Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Московский физико-технический институт
(национальный исследовательский университет)»

Физтех-школа Прикладной Математики и Информатики

Кафедра банковских информационных технологий

Интерполяционное дерево поиска с параллельным применением набора операций

(бакалаврская работа)

Направление: 03.03.01 «Прикладные математика и физика»

Выполнил студент гр. 791 _____ А.Р. Марценюк

Научный руководитель,
к.т.н. _____ В.Е. Аксенов

Москва – 2021

Аннотация

Цель исследования — разработать структуру интерполяционного дерева поиска для работы с параллельным применением набора запросов на вставку, удаление и поиск.

Решаемые задачи — разработать интерполяционное дерево поиска с параллельным применением набора операций; оценить асимптотическую сложность работы разработанной структуры данных; реализовать разработанную структуру данных; оценить возможности масштабирования представленной реализации на практике.

В рамках исследования была разработана и реализована структура интерполяционного дерева поиска с параллельным применением набора операций вставки, удаления и поиска. Показано, что предложенная структура является оптимальной в плане работы и обладает полилогарифмическим спаном. Экспериментально доказана высокая масштабируемость предложенной структуры.

Разработанное дерево поиска может стать хорошей альтернативой существующим решениям для реализации индексов в базах данных, а также быть использовано в других задачах с необходимостью быстрого параллельного применения заранее заданного набора операций.

Оглавление

Введение

В современных базах данных для оптимизации времени выполнения запросов широко используются индексы, которые, как правило, реализованы при помощи одного из следующих сбалансированных деревьев: AVL, B-tree, Treap и др. Лучшее время выполнения операций, которого позволяют добиться используемые в настоящее время решения — $O(\log n)$ при $O(n)$ памяти.

В этой работе мы рассмотрели идейно отличающуюся от популярных решений структуру данных под названием «Интерполяционное дерево поиска», ожидаемое время выполнения вставки, удаления и поиска по которой равняется $O(\log \log n)$ при тех же $O(n)$ памяти [1]. Кроме того, в работе описано как изменить такую структуры данных, чтобы она могла выполнять набор операций параллельно.

Глава 1

Интерполяционное дерево поиска

В этой секции мы рассмотрим последовательную реализацию интерполяционного дерева поиска (Interpolation Search Tree, IST) и обсудим алгоритмы поиска, вставки и удаления.

Интерполяционное дерево поиска — это сильно ветвистое дерево, в котором степень вершины зависит от мощности множества, расположенного ниже. Степень корня дерева в таком случае равняется $\Theta(\sqrt{n})$. В *идеальном* IST все прямые потомки-поддеревья одной вершины имеют примерно одинаковые размеры. В этом случае размеры поддеревьев, чьим предком является корневая вершина, равны $\Theta(\sqrt{n})$. Следовательно, поддерево-потомок корневой вершины идеального IST имеет степень $\Theta(\sqrt{\sqrt{n}})$.

Определение 1. *Пусть a и b — действительные числа такие, что $a < b$. Интерполяционное дерево поиска (IST) с максимальным и минимальным элементами a и b соответственно для множества $S = \{x_1 < x_2 < \cdots < x_n\} \subseteq [a, b]$ из n элементов состоит из*

(1) Набора R элементов корня $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, $i_1 < i_2 < \cdots < i_k$, записанных в массив $R[1 \ldots k]$, таких, что $R[j] = x_{i_j}$. Кроме того, k удовлетворяет неравенству $\sqrt{n}/2 \leq k \leq 2\sqrt{n}$.

(2) Интерполяционных деревьев поиска $T_1, \ldots, T_{k+1}$ для подмножеств $S_1, \ldots, S_{k+1}$, где $S_j = \{x_{i_{j-1}+1}, \ldots, x_{i_j-1}\}$ для $2 \leq j \leq k$, $S_1 = \{x_1, \ldots, x_{i_1-1}\}$, $S_{k+1} = \{x_{i_k+1}, \ldots, x_n\}$. Кроме того, дерево T_j $(2 \leq j \leq k)$ имеет границы $x_{i_{j-1}}$ и x_{i_j}, T_1 имеет границы a и x_1, и T_{k+1} — границы x_k и b.

(3) *Массива* $ID[1 \dots m]$, *где* m — *целое число, составленного по принципу* $ID[i] = j \Leftrightarrow R[j] < a + i(b-a)/m \leq R[j+1]$.

Массив R состоит из элементов некоторого подмножества множества S. Для идеального S мы потребуем, чтобы это подмножество было равномерно распределено по S.

Также потребуем, чтобы m равнялось n^α для некоторого α при $\frac{1}{2} \leq \alpha < 1$.

Определение 2. *Интерполяционное дерево поиска с параметром* α *для множества* S *с мощностью* n *называется идеальным, если* $i_j = j\lfloor\sqrt{n}\rfloor$ *для всех* $j \geq 1$, *если* $m = \lceil n^\alpha \rceil$ *и деревья* $T_1, \dots, T_{k+1}$ *также являются идеальными.*

Теорема 1. *Пусть* $\frac{1}{2} \leq \alpha < 1$. *Тогда идеальное интерполяционное дерево поиска для упорядоченного множества* S, $|S| = n$, *может быть построено за время* $O(n)$ *и требует* $O(n)$ *памяти. Глубина такого дерева:* $O(\log\log n)$.

1.1. Операции вставки и удаления

Перейдем к описанию алгоритмов вставки и удаления. Для этого последуем очень простому принципу. Предположим, что мы имеем идеальное интерполяционное дерево поиска. Первые K операций вставки и поиска мы оставляем корневую вершину без изменений, помечая элементы при удалении и вставляя новые в листья, после чего перестраиваем дерево. По такому же принципу работаем со всеми вершинами-наследниками.

Определение 3.

(1) *Пусть* v — *вершина интерполяционного дерева поиска, тогда* T_v — *поддерево с корнем* v.

(2) *Вес $w(v)$ вершины — это число всех элементов (и помеченных, и нет), которые хранятся в T_v. Размер вершины v – это число непомеченных элементов, которые хранятся в T_v.*

(3) *Перестройка T_v означает замену поддерева T_v идеальным IST для набора непомеченных элементов, хранящихся в T_v.*

В представленной реализации интерполяционного дерева поиска мы поддерживаем для каждой вершины v дерева счетчик $C(v)$, который отображает число вставок и удалений, произведенных в поддереве T_v. Счетчик $C(v)$ переполняется, когда его значение достигает W/Z, где W — размер поддерева T_v в момент его постройки (т.е. в последний раз, когда счетчик $C(v)$ был обнулен), Z — эмпирически подобранная константа. Сразу после того, как счетчик $C(v)$ оказался переполнен, перестраиваем соответствующее поддерево T_v и инициализируем счетчики всех его вершин нулями.

