000	10000000	20000000
1	1148	1172	1221	1403	1400
2	1023	1058	1120	1263	1277
3	1021	1032	1045	1147	1165
4	1636	1901	1812	2214	2275

Таблица 6.2.22 – Среднее время работы GetTreeState в наносекундах.

S | N	1000000	2000000	4000000	10000000	20000000
1	985	1025	1085	1135	1177
2	842	862	886	966	978
3	874	904	914	949	1001
4	1262	1370	1392	1590	1682

Таблица 6.2.23 – Среднее время работы ModifyTree в наносекундах на медленных данных.

S | N	1000000	2000000	4000000	10000000	20000000
1	35066	35919	38598	42520	43625
2	30477	31604	32672	37268	38237
3	30504	30366	30919	35482	35357
4	53799	60555	60485	71618	78243

Таблица 6.2.24 – Среднее время работы GetTreeState в наносекундах на медленных данных.

S | N	1000000	2000000	4000000	10000000	20000000
1	14423	15039	16122	19047	19518
2	13312	13871	14406	16146	16711
3	12648	13208	13771	14894	15475
4	16973	19292	19275	28118	30437

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.

Программный код реализованных структур и тесты к ним выложены на github: https://github.com/AzyavchikovAlex/Coursework_6sem.

В дальнейшем развитие курсовой работы будет связано с разработкой многопоточных алгоритмов для ускореняя работы более долгих алгоритмов, чем дерево отрезков.

Буферизированный канал

```cpp
template<class T>
class BufferedChannel {
 public:
  explicit BufferedChannel(uint32_t size) : max_size_(size) {
  }

  template<typename K>
  void Send(K&& value) {
    std::unique_lock lock(global_);
    send_cv_.wait(lock,
                  [this]() { return is_closed_ || queue_.size() < max_size_; });
    if (is_closed_) {
      throw std::runtime_error("Channel is closed");
    }
    queue_.push(std::forward<K>(value));
    receive_cv_.notify_one();
  }

  std::optional<T> Recv() {
    std::unique_lock lock(global_);
    receive_cv_.wait(lock, [this]() { return !queue_.empty() || is_closed_; });
    if (queue_.empty()) {
      return std::nullopt;
    }
    T result(std::move(queue_.front()));
    queue_.pop();
    send_cv_.notify_one();
    return result;
  }

  void Close() {
    std::lock_guard lock(global_);
    is_closed_ = true;
    send_cv_.notify_all();
    receive_cv_.notify_all();
  }

 private:
  std::mutex global_;
  std::condition_variable send_cv_, receive_cv_;
  std::queue<T> queue_;
  const uint32_t max_size_{1};
  bool is_closed_{false};
};
```

Буферизированная очередь

```cpp
#pragma once

#include <atomic>
#include <vector>
#include <thread>

template<class T>
class FastQueue {
 private:
  struct Node;

 public:
  explicit FastQueue(size_t size) : queue_(size), mod_(size - 1) {
    shift_ = 0;
    while (size > 1) {
      ++shift_;
      size >>= 1;
    }
  }

  bool Enqueue(const T& value) {
    for (uint64_t push_index = push_index_.load(std::memory_order_acquire);;) {
      auto& node = queue_[push_index & mod_];
      uint64_t expected_epoch = (push_index >> shift_) << 1;
      if (expected_epoch == node.epoch.load(std::memory_order_acquire)) {
        if (push_index_.compare_exchange_strong(push_index, push_index + 1)) {
          node.value = value;
          node.epoch.fetch_add(1, std::memory_order_release);
          return true;
        }
        std::this_thread::yield();
      } else {
        uint64_t old_push_index = push_index;
        push_index = push_index_.load(std::memory_order_acquire);
        if (old_push_index == push_index) {
          return false;
        }
      }
    }
  }

  bool Dequeue(T& data) {
    for (uint64_t pop_index = pop_index_.load(std::memory_order_acquire);;) {
      auto& node = queue_[pop_index & mod_];
      uint64_t expected_epoch = 1 + ((pop_index >> (shift_) << 1));
      if (expected_epoch == node.epoch.load(std::memory_order_acquire)) {
        if (pop_index_.compare_exchange_strong(pop_index, pop_index + 1)) {
          data = std::move(node.value);
          node.epoch.fetch_add(1, std::memory_order_release);
          return true;
        }
        std::this_thread::yield();
      } else {
        uint64_t old_pop_index = pop_index;
        pop_index = pop_index_.load(std::memory_order_acquire);
```

```
        if (old_pop_index == pop_index) {
          return false;
        }
      }
    }
  }

 private:
  struct Node {
    char pad1[64];
    std::atomic<uint64_t> epoch;
    char pad2[64];
    T value;
    char pad3[64];
  };
  char pad1_[1024];
  std::vector<Node> queue_;
  char pad2_[1024];
  std::atomic<uint64_t> push_index_ = 0;
  char pad3_[1024];
  std::atomic<uint64_t> pop_index_ = 0;
  char pad4_[1024];
  const uint64_t mod_;
  char pad5_[1024];
  size_t shift_;
  char pad6_[1024];
};
```

ПРИЛОЖЕНИЕ В

Структура запроса в первой и второй реализациях

```
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};
  };
```

ПРИЛОЖЕНИЕ Г

Общий интерфейс реализованных структур.

```
template<typename T, typename M, typename P, size_t ThreadsCount = 0>
class AbstractTree {
 public:
  virtual void ModifyTree(size_t l, size_t r, M modifier) = 0;
  virtual T GetTreeState(size_t l, size_t r) = 0;
  virtual std::future<T> GetFutureTreeState(size_t l, size_t r) {
    std::promise<T> result;
    result.set_value(GetTreeState(l, r));
    return result.get_future();
  }
};
```

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] Дерево отрезков [Electronic resource]. Mode of access: http://e-maxx.ru/algo/segment_tree. Date of access: 16.05.2024.

[5] Тель Ж. Введение в распределенные алгоритмы, 2009.