Далее приведем описания алгоритмов вставки и удаления.

Алгоритм *вставки ключа x:*

(1) Проверяем, находится ли ключ x в текущей вершине. Если ключ там уже есть и не помечен — заканчиваем выполнение процедуры и возвращаем *false*. В случае, если ключ в вершине есть, но помечен — убираем с него метку, уменьшаем счетчик $C(v)$, увеличиваем вес текущей вершины на один и возвращаем *true*.

(2) Проверяем, не переполнится ли счетчик $C(v)$ текущей вершины в случае вставки в дерево нового элемента.
В случае переполнения перестраиваем дерево T_v с учетом нового ключа x, а затем возвращаем результат перестройки, увеличивая счетчик $C(v)$, размер и вес дерева на один, если результат положителен.

В противном случае определяем индекс i потомка, в которого запишется ключ x, и применяем процедуру вставки уже к поддереву T_{v_i}. Возвращаем результат выполнения такой процедуры, увеличивая при этом счетчик $C(v)$, размер и вес дерева при необходимости.

Удаление ключа x производим аналогично вставке:

(1) Проверяем, находится ли ключ в текущей вершине. Если ключ там уже есть и не помечен — помечаем его, увеличиваем счетчик текущей вершины, уменьшаем вес, заканчиваем выполнение процедуры и возвращаем *true*. Если ключ помечен — также заканчиваем процедуру и возвращаем *false*.

(2) Проверяем, не переполнится ли счетчик $C(v)$ текущей вершины в случае удаления из дерева элемента.

В случае переполнения перестраиваем дерево T_v с учетом удаления ключа x, а затем возвращаем результат перестройки, увеличивая счетчик $C(v)$ и уменьшая размер и вес дерева на один, если результат положителен.

В противном случае определяем индекс i потомка, из которого будет удален ключ x, и применяем процедуру удаления уже к поддереву T_{v_i}. Возвращаем результат выполнения такой процедуры, увеличивая при этом счетчик $C(v)$ и уменьшая размер и вес дерева при необходимости.

1.2. Перестройка дерева

Приведем также краткий алгоритм перестройки дерева:

Algorithm 1: Функция `Rebuild`

Data: T — дерево, которое необходимо перестроить; x — ключ, который необходимо вставить/удалить

составляем набор ключей S из непомеченных вершин дерева T;

добавляем/удаляем из набора S ключ x;

строим идеальное IST из набора ключей S;

устанавливаем $C(v) \leftarrow 0$ и соответствующий первоначальный размер для каждого нового дерева.

Пусть S — отсортированный массив ключей будущего дерева размера n. Тогда процесс построения идеального IST будет следующим:

(1) Выбираем $\lfloor \sqrt{n} \rfloor$ равномерно распределенных ключей из массива S и записываем их в массив R новой вершины v_0.

(2) Рассчитываем массив $ID[1 \dots m]$, где $m = \lceil n^{\frac{\alpha}{2}} \rceil$, $\frac{1}{2} \leq \alpha < 1$.

(3) Создаем $\lfloor \sqrt{n} \rfloor + 1$ вершин-потомков и для каждой рекурсивно запускаем операции построения идеального IST из соответствующего подмножества элементов.

Помимо алгоритма построения идеального IST для понимания процесса перестройки важно также описать, как составляется набор ключей из непомеченных вершин дерева. Очевидно, что проще всего такой набор составить с помощью обхода всех вершин дерева поочередно, проверки помеченности их элементов и занесения их в общий массив. Так и поступим:

Algorithm 2: Функция `CompoundRebuildingVector`

Data: *rebuildingElements* — искомый массив элементов для перестройки дерева, *pos* — позиция для вставки элементов текущей вершины в массив *rebuildingElements*, r — массив элементов текущей вершины, *children* — массив потомков текущей вершины

$childrenPos \leftarrow$ Vector(size: Len($children$) + 1);

$childrenPos[0] \leftarrow pos$;

for ($i \leftarrow 1$; $i <$ *Len(children);* $i \leftarrow i + 1$) **do**

  $markBias \leftarrow 0$, если ключ $r[i-1]$ помечен и один в противном случае;

  $childrenPos[i] \leftarrow childrenPos[i-1] +$ Len($children[i-1]$) $+ markBias$;

end

for ($i \leftarrow 0$; $i <$ *Len(r);* $i \leftarrow i + 1$) **do**

  if $r[i]$ *не помечен* **then**

    $rebuildingElements[childrenPos[i+1]-1] = r[i]$;

  end

end

for ($i \leftarrow 0$; $i <$ *Len(children);* $i \leftarrow i + 1$) **do**

  $children[i] \rightarrow$ `CompoundRebuildingVector`(pos : $childrenPos[i]$);

end

return rebuildingElements

Теорема 2. *Пусть μ — это плотность, определенная на конечном отрезке $[a, b]$. Тогда ожидаемая суммарная стоимость обработки последовательности из n μ-случайных вставок и удалений в изначально пустое интерполяционное дерево поиска равняется $O(n \log \log n)$, отсюда ожида-*

емая амортизированная стоимость вставки и удаления — $O(\log\log n)$.

1.3. Операция поиска

Поиск в интерполяционном дереве поиска устроен достаточно просто. Допустим, мы хотим найти элемент с ключом $y \in \mathbb{R}$. Будем использовать значение из массива ID, которое, в силу своего построения, позволит получить достаточно хорошую оценку позиции ключа x в массиве элементов вершины. После чего мы сможем определить точную позицию y при помощи линейного поиска. Приведем более детальное описание:

Пусть T — интерполяционное дерево поиска с максимальным и минимальным элементами a и b соответственно. Пусть также $R[1 \dots k]$ — массив с набором элементов корневой вершины, а $ID[1 \dots m]$ — с приближением обратного распределения элементов корневой вершины. Тогда:

Algorithm 3: Операция поиска элемента по ключу y

$j \leftarrow ID(\lfloor m(y-a)/(b-a) \rfloor)$;
while $y \geq R[j+1]$ **and** $j < k$ **do**
  $j \leftarrow j+1$;
end
рекурсивно запускаем поиск в j-ом поддереве;

Определение 4. *Плотность* μ *называется гладкой для параметра* α, $\frac{1}{2} \leq 1$, *если существуют числа* a, b *и* d *такие, что* $\mu(x) = 0$ *для* $x < a$ *и* $x > b$, *а также для любых* $c_1, c_2, c_3, a \leq c_1 < c_2 < c_3 \leq b$ *и для любых целых* m *и* n, $m = \lceil n^\alpha \rceil$ *справедливо*

$$\int\limits_{c_2-(c_3-c_1)/m}^{c_2} \mu[c_1, c_3](x)dx \leq dn^{-\frac{1}{2}},$$

где $\mu[c_1, c_3] = 0$ *для* $x < c_1$ *или* $x > c_3$ *и* $\mu[c_1, c_3](x) = \mu(x)/p$ *для* $c_1 \leq x \leq c_3$, *где* $p = \int_{c_1}^{c_3} \mu(x)dx$.

Теорема 3. *Пусть* μ *— гладкая плотность, а* T *– случайное IST с параметром* α. *Тогда ожидаемое время поиска элемента в вершине равняется* $O(1)$, *ожидаемое время поиска элемента во всем дереве размера* n *—* $O(\log \log n)$.

Теорема 4. *Время работы поиска элемента в интерполяционном дереве в худшем случае равняется* $O((\log n)^2)$.

Глава 2

Применение набора операций

Опишем теперь процедуру применения не одной операции, а целого набора сразу.

Формально задача выглядит следующим образом: на вход поступает массив из n элементов, каждый из которых представляет собой пару вида *(ключ, тип операции)*, где тип операции — это вставка, удаление или поиск. При этом для удобства мы будем считать, что массив отсортирован по возрастанию ключей и каждый ключ в нем встречается ровно один раз.

Пусть $w[1], w[2], ..., w[l]$ — массив из l отсортированных пар вида *(ключ, тип операции)*. Тогда, опуская детали, процесс выполнения операций будет следующим:

(1) Проверяем, переполнится ли счетчик $C(v)$ для текущей вершины v в случае выполнения m операций в поддереве T_v, где m — число модифицирующих дерево операций (вставка и удаление) среди поступившего на вход массива. В случае переполнения перестраиваем текущее поддерево и завершаем выполнение процедуры.

(2) Проходимся по всем элементам массива w и проверяем, не находится ли рассматриваемый ключ в текущей вершине. В случае, если поиск ключа в вершине выполнен успешно, можем выполнить соответствующую ключу операцию и сохранить ее результат. В противном случае, вычисляем индекс вершины-потомка, в которую будем передавать для выполнения текущий элемент, и записываем текущий элемент в массив действий найденного дерева-потомка.

(3) Для всех вершин-потомков, чьи массивы действий оказались непусты, запускаем процедуру рекурсивно.

При переходе от алгоритма с обработкой одной операции к алгоритму с обработкой нескольких возникает ряд вопросов: как возвращать массив результатов выполнения операций? Как поддерживать размер и вес вершин? Ответы на них мы и постараемся дать в этой главе.

Прежде всего разберемся с тем, как передавать и формировать массив результатов операций. Для этого при формировании массива операций помимо ключа и типа операции будем хранить позицию этой операции в отсортированном массиве. Так, каждый элемент массива w теперь имеет вид *(ключ, тип операции, позиция элемента в массиве)*. При этом важно отметить, что позиция элемента отражает его индекс сразу после формирования исходного массива операций и не меняется после инициализации. Кроме того, будет удобно также перед началом выполнения операций создать неизменяемый массив p префиксных сумм, i-й элемент которого содержит число модифицирующих операций (добавления и удаления) среди первых $i + 1$ элементов. Впоследствии это поможет нам быстро принимать решения о необходимости перестройки рассматриваемого поддерева.

Благодаря хранению с каждым элементом массива операций его позиции в исходном массиве, мы очевидным образом решаем проблему с формированием массива результатов выполнений этих операций — создаем массив результатов операции (с размером равным количеству этих операций) и при каждом рекурсивном вызове функции выполнения набора операций передаем в качестве ее аргументов помимо самих операций еще и массив результатов. Теперь, при выполнении операции мы знаем ее позицию в массиве результатов, а, значит, можем без проблем сохранить результат ее работы.

Вернемся теперь к вопросу о том, как выполнять поддержку размера и веса каждой из вершин. Для реализации этого предлагается в качестве результата работы функции для вершины v возвращать число вставленных и удаленных из поддерева T_v элементов. После чего размер вершины задается как разность суммарного количества вставленных и удаленных элементов среди всех потомков вершины, а вес как их сумма.

Глава 3

Базовые концепции параллельных вычислений

Прежде чем говорить о том, как распараллелить работу приведенного алгоритма интерполяционного дерева поиска, опишем базовые инструменты параллельных вычислений.

3.1. Fork-join вычисления

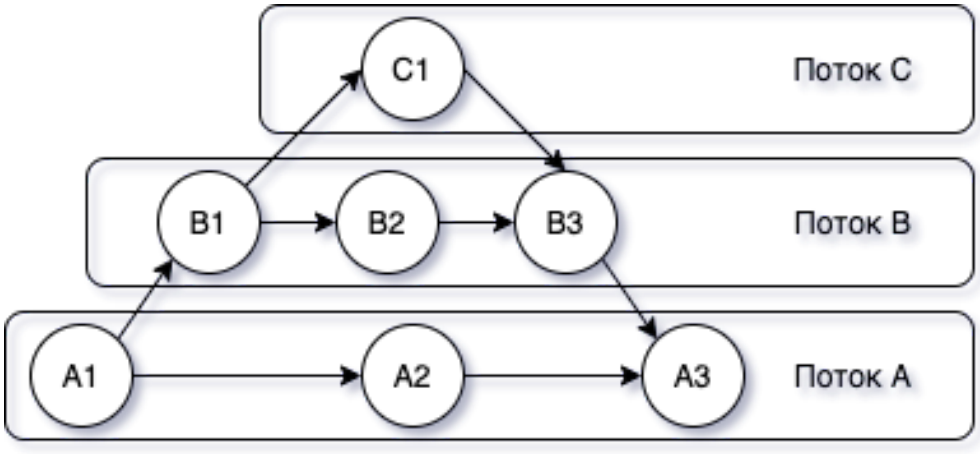
Большой класс различных многопоточных вычислений может быть обобщен благодаря более структурированному подходу, в котором потоки ограничены в способах синхронизации с другими потоками. Один из таких механизмов, который мы и используем в данной работе – это *fork-join* вычисления [2], где *fork* означает возможность «раздвоить» поток (т.е. с определенного момента работы программы запустить вычисления в двух потоках параллельно), а *join* — синхронизировать параллельные потоки. При этом *join* — единственный механизм синхронизации потоков.

3.2. Work и Span

Параллельные вычисления могут быть представлены в виде ориентированного графа DAG (Directed Acyclic Graph), в котором каждая вершина представляет собой выполнение инструкции *fork* или *join* (Рис. 3.1). Вершины DAG со степенью исхода два обозначают выполнение операции *fork*, вершины со степенью захода два — операции *join*.

Для оценки эффективности и производительности многопоточных программ используются разные функции оценки стоимости алгоритмов,

Рис. 3.1. Пример DAG



однако одними из самых важных являются *work* и *span*.

Определим *work* алгоритма как количество вершин в DAG, а *span* — как длину наиболее протяженного пути в таком графе.

Механизм *fork-join* позволяет реализовать множество повсеместно используемых алгоритмов, самые важные из которых подробно описаны в книге Umut A. Acar «Parallel Computing: Theory and Practice» [2]. Далее мы рассмотрим принципы работы параллельных алгоритмов, которые будут использоваться непосредственно в данной работе.

3.3. Parallel for

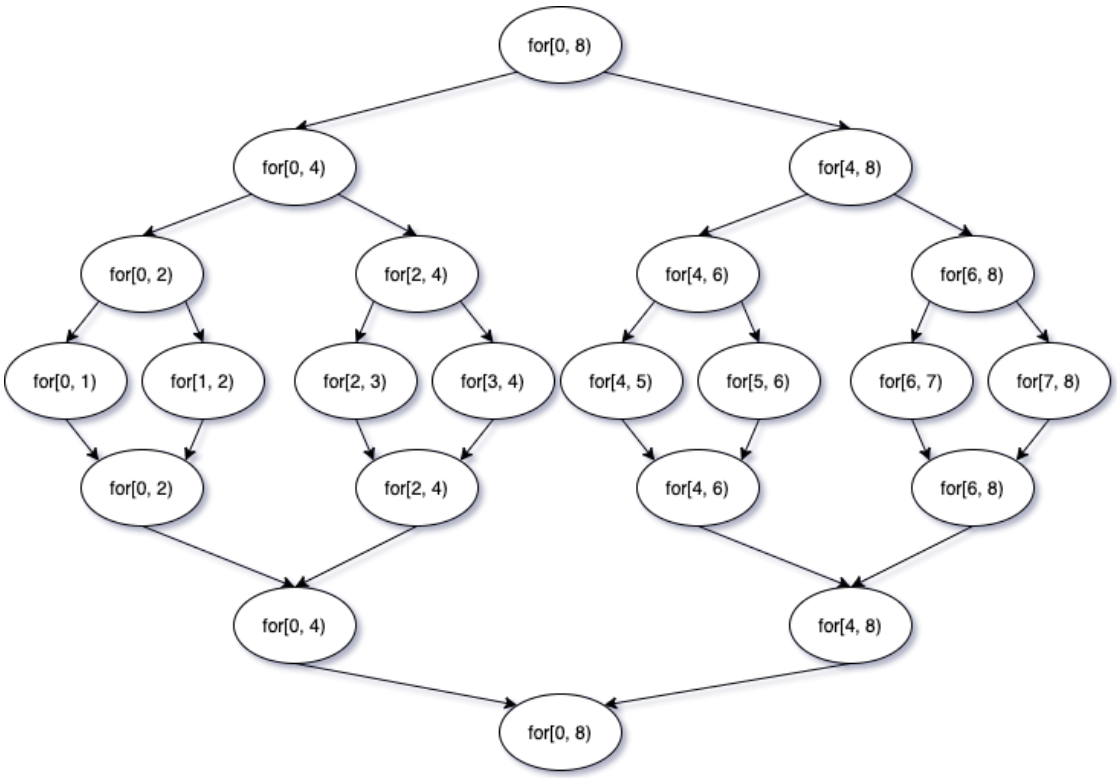
Первый и наиболее используемый в работе — алгоритм параллельной итерации по массиву (параллельный аналог цикла *for*).

Листинг 3.1. Реализация *parallel for* при помощи *fork2*

```
1 parallel_for(int l, int r) {
2     fork2([&] {
3         parallel_for(l, (l + r) / 2);
4     }, [&] {
5         parallel_for(l + r) / 2, r);
6     });
7 }
```

Для того, чтобы реализовать параллельный цикл *for* будем разбивать отрезок итерации на две равные части и запускать выполнение функции рекурсивно от каждой из этих частей в двух параллельных процессах. Поступаем так до тех пор, пока величина длина отрезка разбиения не окажется равной единице. После этого выполняем *join* процессов в порядке обратном тому, в котором выполнялся *fork*.

Рис. 3.2. DAG для *parallel for* на множестве $[0 \dots 7]$



Из Рис. 3.2 легко увидеть, что *work* параллельного цикла for равен $O(n)$, а *span* — $O(\log n)$.

3.4. Scan

Определим сканирование как отображение вида: $[x_0, x_1, \dots, x_{n-1}] \to [x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_{n-1}]$. Так, например, $Scan([1, 2, 3, 4, 5]) = [1, 3, 6, 10, 15]$.

Span-эффективная версия алгоритма сканирования была предложена В. Хиллсом и Г. Стилом в статье «Data parallel algorithms»[3], ее *span* равен $O(\log n)$, а *work* — $O(n)$, где n — число элементов сканируемого массива.

3.5. Filter

Последняя параллельная операция, которую мы рассмотрим — операция, копирующая элементы из указанного массива в новый согласно заданной предикатной функции.

Пусть $a[1..n]$ — массив элементов из множества X, а $f : X \to [0, 1]$ — предикатная функция, определенная на множестве X. Тогда:

Algorithm 4: Функция `Filter`

Data: $A[1 \dots n]$ — массив чисел размера n, f — предикатная функция

parallel for $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ **do**
  $B[i] \leftarrow f(A[i])$;
end
$B \leftarrow Scan(B)$;
Создаем массив C размера $B[n]$;
if $B[1] = 1$ **then**
  $C[1] \leftarrow A[1]$;
end
parallel for $i \leftarrow 2; i \leq n; i \leftarrow i + 1$ **do**
  if $B[i] \neq B[i - 1]$ **then**
    $C[B[i]] \leftarrow A[i]$;
  end
end
return C;

Алгоритм фильтрации построен на последовательных операциях параллельного цикла for и сканирования, каждая из которых имеет *span* равный $O(\log n)$ и *work* $O(n)$. Таким образом, суммарные *span* и $work$ алгоритма фильтрации равняются $O(\log n)$ и $O(n)$ соответственно.

Глава 4

Параллельное IST

Поговорим теперь о том, как распараллелить работу интерполяционного дерева поиска. Для того, чтобы это осуществить, определим места алгоритма, в которых выполнение команд можно осуществлять независимо друг от друга, т.е. так, что ни в какой момент времени они не используют для записи общую память или результаты работы друг друга.

Рассмотрим функцию применения набора операций, состоящую из трех логических частей, описанных в Главе 2, для каждой из которых мы предложим параллельное решение.

4.1. Перестройка дерева

На этапе перестройки дерева, после получения всех хранящихся в нем элементов, на их основе выполняется построение нового, идеального дерева. Пусть $x[1...n]$ — массив элементов нового дерева, составление которого подробно разобрано в описании алгоритма (Алгоритм 2) работы функции `CompoundRebuildingVector`. Тогда выберем первые $\sqrt{n}$ равномерно распределенных элементов массива, запишем их в корневую вершину идеального дерева, а затем рекурсивно запустим построение потомков текущей вершины. При рекурсивном запуске функции в качестве массивов элементов мы передаем отрезки массива x, находящиеся между двумя ближайшими элементами массива, сохраненными в текущую вершину. То есть если обозначить элементы вершины как $x_{k_1}, x_{k_2}, \ldots, x_{k_z}$ (где $z = \lfloor\sqrt{n}\rfloor$), то в первого потомка мы будем передавать элементы $x_1, \ldots, x_{k_1-1}$, во второго — $x_{k_1+1}, \ldots x_{k_2-1}$ и т.д.

Опишем формирование массива элементов вершины и поддеревьев-потомков:

(1) Подготавливаем данные для фильтрации. Для этого заполняем новый массив y размера n так, что для всех i, для которых $x[i]$ — элемент вершины, $y[i] = -i$, для всех остальных значений i: $y[i] = i$.

Algorithm 5:

Data: $y[1 \dots n]$ — массив размера n, $start$ — индекс первого элемента рассматриваемого отрезка массива, end — последнего

$s \leftarrow \lfloor \sqrt{n} \rfloor$;

parallel for $i \leftarrow start; i \leq end; i \leftarrow i + 1$ **do**

  if $i >= start + s$ ***and*** $(i - start - s) \mod s = 0$ **then**

    $y[i] \leftarrow -i$;

  else

    $y[i] \leftarrow i$;

  end

end

(2) Выполняем фильтрацию элементов массива y, в качестве предикатной функции используем $z < 0$, записываем результат в массив ids. После выполнения указанных операций в массиве ids мы имеем индексы тех элементов, которые впоследствии будут сохранены в текущую вершину.

(3) Создаем новый массив r того же размера, что и ids, и заполняем его в цикле *parallel for* по следующему принципу: $r[i] \leftarrow -x[ids[i]]$. Теперь массив r хранит в себе те и только те элементы, которые будут записаны в текущую вершину.

(4) Создаем массивы индексов начал и концов отрезков массива x, которые содержат в себе элементы будущих поддеревьев.

Algorithm 6:

Data: $start\_ids[1 \dots n]$ — новый массив размера n,
$y[1 \dots n]$ — массив размера n, составленный на этапе (1),
$start$ — индекс первого элемента рассматриваемого отрезка массива, end — последнего

$s \leftarrow \lfloor \sqrt{n} \rfloor$;
parallel for $i \leftarrow start; i \leq end; i \leftarrow i + 1$ **do**
    if $y[i] > 0$ ***and*** $(i = start$ ***or*** $y[i-1] < 0)$ **then**
        $start\_ids[i] \leftarrow i$;
    else
        $start\_ids[i] \leftarrow 0$;
    end
end

С изменением условий $y[i-1] < 0$ на $y[i+1] < 0$ и $i = start$ на $i = end$, но в остальном полностью аналогично, заполняем массив индексов концов отрезков end_ids.

(5) Фильтруем элементы массивов $start\_ids$ и end_ids, в качестве предикатной функции используем $z > 0$, записываем результат в массивы $children\_starts$ и $children\_ends$ соответственно. Так, элементы первого потомка текущей вершины содержатся между элементами с индексами $children\_starts[1]$ и $children\_ends[1]$, второго — между $children\_starts[2]$ и $children\_ends[2]$ и т.д.

(6) Последним шагом рекурсивно запустим функцию `Rebuild` для всех отрезков с принадлежащими поддеревьям элементами, полученных на предыдущих этапах.

Algorithm 7:

Data: $children\_starts[1 \dots K]$, $children\_ends[1 \dots K]$ — массивы индексов концов и начал отрезков массива x, принадлежащих соответствующим поддеревьям

parallel for $i \leftarrow 1; i \leq K; i \leftarrow i + 1$ **do**

создаем новую вершину c;

$c \rightarrow Rebuild(x, children\_starts[i], children\_ends[i])$;

end

После этого остается только сохранить указатели на созданные вершины потомков в текущей вершине.

4.2. Поиск элементов по текущей вершине

Вспомним алгоритм выполнения набора операций. Прежде, чем вызывать функцию выполнения операций рекурсивно от потомков, необходимо для каждой операции проверить, находится ли целевой для операции элемент в текущей вершине. Все операции, элементы которых найдены в вершине не были, нужно распределить в соответствии со значениями их ключей по вершинам-потомкам.

Будем руководствоваться той же идеей, что и с перестройкой дерева. Пусть x — набор операций, тогда алгоритм поиска выглядит следующим образом:

(1) Создаем новый массив y размера m (число операций в наборе). В цикле *parallel for* проходимся по всем элементам массива операций. В случае, если элемент операции $x[i]$ находится в текущей вершине — выполняем операцию и указываем $y[i] \leftarrow 0$, в противном случае $y[i] \leftarrow i$.

(2) Создаем два массива $start\_ids$ и end_ids длины m и заполняем их по принципу, описанному в секции про перестройку дерева.

(3) Выполняем фильтрации массивов $start\_ids$ и end_ids, используя в качестве предиката функцию $z > 0$. После этого в массивах $starts$ и $ends$ у нас окажутся начала и концы отрезков массива x между двумя ближайшими элементами, найденными в вершине.

В качестве результата алгоритма имеем массивы $starts$ и $ends$, которые впоследствии будут использованы для идентификации наборов операций, выполняемых над потомками текущей вершины. Так, операции $x[starts[1]], \ldots, x[ends[1]]$ будут переданы для обработки первым потомком, $x[starts[2]], \ldots, x[ends[2]]$ — вторым и т.д.

4.3. Рекурсивные вызовы функции `pExecute`

После всех описанных выше этапов остается только запустить рекурсивное выполнение функции применения наборов операций `pExecute` от каждого из потомков текущей вершины, для которых были установлены непустые наборы операций.

Algorithm 8:

Data: *children* — массив потомков текущей вершины, *actions* — массив операций, выполняемых над соответствующим потомком, $starts[1 \dots K]$, $ends[1 \dots K]$ — массивы индексов концов и начал отрезков массива x, принадлежащих соответствующим поддеревьям, p — массив префиксных сумм, *results* — массив размера, равного размеру массива операций, выполняемых над всем деревом

parallel for $i \leftarrow 0; i < K; i \leftarrow i + 1$ **do**

$children[y[starts[i]]] \rightarrow$

$pExecute(actions, starts[i], ends[i], p, results)$;

end

Завершающим шагом алгоритма проходим по всем результатам вызова функции и увеличиваем вес и размер текущего дерева на соответствующие значения.

Отметим, что в качестве оптимизации приведенных параллельных алгоритмов имеет смысл передавать с каждым вызовом функции `pExecute` также указатели на массивы $start\_ids$, end_ids и y, созданные при первом вызове функции. То же самое применимо и к функции `Rebuild`.

4.4. Work и Span алгоритма

Приведем теперь оценки *work* и *span* описанного в этой главе алгоритма.

Будем считать, что операции расположены на одном уровне рекурсии, если выполняющие их функции были вызваны спустя равное число

рекурсивных вызовов после начального вызова функции. Тогда на каждом уровне рекурсии выполняется $O(1)$ вызовов функции `Filter` и операций *parallel for*, каждая из которых имеет *span* равный $O(\log m)$, где m — размер набора операций (вставки, удаления и поиска), производимыми над IST. Кроме того, суммарное число уровней рекурсий равно глубине интерполяционного дерева поиска — $O(\log \log n)$. Таким образом получаем, что *span* алгоритма выполнения набора операций над IST равен $O(\log m \log \log n)$.

Для того, чтобы посчитать *work*, необходимо заметить, что каждый уровень рекурсии порождает $O(m)$ новых вершин DAG. Отсюда делаем вывод, что *work* алгоритма равняется $O(m \log \log n)$.

Глава 5

Эксперименты

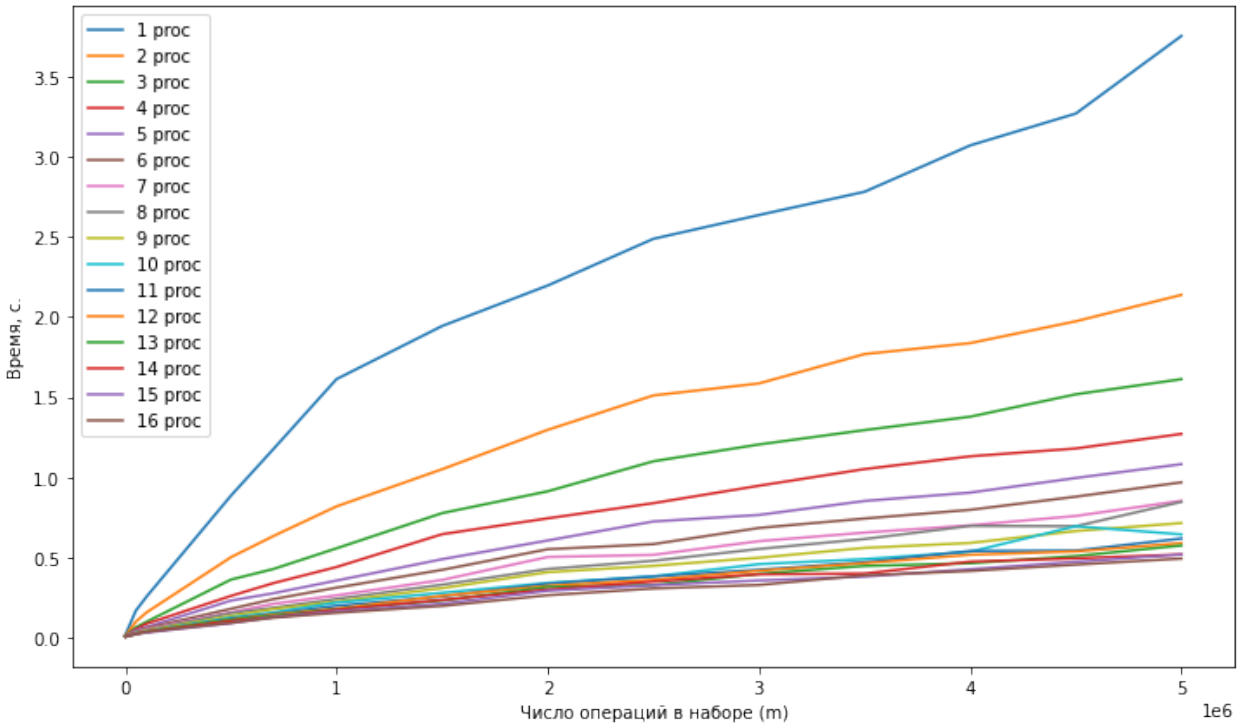
Для реализации структуры дерева поиска с параллельным набором операций был выбран язык C++ и использована библиотека *pctl* [4], содержащая имплементации основных параллельных структур и алгоритмов, из которой в работе широко использовались функции `pctl::parallel_for`, `pctl::filter` и др.

Тестирование производилось на сервере с процессором Intel Xeon Gold 6230 с 16 доступными ядрами и 30 Гб оперативной памяти. На сервере была установлена ОС Ubuntu 20.04.1 LTS. Для оптимизации динамического времени выделения новой памяти было принято решение использовать *tcmalloc* [5] — разработанную компанией Google имплементацию оператора `new`, позволящую эффективно работать с памятью в параллельных программах. В качестве компилятора был выбран OpenCilk 1.0 [6], компилирующий программы с использованием Cilk эффективнее, чем другие существующие компиляторы для паралелльных языков программирования и их расширений.

Прежде всего был произведен замер времени выполнения параллельной программы в зависимости от размера набора производимых операций при разном количестве используемых процессоров (Рис. 5.4).

Все операции выполнялись на деревьях, составленных с идентичными элементами, ключи которых являются целыми числами на отрезке $[0 \dots 5{\cdot}10^7]$, включенными в дерево с вероятностью $1/2$. Таким образом, размер таких деревеьев приблизительно равен $2.5{\cdot}10^7$. При составлении набора операций каждому целому числу из отрезка $[0, \dots, m-1]$ сопоставлялась одна из операций `Insert`, `Remove` и `Contains`, выбранная с вероятностью $1/3$.

Рис. 5.1. Зависимость времени выполнения набора операций от числа операций



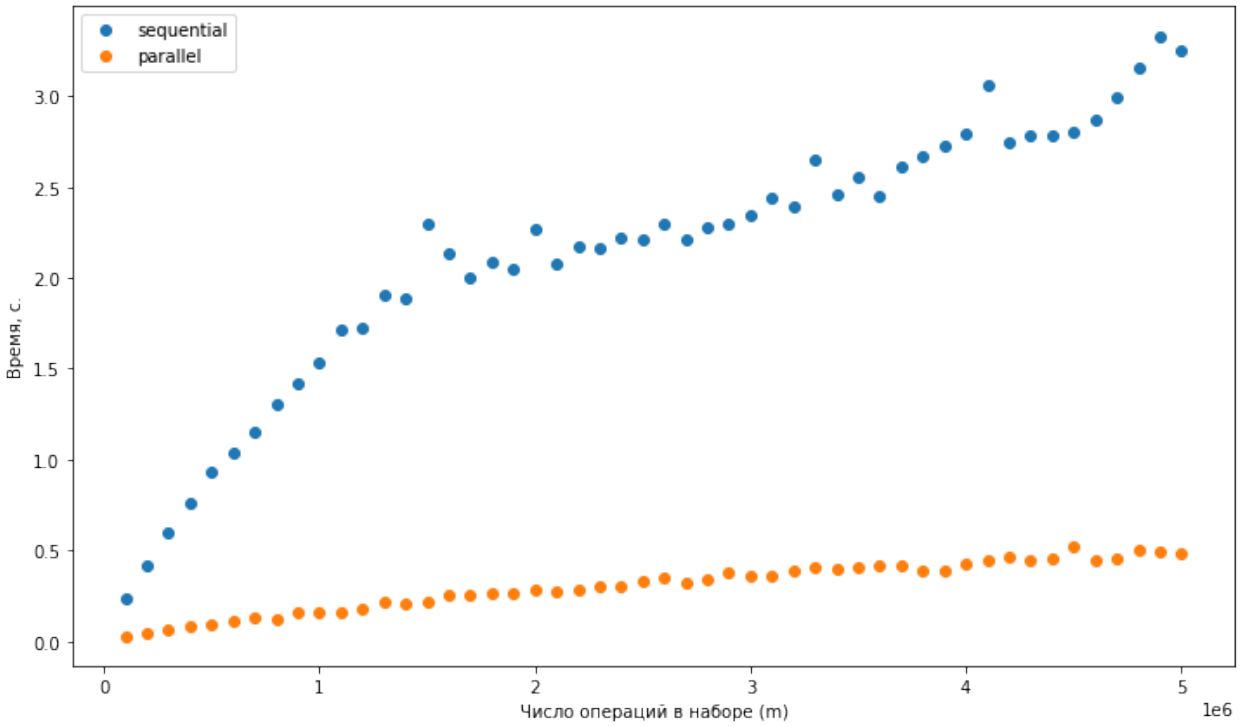
Из Рис. 5.2 четко видно преимущество объединений операций в наборы с последующим их распараллеливанием.

Далее был произведен замер времени выполнения параллельной программы в зависимости от размера исходного дерева (Рис. 5.3) при разном количестве используемых процессоров. Для эксперимента были использованы наборы операций из 10^6 элементов, составленные по описанным выше принципам.

Также было произведено сравнение времени выполнения набора операций в последовательном и параллельном случаях, где под последовательным случаям подразумевается вставка в дерево элементов по одному с последующей перестройстройкой дерева по мере надобности.

Далее была найдена зависимость времени выполнения набора из $m = 10^6$ операций от числа используемых процессоров для дерева размером $n = 5 \cdot 10^7$ (Рис. 5.4).

Рис. 5.2. Сравнение времени выполнения набора операций в последовательном и параллельном случаях



На Рис. 5.5 изображена зависимость ускорения выполнения работы функции (отношение времени работы функции на N процессорах к работе на одном) от числа процессоров. Из рисунка графика видно, что приведенная реализация алгоритма имеет хорошую масштабируемость. Так, на 16 процессорах удалось добиться ускорения более, чем в 10 раз по сравнению с работой на одном процессоре. Кроме того, из приведенных графиков можно сделать вывод, что предел масштабируемости алгоритма в соответствии с законом Амдала на 16 процессорах не достигнут, т.е. при увеличении количества процессоров программа продолжит свое ускорение.

Рис. 5.3. Зависимость времени выполнения набора операций от размера дерева

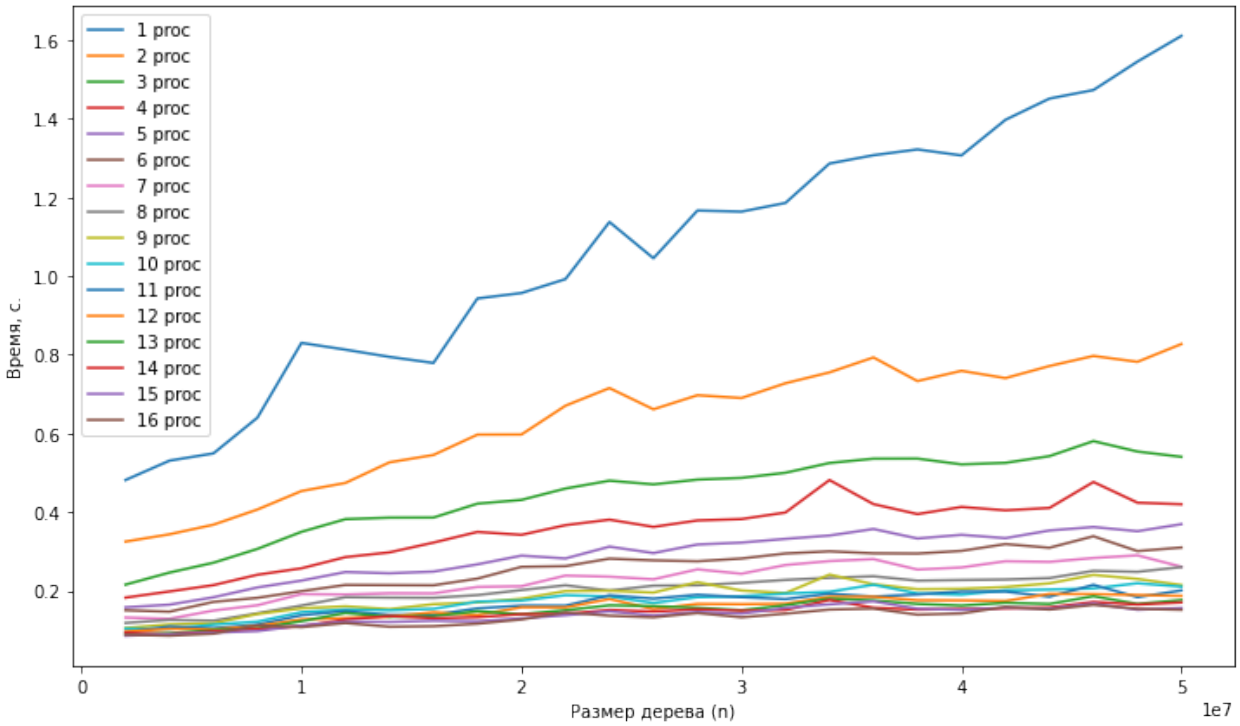


Рис. 5.4. Зависимость времени выполнения набора операций от числа процессоров

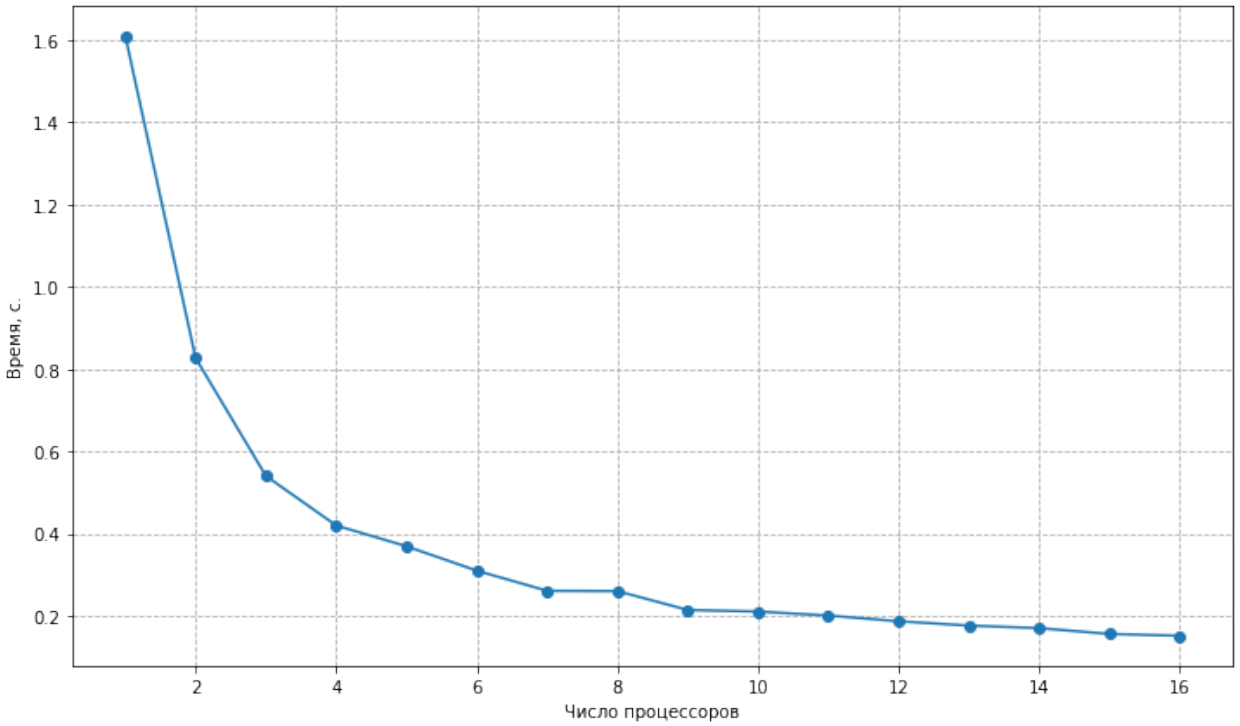
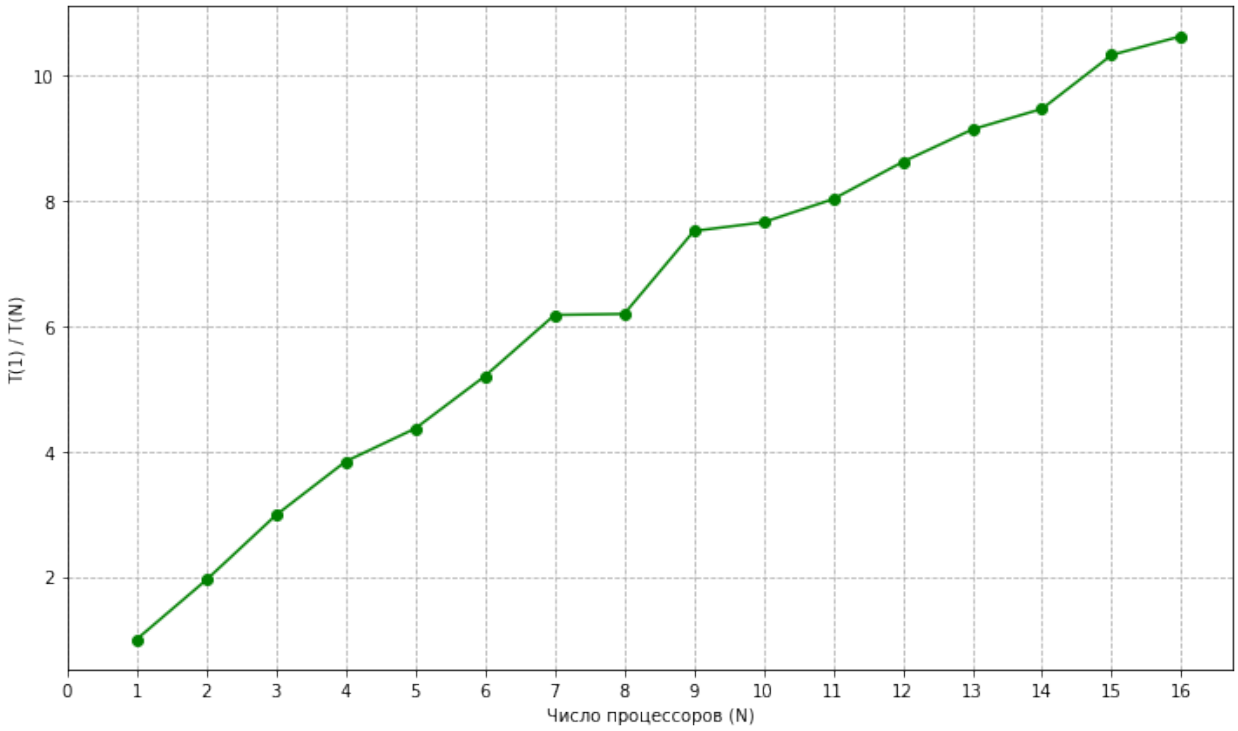


Рис. 5.5. Зависимость ускорения выполнения набора операций от числа процессоров



Заключение

В работе была подробно описана структура интерполяционного дерева поиска, а также представлено ее эффективная оптимизация для работы с параллельным применением набора операций вставки, удаления и поиска элементов по дереву. В экспериментах показано, что выполнение операций в такой структуре данных хорошо поддается масштабированию, а также значительно ускоряет общую работу программы в сравнении с последовательной реализацией.

Полученные результаты доказывают большой потенциал использования описанного дерева поиска при реализации индексов в базах данных, а так же других инструментах с необходимостью быстрого параллельного применения заранее заданного набора операций.

Список литературы

[1] Kurt Mehlhorn, Athanasios Tsakalidis. «Dynamic interpolation search». *Automata, Languages and Programming*. Springer-Verlag, 1985, с. 424—434. DOI: 10.1007/bfb0015768. URL: https://doi.org/10.1007/bfb0015768.

[2] Umut A. Acar. *Parallel Computing: Theory and Practice*. 2016. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html.

[3] W. Daniel Hillis, Guy L. Steele. «Data parallel algorithms». *Communications of the ACM* **29** 12 (дек. 1986), с. 1170—1183. DOI: 10.1145/7902.7903. URL: https://doi.org/10.1145/7902.7903.

[4] Deepsea Project. *pctl*. 2015. URL: https://github.com/deepsea-inria/pctl.

[5] Google. *tcmalloc*. URL: https://github.com/google/tcmalloc.

[6] *OpenCilk*. URL: https://github.com/OpenCilk.